Compile-time Safety and Runtime Performance in Programming Frameworks for Distributed Systems

LARS KROLL

Doctoral Thesis in Information and Communication Technology
School of Electrical Engineering and Computer Science
KTH Royal Institute of Technology
Stockholm, Sweden 2020

School of Electrical Engineering
and Computer Science
KTH Royal Institute of Technology
SE-164 40 Kista
SWEDEN

TRITA-EECS-AVL-2020:13
ISBN: 978-91-7873-445-0

Akademisk avhandling som med tillstånd av Kungliga Tekniska Högskolan framlägges till offentlig granskning för avläggande av teknologie doktorsexamen i informations- och kommunikationsteknik på fredagen den 6 mars 2020 kl. 13:00 i Sal C, Electrum, Kungliga Tekniska Högskolan, Kistagången 16, Kista.

Printed by Universitetsservice US-AB

Abstract

Distributed Systems, that is systems that must tolerate partial failures while exploiting parallelism, are a fundamental part of the software landscape today. Yet, their development and design still pose many challenges to developers when it comes to reliability and performance, and these challenges often have a negative impact on developer productivity. Distributed programming frameworks and languages attempt to provide solutions to common challenges, so that application developers can focus on business logic. However, the choice of programming model as provided by a such a framework or language will have significant impact both on the runtime performance of applications, as well as their reliability.

In this thesis, we argue for programming models that are statically typed, both for reliability and performance reasons, and that provide powerful abstractions, giving developers the tools to implement fast algorithms without being constrained by the choice of the programming model. Furthermore, we show how the design of Domain Specific Languages (DSLs) for such distributed programming frameworks, as well as DSLs for testing these components written in them, can reduce the mental gap between theoretical algorithms and their implementation, in turn reducing the risk of introducing subtle bugs.

Concretely, we designed and implemented two different versions of the Kompics Component Model. The first is called Kompics Scala and focuses on pattern matching of events, as well as programming ergonomics and similarity to theoretical algorithms. The second version is called Kola and is a language with a compiler of its own, focusing on compile-time safety. Finally, we present a third framework, called Kompact, implementing a hybrid Actor–Component model which is designed around runtime-performance and static typing, and is implemented in the Rust language. In order to compare our solutions to the state-of-the-art, we present the first cross-language, distributed, message-passing benchmarking suite. We evaluated the impact of network abstractions in these models on performance, and show that our approach offers between 2 × and 5 × improvements on throughput for certain applications. We also evaluated the performance tradeoffs of different message-passing models and different implementations of the same model. We show that all our implementations are competitive, and our hybrid model, in particular, has significant benefits in a wide range of scenarios, leading to improvements of up to 27 × compared to the state-of-the-art of message-passing systems.

Sammanfattning

Distribuerade system, det vill säga system som tolererar partiella fel medan parallelism utnyttjas, är en grundläggande del av landskapet för mjukvaruutveckling idag. Ändå utgör dessas utveckling och design fortfarande många utmaningar för utvecklare när det gäller tillförlitlighet och prestanda, och dessa utmaningar har ofta en negativ inverkan på utvecklarens produktivitet. Distribuerade programmeringsramverk och -språk försöker att ge lösningar på vanliga utmaningar, så att applikationsutvecklare kan fokusera på applikationslogik. Sådant ramverk eller språk är baserad på en programmeringsmodell, vars val har en betydande inverkan både på applikationernas runtime-prestanda, liksom dessas tillförlitlighet.
I den här avhandlingen argumenterar vi för programmeringsmodeller som är statiskt typade, både av tillförlitlighets- och prestandaskäl, och som ger kraftfulla abstraktioner, vilket ger utvecklarna verktyg för att implementera snabba algoritmer utan att begränsas av valet av programmeringsmodell. Dessutom visar vi hur utformningen av domänspecifika språk (DSL) för sådana distribuerade programmeringsramverk, liksom DSL:er för att testa dessa komponenter skrivna i dem, kan minska den mentala klyftan mellan teoretiska algoritmer och dessas implementation, vilket i följd minskar risken för introduktion av subtila buggar.
Specifikt, har vi utformat och implementerat två olika versioner av Kompics Component modellen. Den första heter Kompics Scala och fokuserar på mönstermatchning av event, samt programmeringsergonomi och likhet med teoretiska algoritmer. Den andra versionen heter Kola som är ett språk och sin egen kompilator med fokus på säkerhet i kompileringstid. Slutligen presenterar vi ett tredje ramverk, kallad Kompact, som implementerar en hybrid Actor-Component modell. Ramverket är utformad kring runtime-prestanda och statisk typning, och är implementerad i språket Rust.
För att jämföra våra lösningar med andra state-of-the-art lösningar, presenterar vi den första flerspråkiga, distribuerade, message-passing benchmarking-sviten. Vi utvärderade effekterna av nätverksabstraktioner i dessa modeller på prestanda och visar att vår strategi erbjuder mellan 2 × till 5 × förbättringar av kapaciteten för vissa applikationer. Vi utvärderade också prestanda-tradeoffs för olika message-passing modeller och olika implementationer av samma modell. Vi visar att alla våra implementationer är konkurrenskraftiga, och särskilt att vår hybridmodell har betydande fördelar i ett brett spektrum av scenarion, vilket leder till förbättringar på upp till 27 × jämfört med den mest kända state-of-the-art message-passing system.

Acknowledgements

Like the proverbial Rome, a doctoral dissertation is not built (or written) in a day, and certainly not by a single person alone. Over the years in which the work, that finally resulted in this dissertation, took place, many people have shared ideas with me, let me bounce my own ideas off of them, pointed out challenges, fixed issues, showed me possible directions for progress or inquiry, and supported me in a myriad of other ways.

First and foremost among the guides and supporters is my main advisor, *Seif Haridi*, who time and again has listened to all the wild ideas running around in my head and then subsequently helped me shape them into something that can actually be called "research". Without him, I would likely still be writing lines upon lines of code in my office and this dissertation would simply be a dream for a far off future "when I am done". In those situations when it did happen that I got badly stuck, with no clue where to go from there, it was usually the bursting fountain of ideas that is my secondary supervisor, *Jim Dowling*, who sparked the idea that led to the next step or goal. And, of course, it was also Jim who gave me the opportunity and convinced me to start a PhD in the first place. Thus, this dissertation would also have been equally impossible without him. For this, as well as every inspiring conversation, every boat trip, and every lunch, dinner, and beer we have shared over the past years, I am profoundly grateful.

Over an even greater number of lunches, dinners, beers, coffees, whiskies, and pipes my friends, colleagues, and occasional house and office mates — *Paris Carbone* and *Alex Ormenișan* — have acted as sounding boards for silly ideas, sources of inspiration and wild dreams, as testers, fixers, and collaborators, or simply as much needed stress relief and emotional support. Above everyone else, they helped me to get through the last few years and maintain some semblance of sanity and for that I cannot be thankful enough. Much the same holds true, of course, for my more remote friends, who have turned all my vacations from "working from a different location" to a joyful and re-energising experience and many of which share my favourite hobby of playing TTRPGs with me every single week: *Chrischi*, *Ercan*, *Ivy*, *Jakob*, *Jenny*, *Malte*, *Olga*, *Rain*, *Sergio*, and *Teri*.

Of course, I could not have made it through all these years of education — or life, for that matter — without the support of my *parents*, *grandparents*, and my *sister*. They were always there, whether I needed advice or help, a place to stay for a few weeks, extra funds, or simply a lovely meal and a pleasant conversation. I may have spent a few more years in education than either of us bargained for, but I am immensely grateful to my family for not just putting up with it, but even encouraging me to go further.

I would also like to thank all the master's students who worked on bringing our shared ideas to life over the years: *Ifeanyi Ubah* has been integral in the development of the Kompics Testing framework and I have greatly enjoyed our extremely fruitful brainstorming sessions. *Johan Mickos* has been responsible for Kompact's original networking implementation and his work is being continued

now by *Adam Hasselberg*. Both of them are smart and driven individuals, who, I can only say, are and have been a pleasure to work with. *Harald Ng* has put countless hours of work into the benchmarking suite, and without his invaluable help the suite would not be as large as it is today. Finally, *Max Meldrum* and *Klas Segeljakt* have provided me with feedback, pointed out and also fixed issues, and been great company over many a beer. As both of them have started their own PhD studies by now, I wish them the best of luck on their journey to the point where they get to write acknowledgements in their own dissertations.

There is such a long list of colleagues from KTH, RISE, and beyond who over the years have helped me in a great variety of ways, that it is impossible to name them all here. Thus, even though I must cherry-pick only a handful of people, I wish to thank every single person who has given me feedback or advice, who has made good things happen or saved me from doing something stupid, who has taught me skills and given me insights, and everyone who has shared their ideas and listened to mine. That being said, the following individuals require some special consideration: *Cosmin Arad*, who designed Kompics, got me involved with it in the first place. Everything in this dissertation rests on his prior work and our discussions and collaboration around it. Of course, Cosmin also told me not to do another thesis on Kompics, and yet here we are. Needless to say, I appreciate advice, even if I do not actually listen to it. Furthermore, I would like to thank *Christian Schulte* for not only teaching me practically everything I know about compilers, but also for always being there to answer my many questions about the PhD process, giving me advice about publications and research direction, and always pointing out a connection to other work I would have otherwise overlooked. I am also grateful to *Sverker Janson*, not only for letting me sit in SICS/RISE offices during all of my PhD, but more importantly for taking the time every single year to have a chat about how I was doing and where I wanted to be going. I greatly enjoyed those talks and I am looking forward to the next one, now that I am actually an employee. For many great conversations and for indulging all my silly questions, I would like to thank *Gabriel Hjort Blindell*, *Ian Marsh*, *Mahmoud Ismail*, and *Roberto Castañeda Lozano*. *Vladimir Vlassov* has been a steady source of advice and insight over the years, and also the quality assurance reviewer for this dissertation. *Philipp Haller* has been the reviewer for my thesis proposal seminar, but more importantly he has been a steady presence at every programming language conference I went to and has introduced me to many interesting people. He has always made me feel like I belong in that community, even though my background is not in programming languages, and for that I am immensely grateful. Finally, I would like to thank *Rodrigo Caballero* from Stockholm University and *Matthew Huber* from Purdue University for conspiring to get me access to Purdue's Halstead compute cluster, which allowed me to run a series of experiments that would have otherwise been impossible.

Finally, two additional people have contributed significantly to the finalisation of this document itself: Adam O'Brien let me stay at his beautiful place in Auckland, New Zealand, while I was writing the first draft of my dissertation, inspired by the stunning natural beauty around me. And Ren Atkins took upon herself the

grave task of reading through the whole 250-odd pages of this document and exterminating any and all mistakes in my English grammar or spelling. This dissertation would certainly not be the same without their contributions.

Contents

List of Figures **xii**

List of Tables **xiv**

List of Algorithms **xvi**

List of Listings **xix**

1 Introduction **2**
1.1 Motivation 4
1.2 Design Principles 5
1.3 Primary Contributions 5
1.4 Previously Published Work 6
1.5 Dissertation Outline 8

2 Background **10**
2.1 The Actor Model 12
2.2 The Kompics Component Model 18

3 From Algorithm to Implementation **26**
3.1 The Kompics Scala DSL 32
3.2 Kompics Scala Implementation 41
3.3 Evaluation 52
3.4 Related Work 56

4 Preventing Bugs at Compile Time **58**
4.1 Common Mistakes in Kompics Java 61
4.2 Typing Rules 64
4.3 Typed Kompics 68
4.4 The Kola DSL 69
4.5 Kola Implementation 78
4.6 Evaluation 80
4.7 Related Work 83

5 Unit Testing Message-Passing Systems **86**
5.1 Background 90
5.2 The Language of Event-Streams 92
5.3 KompicsTesting 102
5.4 Example — A Chat Application 104

5.5 Evaluation . . . 106
5.6 Related Work . . . 109

6 Benchmarking Message-Passing Systems 112
6.1 A Distributed Benchmarking Framework . . . 115
6.2 Benchmarks . . . 124
6.3 Frameworks and Languages . . . 136
6.4 Results . . . 138

7 The Power of Network Abstractions 156
7.1 Kompics Messaging . . . 159
7.2 Automatic Protocol Selection . . . 164
7.3 Related Work . . . 171

8 A Fusion of Components and Actors 174
8.1 The Kompact DSL . . . 178
8.2 Kompact Implementation . . . 191
8.3 Evaluation . . . 196
8.4 Related Work . . . 207

9 Conclusions 208
9.1 Discussion & Future Work . . . 210

A Syntax of Kola Language Extensions 214

B Primer on the Rust Language 218
B.1 Data . . . 219
B.2 Behaviour . . . 220
B.3 Type Parameters, Trait Objects, and Associated Types . . . 222
B.4 Macros . . . 224

Bibliography 228

Acronyms 239

List of Figures

2.1 Decomposition of a Paxos service in Kompics. 19
2.2 Relationships between port position and event direction in Kompics. . 20
2.3 State transition diagram of the Kompics component lifecycle. 22

3.1 Subscription-checking example. 45
3.2 Pattern Size Distribution. 51
3.3 Kompics Scala throughput comparison. 54

4.1 Strict typing rules for the Kompics kernel language. 66
4.2 Strict typing rules for Kompics statements and expressions. 67
4.3 Typing rules for Kompics with event subtyping. 68

5.1 Overview of the KompicsTesting approach. 89
5.2 DFA recognising a sequence of events. 92
5.3 Classification overview. 107
5.4 Test execution time. 108
5.5 State number distribution. 109

6.1 Overview of the MPP flow. 115
6.2 Streaming Window overview. 134
6.3 Ping Pong Benchmark Results. 141
6.4 Throughput Ping Pong Benchmark Results (AWS). 142
6.5 Throughput Ping Pong Benchmark Results (Desktop). 142
6.6 Fibonacci Benchmark Results. 144
6.7 Chameneos Benchmark Results. 146
6.8 All-Pairs Shortest Path Benchmark Results. 148
6.9 Net Ping Pong Benchmark Results. 149
6.10 Net Throughput Ping Pong Benchmark Results. 151
6.11 Atomic Register Benchmark Results (Desktop). 152
6.12 Atomic Register Benchmark Results (AWS). 152
6.13 Streaming Windows Benchmark Results. 154

7.1 Component hierarchy for the file transfer experiment. 165
7.2 Transfer throughput between AWS datacentres. 166
7.3 Transfer throughput results on AWS. 170

8.1 The subscription-checking bottleneck. 176
8.2 Kompact Ping Pong Benchmark Results. 198
8.3 Kompact Throughput Ping Pong Benchmark Results. 198

8.4 Kompact Fibonacci Benchmark Results. 199
8.5 Kompact Chameneos Benchmark Results. 200
8.6 Kompact All-Pairs Shortest Path Benchmark Results. 201
8.7 Kompact Net Ping Pong Benchmark Results. 202
8.8 Kompact Net Throughput Ping Pong Benchmark Results. 203
8.9 Kompact Atomic Register Benchmark Results (AWS). 204
8.10 Kompact Streaming Windows Benchmark Results. 205

List of Tables

3.1 Code maintainability for Kompics Java and Scala. 52
3.2 On-premise Kompics Java and Scala usage. 55

4.1 Code maintainability for Kompics Java, Scala, and Kola. 80
4.2 Kola average compile time. 82

5.1 Transition table. 93
5.2 Classification results. 106

6.1 Overview over all implemented benchmarks. 126

7.1 Average throughput in the ICE to Halstead bare metal experiment. 171

8.1 Benchmark implementation variants for Kompact. 196

List of Algorithms

1 Basic Broadcast (*adapted from [16, p. 76]*) 15
2 Read-One Write-All (*adapted from [16, p. 144]*) 28
3 Floyd-Warshall all-pairs shortest path (*adapted from [55, alg. 141]*) . . 129
4 Read-Impose Write-Consult-Majority (*adapted from [16, p. 165f]*) . . . 132

List of Listings

2.1 Basic Broadcast in Erlang. 15
2.2 Basic Broadcast in Akka (Scala). 18
2.3 Basic Broadcast in Kompics Java. 24
3.1 Read-One Write-All in Kompics Java (*part 1*). 29
3.2 Read-One Write-All in Kompics Java (*part 2*). 30
3.3 Read-One Write-All in Kompics Scala. 33
3.4 Event and port declarations in Kompics Scala. 34
3.5 Component definitions with port positions in Kompics Scala. 35
3.6 Event handlers for native Scala events in Kompics Scala. 38
3.7 An event handler for matching Kompics Java events in Kompics Scala. 38
3.8 Component and connection setup in Kompics Scala. 39
3.9 Using a Kompics Scala port from Kompics Java. 43
4.1 Two misbehaving Actor programs 60
4.2 A failing program in Kompics Java. 60
4.3 A failing program in Kompics Scala. 61
4.4 Common Kompics Java mistakes. 62
4.5 Read-One Write-All in Kola. 70
4.6 Event declarations in Kola. 72
4.7 A port declaration in Kola. 73
4.8 Component definition with init block in Kola. 74
4.9 Port position declarations in Kola. 75
4.10 Child component declarations in Kola. 75
4.11 Handler declarations in Kola. 77
5.1 Trivial Broadcast 88
5.1 An example CFG for the specification of regular tests. 94
5.2 An example CFG for the specification of online tests. 100
5.3 An example CFG for the specification of white-box tests. 101
5.4 The CFG for the test specification DSL of the prototype. 102
5.5 A test specification in Java created using a builder pattern. 103
5.2 Chat component test example. 105
6.1 Example parameter space description. 118
6.2 Scala trait for local benchmarks. 122
6.3 Scala trait for distributed benchmarks. 123
6.4 The Pong Pong benchmark implemented in Akka (Scala). 125
7.1 Kompics' Network Port. 160
7.2 Kompics' address, header, and message interfaces. 161
7.3 A multi-hop routing header. 162
8.1 The Read-One Write-All algorithm in Kompact (*part 1*). 179
8.2 The Read-One Write-All algorithm in Kompact (*part 2*). 180

8.3 Ports and events in Kompact. 182
8.4 Minimal “Hello World” component in Kompact. 183
8.5 A Pure component BasicBroadcast struct in Kompact. 184
8.6 Pure component BasicBroadcast event handlers in Kompact. 185
8.7 Mixed BasicBroadcast event and message handlers in Kompact. . . 187
8.8 Mixed BasicBroadcast with NetworkActor in Kompact. 188
8.9 Setup code for components in Kompact. 189
8.10 Setup code for networked components in Kompact. 190

CHAPTER

1

Introduction

This is how you do it: you sit down at the keyboard and you put one word after another until its done. It's that easy, and that hard.

– Neil Gaiman

Distributed and concurrent software architectures have become the norm over the past two decades. During this time, previously rapid increases in CPU clock frequency have been replaced with more and more cores per die, more and more CPUs per server, and more and more servers per data centre. Just like individual core performance becoming a secondary concern to hardware designers, so have single-threaded performance optimisations given way to a focus on scalable computing approaches, which promise to exploit the inherent parallelism in many algorithms and applications, and thus efficiently utilise available hardware.

The stratification into layers of concurrency — between different cores, different servers, and even different data centres half way around the world — has necessitated the creation of programming models, that allow for the expression of solutions, which can take advantage of parallelism at every level, as well as programming languages, frameworks, and systems that implement those models.

At the data centre level, solutions have emerged that allow programmers to express data analysis and management pipelines in high-level, declarative domain-specific languages (DSLs), which leave the many problems of distributed resource management, scheduling, and execution to their runtime libraries, hidden away from the programmer. Among such solutions are batch processing systems like Google's MapReduce [26] and Apache Spark [116], as well as real-time "streaming" analytics systems like Apache Flink [18].

While systems like these enable scalable data analytics applications at the data centre level, they are not designed as general purpose tools to develop distributed applications; applications such as distributed analytics systems themselves, for example. They are too high-level to allow developers to properly address challenges on the level of individual network messages, for example. For the purpose of developing such distributed applications, a number of programming languages and frameworks have made the implementation of concurrent, scalable, networked applications their particular design focus. First and foremost among them has been the Erlang language [9], with its fundamental notion of lightweight, concurrent processes communicating via messages. Many other such *message-passing* languages and frameworks have been created since Erlang, among them Akka [69], Elixir [97], Go [40], Orleans [101], and Kompics [5].

As these solutions have matured, having proven their suitability for developing scalable applications, other shortcomings in their design have become more noticeable: Difficulties in static typing and correctness testing cause lengthened development cycles, as do unergonomic DSLs [29, 45]. Unsuitable abstractions cause compute cycles to be wasted unnecessarily, or network communication to be slower than desired [63]. As is often the case with maturing software, such inconveniences and inefficiencies, which were considered tolerable previously, eventually become the focus of improvements.

Precisely in this spirit, the goal of this dissertation is to improve existing message-passing frameworks and languages with respect to safety, ergonomics, correctness, and performance.

1.1 Motivation

In any software development task there is a tradeoff between two fundamental goals that must be managed: *Developer time* and *software quality*. The latter can come in many forms, such as guarantees about *correctness* of implemented solutions, their *performance* or *efficiency*, or more difficult to measure quantities, like how easy the software is to maintain over time, for example. This tradeoff is most clearly reflected in the design of programming languages, frameworks, and libraries, as the only purpose of their existence is to save developer time without reducing software quality.

One of the great time savers in the development of distributed systems, in particular, is the ability to discover or prevent bugs before the first deployment of the application onto a cluster of distributed machines. The fundamental reason for this is the long timescale of the debugging cycle for deployed software: Depending on the size of the application to be deployed, the deployment procedure itself can take minutes or hours. As a consequence of the large number of possible message interleavings in a real distributed application, certain paths through an execution are taken exceedingly rarely, and finding a bug in one of those paths can be all but impossible or take a very long time, at least. Once an issue has occurred, log files must typically collected from all hosts that were involved, and then compared, in order to reconstruct a consistent timeline that led to the faulty event. And finally, the offending code must be located, and then altered with the intention to correct the mistake, before the cycle begins anew.

Since the major time consumers are deployment, reproduction of bugs, and log collection and analysis, it is clear that avoiding all — or any — of these steps would allow for a huge improvement in development efficiency. For this reason, we have sought mechanisms and methodologies to either prevent bugs altogether or at least discover them early on the developers' machines, or perhaps as part of a continuous integration (CI)/continuous delivery (CD) solution.

While safe abstractions save developer time, they are only useful if they can still fulfil the performance requirements of the application domain. Thus, in any software engineering venture, performance in some sense or another is always a concern. As the performance of concurrent and distributed software depends mostly on the avoidance of scalability bottlenecks, we seek solutions that scale well and do not produce contention on a limiting resource. We are also interested in mechanisms to fairly measure and compare the performance of different approaches, so that developers can make informed judgements concerning tradeoffs between different solutions.

To summarise, our motivation is to contribute mechanisms that save developer time by reducing debugging effort, or simply by providing better programming ergonomics, as well as increasing performance of the related solutions, which we want to measure via benchmarking approaches.

1.2 Design Principles

Throughout this dissertation we follow a small set of key design ideas that we believe are conducive to the efficient development of correct and performant distributed applications, as described in the previous section. Most fundamentally, we favour *message-passing concurrency* over any other concurrency model, because of its ingrained scalability, intuitive semantics, and — most importantly — the natural correspondence to networked communication patterns. In addition to this very general philosophy, the following concrete principles permeate this dissertation:

D1 Domain-specific languages that imitate the formal descriptions of (distributed) algorithms do not only lead to more readable code, but also reduce the occurrence of subtle implementation (or transliteration) bugs. Both properties aid our goal to increase developer productivity and reduce software maintenance effort.

D2 Any approach that can prevent or identify bugs *before* a distributed application is deployed on a cluster is a net gain for developer productivity, as explained in the previous section. This includes attempts at static typing, as well as unit and integration testing.

D3 Networking abstractions for efficient applications can not simply hide away fundamental properties of networked communication. *Location transparency*, while convenient for dynamically adjusting systems, prevents the implementation of applications that can perform at high efficiency, as it forces unnecessary redundancies on the design. This is in clear contrast with our goal of designing distributed solutions with good performance characteristics.

D4 When performance is at stake, it is not always sufficient to blindly stick to a single paradigm, even if it appears to be convenient and of sufficient expressive power. Instead, hybrid solutions between multiple models can provide developers the power to exploit the strengths of each, without suffering from their respective weaknesses.

1.3 Primary Contributions

We summarise the main results of this work into the following set of individual contributions:

C1 We provide a new implementation of the *Kompics Component model* that very closely matches abstract distributed algorithm descriptions, particularly those from "Introduction to Reliable Distributed Programming" [16]. Our primary motivation for this work is to minimise the mental transfer overhead for students of courses on distributed systems. A secondary, yet important, concern was to reduce bugs incurred in any distributed software based on

such algorithm descriptions. Our implementation is an embedded domain-specific language in the Scala language, and is appropriately called *Kompics Scala*.

C2 We provide rules for statically typing the Kompics Component model, and give two different implementations based on these rules: The first is an external domain-specific language called *Kola*, which compiles to Java source code, thus reusing the existing Kompics (Java) implementation. The second implementation is an embedded domain-specific language in the Rust language, which we call *Kompact*, and is targeted at high-performance applications.

C3 We provide both a theoretical framework and an embedded domain-specific language for *unit testing* message-passing systems. We also describe a prototype implementation of this model for Kompics, which is called *KompicsTesting*. The prototype itself was *implemented* by a Ifeanyi Ubah, as part of a master's thesis supervised by the author of this dissertation.

C4 We perform a quantitative evaluation of the performance space of a number of state-of-the-art message-passing frameworks. This is based on a new benchmarking framework we describe, which allows cross-language comparisons of message-passing implementations both excluding and involving actual network communication.

C5 We argue for *less transparent* networking abstractions, and support our claims with an evaluation of how application-level protocols can outperform transport-level protocols for specialised tasks, given the appropriate level of abstraction is provided to them. Concretely, we use the example of a file transfer application to demonstrate performance gains of an application level protocol over other common file-transfer protocols, as well as our own implementation relying solely on transport level protocols.

C6 Finally, we design and implement a framework that combines the *Actor Model* with the *Kompics Component model*, called *Kompact*. We evaluate how the combination of an efficient implementation in the Rust language, together with the appropriate choice of communication model for a particular task, can confer significant performance benefits, even to relatively simple applications.

1.4 Previously Published Work

The majority of the content of this dissertation is based on previously published work, much of which has been peer-reviewed.

The complete list of material published and presented in conferences, journals and chapters along with assigned contributions is the following:

Conference Papers

P1 Lars Kroll, Jim Dowling, and Seif Haridi. 2016. **"Static Type Checking for the Kompics Component Model: Kola — The Kompics Language"**. In *First Workshop on Programming Models and Languages for Distributed Computing (PMLDC '16)* [62]. ACM, New York, NY, USA.

Contribution: The author of this dissertation designed and implemented the solution, and led the writing of the paper.

P2 Lars Kroll, Alexandru A. Ormenişan, and Jim Dowling. 2017. **"Fast and Flexible Networking for Message-Oriented Middleware"** In *IEEE 37th International Conference on Distributed Computing Systems (ICDCS '17')* [63]. IEEE, Atlanta, GA, USA.

Contribution: The author of this dissertation co-designed and implemented the networking abstractions and library implementation. The author of this dissertation designed and executed the evaluation, including the *automatic protocol selection* application in its entirety. The author of this dissertation also led the writing of the paper.

P3 Lars Kroll, Paris Carbone, and Seif Haridi. 2017. **"Kompics Scala: Narrowing the Gap between Algorithmic Specification and Executable Code (Short Paper)."** In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala (SCALA 2017)* [61]. ACM, New York, NY, USA.

Contribution: The author of this dissertation designed and implemented the solution, and co-designed and executed the evaluation. The author of this dissertation led the writing of the paper.

ArXiv Articles

P4 Ifeanyi W. Ubah, Lars Kroll, Alexandru A. Ormenişan, and Seif Haridi. 2017. **"KompicsTesting — Unit Testing Event Streams"**. In *The Computing Research Repository (CoRR)* [104].

Contribution: This technical report mostly concerns the Kompics specific implementation of the unit testing model. The author of this dissertation co-designed this implementation, and provided the theoretical model it is built on, which is reported in this dissertation.

1.5 Dissertation Outline

The remainder of this dissertation is structured as follows:

- We begin by describing programming models for distributed systems, in particular the *Actor Model* and the *Kompics Component Model*, as background for the following chapters, in chapter 2.
- In chapter 3, we explore the use of an embedded domain-specific language (eDSL) for Kompics in the Scala language, to reduce the mental gap experienced by students when faced with the task of translating formal descriptions of distributed algorithms into real code.
- Chapter 4 addresses the issue of preventing certain classes of bugs at compile time, by the use of static typing for the Kompics model. This chapter also provides an external DSL that implements the developed typing rules, while a second implementation in the form of an internal eDSL is provided later as part of chapter 8.
- As not all classes of bugs can be caught via static typing, chapter 5 investigates the use of the *unit testing* methodology on event streams. It also describes an implementation of such a testing model for Kompics Java.
- In chapter 6 we shift our focus from safety and correctness concerns to performance issues, and develop a framework for distributed benchmarking as a foundation for this.
- Chapter 7 investigates the performance impact of choosing appropriate network abstractions in message-passing frameworks. It also evaluates this impact quantitatively on the example of a file transfer application, that can select the appropriate transfer protocol for the current network conditions automatically at runtime.
- Drawing from the experiences described in previous chapters, we recognise the limitations of pure Actor or Component model implementations in chapter 8, and design and implement a hybrid model between the two, with a particular focus on performance. We show how choosing the correct model for the right application can lead to significant performance benefits, which can even be enhanced by the use of statically typed message-passing abstractions.
- Finally, chapter 9 summarises the dissertation, and provides directions for future work in the area, after discussing some limitations of the presented solutions.

CHAPTER

2

Background

In this chapter, we describe established, fundamental material used in many parts of the thesis. In particular we focus on the Actor model and the Kompics Component model for writing concurrent or distributed software.

Many models and formalisms have been developed to describe parallel and distributed computation, that is computation which aims to exploit the now dominant paradigm of multi-core processors, and potentially also addresses the challenges of partial failures when some of the processing elements fail while others continue their tasks.

A major distinguishing factor of those models is whether they are *message-passing* or based on *procedure calls*.

Message Passing In a message-passing model, processing elements communicate by handing data items called *messages* over to other processing elements, which in turn then handle these messages and potentially produce more messages, that they either hand off to other processing elements or perhaps return to the original element in the form of a *response*. In the latter case, where a response is expected, awaiting the response can happen either *synchronously* or *asynchronously*. In the synchronous case, the sending element suspends processing while waiting for a response like in the *Communicating Sequential Processes (CSP)* model [50, 89], for example. When responses are handled asynchronously, the sending element is left to continue processing other messages until the response arrives, at which point it can continue its previous execution. The asynchronous semantics are typically more common in message-passing models specifically designed for parallel and distributed computing, as they model more accurately what happens in a real network, thereby providing an abstraction that matches the real world more closely and allowing better resource exploitation.

In either case, the processing elements themselves may either be local to the same process and host machine, or distributed over different processes on different machines, in which case messages must be marshalled, potentially sent over the network, and then unmarshalled again, before being passed to the recipient. Individual processing elements also typically have some local state which is exclusive to them and not shared with other elements.

Examples of message-passing models are the *Actor* model [48], and the *Kompics* component model [7, 8].

Concurrent/Remote Procedure Calls In concurrent or remote procedure calling models, on the other hand, processing elements invoke a specific named piece of code on another processing element, passing it a predefined set of arguments, and typically expecting a predefined kind of response. Such invocations can be *synchronous*, causing the invoking element to suspend execution until the response is available, or *asynchronous*, allowing the invoking element to continue processing other invocations on itself until the result is available, at which point it may continue its previous execution that required the result. In the context of distributed systems, procedure calls may additionally happen *remotely*, that is, on a processing element that is running in a different process on a different host machine. Typically, a local "stub" object is available to invoke the target procedure on, which handles

marshalling of the arguments, sending the request over the network, where the data is unmarshalled and converted into the actual invocation. An equivalent path is followed in reverse for the result of the invocation. The issue of synchronous vs. asynchronous invocation is orthogonal to whether or not the invocation is remote. However, remote invocations are usually slower than local ones, causing longer suspension of the invoking element in synchronous models, which can negatively affect performance.

As *sequential* procedure calls are very common in most object-oriented programming languages, the syntactical similarities provide programmers of concurrent or remote procedure call models with a certain familiarity. However, the execution semantics of concurrent or remote procedure calls are almost always different from those of sequential procedure calls, rendering this familiarity potentially misleading.

One of the most popular remote procedure call (RPC) frameworks is gRPC, which is based on Google's Protocol Buffers[1] marshalling framework.

Both message-passing and concurrent or remote procedure calling models have the same expressive power, as can easily be seen by modelling any procedure call with a "request" message being answered by a "response" message. All concerns about synchronous vs. asynchronous and remote vs. local execution apply to both models.

This dissertation will focus almost exclusively on *message-passing models*, as they align more naturally with distributed execution, which itself is based on messages sent over an underlying network. The two models, in particular, that are heavily featured in this dissertation, are the *Actor* model and the *Kompics* component model, which are described in more detail below.

2.1 The Actor Model

The *Actor* model was orginally described by Hewitt in 1973 as a programming model for artificial intelligence [48], but has since been popularised for implementing distributed systems by — among others — the Erlang programming language [9], Scala Actors [44], and the Akka framework [69] for Java and Scala.

The processing elements in the Actor model are, perhaps unsurprisingly, called *actors* and represent a combination of exclusive *internal state* with a particular *behaviour* identified by a unique *name*. Given another actor A's name, any actor may send an arbitrary message m to A, and this is also the only way actors may interact. Message processing is decoupled from sending by use of a "mailbox", that is a message queue, which incoming messages are added to, while waiting for the receiving actor to be scheduled to process them. When m reaches the head of the message queue, A's behaviour determines how and if m will be handled. During message handling A may both consult and update its internal state, as well

[1] `https://developers.google.com/protocol-buffers/`

as produce one or more new messages by sending them to other actors with known names. Different implementations allow different semantics for mailboxes and message delivery, but first-in, first-out (FIFO) order with *exactly-once* or *at-most-once* delivery are the most common. Ordering is typically not enforced between messages from different actors.

As names can be passed as part of messages, actor systems can form arbitrary ad-hoc connection topologies that constantly evolve over their lifetimes. While this makes actors a very powerful abstraction, it can pose a particular challenge to determine how the rest of the system is affected when individual actors or whole subsystems fail. Different implementations provide different solutions for these challenges, but many support managed supervision hierarchies, where the actor system forms a tree of supervisors with actors higher up in the tree dealing with failures of actors below them. Additionally, many implemenations provide some indirection for actor names, such that a restarted actor can handle new messages sent to the name of its previously failed instance.

2.1.1 Implementations

The Actor model itself leaves out many details required for an actual implementation, for example, concerning scheduling, mailbox semantics, message matching, treatment of unmatched or unhandled messages, or how to deal with distributed deployments. In this section we will describe two particular implementations, namely Erlang and Akka, in more detail, as they are used in later parts of the dissertation.

2.1.1.1 Erlang

Designed in 1986 at Ericsson [10], Erlang [9] is a general-purpose, functional programming language, that has actors as its fundamental concurrecy abstraction. Actor behaviours are supported via an explicit language construct of the following form, where `PatternX` describes pattern to match incoming messages against, and depending which pattern matches, the corresponding `BodyX` is executed:

```
1 receive
2   Pattern1 ->
3     Body1;
4   ...;
5   PatternN ->
6     BodyN
7 end
```

Note that patterns are tried against all messages in the "mailbox" of the actor (which Erlang calls a *process*), and the first matching message is removed and executed, while unmatched messages are left in the queue for later when the behaviour changes via another `receive`-block. This has two important consequences: Firstly,

messages are not necessarily executed in FIFO order, as a behaviour can match the second message in the queue without matching the first one, while the following behaviour can match the first message afterwards. Secondly, if no behaviour ever matches a particular message *m*, then *m* will never be removed from the actor's "mailbox". This can lead to slow and difficult to detect resource leaks, especially if messages of the form *m* arrive with some regularity.

The `receive`-block described above is, in fact, an *expression* and only handles a single message before returning the result of the body of the matched pattern. As actor behaviours are typically continuous, a new `receive`-block must be invoked within this body. Typically, this is done by wrapping the `receive`-block in a function `f` and recursively calling `f` at the very end of every body with the current (updated) state as arguments, in order to make use of tail-call optimisation[2]. If the actor is meant to change behaviour as a result of handling a certain message, a different function with a different `receive`-block can be called instead.

A message `Msg` is sent to a process `P` via the expression `P ! Msg`, where `P` must either be a unique *process id*, a registered *name* (an *atom*), or a tuple `{Name, Node}` which identifies an actor globally (see below). Sending a message does not provide any feedback on whether or not the target is alive, or if the message gets handled at all. As targets can fail and restart between sending a message and processing, message delivery semantics can be considered to be at-most-once.

In addition to a very efficient single-machine implementation of the Actor model via its BEAM virtual machine (VM), Erlang also provides a runtime that allows different Erlang instances on different hosts to be connected, such that messages can be sent across the network with no further effort than to use a globally valid target identification, such as a name-node-tuple. The property that local and remote sending of messages looks essentially identical is referred to as *location transparency*, and is a common feature in many Actor implementations. It has the advantage that individual actors can be moved or restarted without necessarily having to update all outstanding references to them in other actors. However, it forces all communication to the common denominator standard of at-most-once semantics, which is typically weaker than necessary for purely local communication.

Example An implementation of a simple *Basic Broadcast* example, adapted from Cachin et al. [16, p. 76] as shown in algorithm 1, can be seen in listing 2.1. The actor gets initialised with a *parent* it is supposed to deliver messages to, and a list of *peers*, which are other broadcast actors. When it is asked to broadcast a message by its parent (or any other process), it forwards it with the `deliver` label to all its peers. Upon receiving such a message, each peer simply forwards it to its parent. In order to avoid the unbounded mailbox issue mentioned above, the broadcast actor simply ignores all messages that are not labelled with either `broadcast` or `deliver`.

[2] That is, avoiding the allocation of a new stack frame for recursive calls.

Algorithm 1 Basic Broadcast (*adapted from [16, p. 76]*)

Implements:

BestEffortBroadcast, **instance** *beb*.

Uses:

PerfectPointToPointLink, **instance** *pl*.

1: **upon event** $\langle beb, \text{BROADCAST} \mid m \rangle$ **do**
2: **for all** $q \in \Pi$ **do**
3: **trigger** $\langle pl, \text{SEND} \mid q, m \rangle$

4: **upon event** $\langle pl, \text{DELIVER} \mid p, m \rangle$ **do**
5: **trigger** $\langle beb, \text{DELIVER} \mid p, m \rangle$

```
1  -module(basicbroadcast).
2  -export([start/2]).
3
4  start(Parent, Peers) ->
5    loop(Parent, Peers).
6
7  loop(Parent, Peers) ->
8    receive
9      {broadcast, Msg} ->
10       lists:foreach(fun(P) -> P ! {deliver, Msg, Parent} end, Peers);
11       loop(Parent, Peers);
12     {deliver, Msg, From} ->
13       Parent ! {Msg, From};
14       loop(Parent, Peers);
15     _ -> loop(Parent, Peers) % ignore other messages
16   end.
```

Listing 2.1: Basic Broadcast in Erlang.

2.1.1.2 Akka

Originally developed by Jonas Bonér in 2009, the Akka toolkit [69] is now maintained as an open-source project by Lightbend Inc. and provides both Java and Scala implementations of the Actor model that are heavily inspired by Erlang. We will focus on the Scala application programming interface (API) in this section, as it is more readable.

Being embedded in an object-oriented language with inheritance, Akka actors must be *classes* that *extend* the `Actor` *trait*. Being classes automatically gives access to internal state in the form of *private* fields, however technically *public* state could still be defined and instance references leaked, which must, of course, be avoided if the state exclusivity of the Actor model is to be maintained. Actor behaviour is defined by *overriding* (i.e., implementing) the `receive: PartialFunction[Any, Unit]` method, typically with a (not necessarily exhaustive) `match`-block, of the following form:

```
1 override def receive = {
2   case Pattern1 => Body1
3   ...
4   case PatternN => BodyN
5 }
```

Behaviours defined in this manner behave in a similar manner to Erlang, that is executing the body associated with the matching pattern, but do not require the same recursive call to continue processing messages. As is evident from the signature, the return value from the body is also discarded. However, by virtue of being a class method, `receive` does have access to the actor's internal state, and as part of that an Akka defined `context` field. This context provides, for example, a reference to the *sender* of the message currently being processed, in the form of an `ActorRef` instance accessed via `sender()`. The implicit availability of sender information for every message sets Akka apart from most other Actor model implementations, and has two important consequences:

1) Sender information enables the convenient usage of the "ask"-pattern, which perfectly models the common request-response-pattern described earlier. In case of Akka an ask invocation via `target ? message` returns a `Future` f, which will eventually be fulfilled by a response created at the receiving actor via `sender() ! response`. While this pattern is often very convenient, it comes at a relatively high resource and performance cost, incurred due to the creation of a temporary actor that manages the writing side of f. This is necessary, because the value of the `sender()` variable on the receiver side must, of course, be an `ActorRef`. However, simply providing `ActorRef` of the actual sending actor would not allow the implementation to match up the response with the correct `Future` instance f, as it is missing additional context identifying f. This context is provided by the one-off actor created during the ask invocation, which handles only f. Another inconvenient consequence of Akka's implementation

of this model is that any `onComplete`-handler h subscribed on f is not executed in a way that prevents actor a, which originally sent the request `message`, from concurrently handling other messages itself. Thus, access to any internal state of a closed over by h is inherently thread-unsafe.

This issue could be worked around with the use of a custom execution context, linking f back to simple messages to a, which then cause the actual execution of h in the context of a instead. The fact that this approach is *not* the default behaviour in Akka can certainly lead to some rather subtle concurrency bugs.

2) Inconveniently, having implicit sender information for every message essentially prevents static typing of actor communication. That is, given an `ActorRef` we can not statically tell which message types the actor will handle and which it will ignore. While it is clearly possible to augment the `ActorRef` type with a generic type argument that specifies the messages that are allowed to be sent (via a common trait, typically), this does not work with implicit sender references. To see why this is the case, consider an actor a of type `A`, which accepts messages of type `MsgA`. To make sure it only receives messages of type `MsgA`, it must always be referred to by an `ActorRef[MsgA]` when being used as the target of a send. Now assume equivalent definitions for two more actors b and c. Assume both a and b must communicate with c and have received appropriate references of type `ActorRef[MsgC]`, to which they both send messages m_a and m_b, respectively. When c handles m_a the type of `sender()` should be `ActorRef[MsgA]`, so that it can only send replies actually handled by a. Similarly when c handles m_b the type of `sender()` should be `ActorRef[MsgB]`. But clearly both can not be true, as `sender()` must have a single type known at compile-time. In general, if we must be able to accept messages coming from arbitrary unknown actors, the only type for `sender()` we can infer with certainty is `ActorRef[Any]`, which is equivalent to the plain untyped `ActorRef`. Thus in any statically typed implementation of the Actor model, sender information must be passed explicitly as part of the messages, where it can be assigned an appropriate type unique to the message type. This is exactly the way the newer Akka Typed [71] API is designed.

Another addition that sets Akka apart from other Actor model implementations, is the introduction of *actor systems*, that is independent supervision hierarchies with their own executors and configuration, including network settings, for example.

In addition to the features described, the Akka toolkit offers many subpackages for clustering, persistence, service discovery, and streams, for example, but they are of limited interest in the context of this dissertation.

Example The same example from the previous section 2.1.1.1 can be seen in Akka in listing 2.2. Note that instead of using atoms and tuples to differentiate message types, we declare two case classes to tell broadcast and deliver messages apart.

```scala
package basicbroadcast;

import akka.actor.Actor

final case class Broadcast(msg: Any)
final case class Deliver(msg: Any, from: ActorRef)

class BasicBroadcast(parent: ActorRef,
                     peers: List[ActorRef]) extends Actor {
  override def receive = {
    case Broadcast(msg) => peers.foreach(p => p ! Deliver(msg, parent))
    case Deliver(msg, from) => parent ! (msg, from)
  }
}
```

Listing 2.2: Basic Broadcast in Akka (Scala).

2.2 The Kompics Component Model

Much, if not all, of this dissertation revolves around the *Kompics* component model [7, 8], a message-passing model for describing parallel computation using stateful *components* connected via *ports* and *channels*.

The core philosophy of the model is a strict decoupling of a service specification, called a *port*, and its implementation, called a *component*, which has no information about its environment including any other components providing services to it or making use of its services. This approach allows for a powerful hierarchical composition of large services from smaller building blocks and high modularity upon selecting particular implementations of required services upon deployment. Figure 2.1 demonstrates such a compositional approach on the example of the well-known Paxos [64] consensus algorithm.

2.2.1 Semantics

Kompics is a programming model for parallel and distributed systems that implements protocols as event-driven *components* connected via *channels*. However, instead of being globally connected to a component, channels are instead connected to the *ports* that are declared on a component. The "messages" in the Kompics component model are called *events*, to differentiate them from Actor-style messages, which are addressed to a recipient. Events, on the other hand, are not addressed, but travel along channels and through ports from a producer component to one or more consumer component. Ports impose restrictions on which events may enter or leave a component, thus providing a form of type system for components. They are declared as either *required* or *provided position* by each component, and the position of declaration affects the direction that events carried by that port. As part

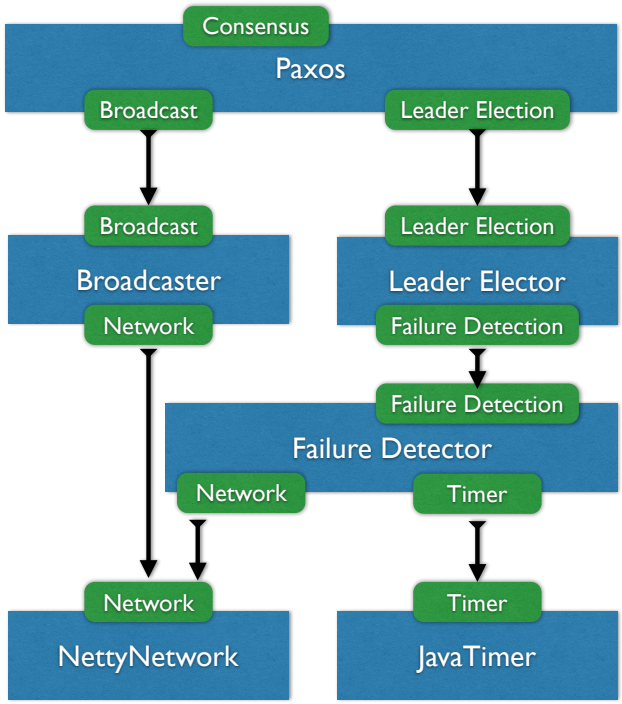


Figure 2.1: Decomposition of a Paxos service in Kompics.

of the specification of a port, events carried on it are declared as either *indications* or *requests*.

A simple, if not always accurate, mental model for this is to imagine ports as service interfaces. In this model, *request* events allow the invocation of the services that are provided by a component that *provides* the port. On the other hand, *indication* events are responses that a component *requiring* the port should expect.

Figure 2.2 summarises travel directions of events with respect to port positions. Events that are leaving a component C are called *outgoing* at C, and are produced by *triggering* — a term we will use instead of "send", when talking about Kompics events as opposed to Actor messages — on a declared port P. *Incoming* events at C, on the other hand, must be *handled* by one of C's (event-)*handlers*.

Let I and R be events both carried by P, and I is declared an *indication* event, while R is declared a *request* event on P. Assuming that P is a *provided* port on C, then I is *outgoing* on C and may be triggered, while R is *incoming* and must be handled. Conversely, if P is a *required* port on C, then R is *outgoing* and I is *incoming*.

Figure 2.2: Relationships between port position and event direction in Kompics.

Example In order to make the metaphor expressed in port position and event direction clearer, we will employ the example of a subscription-based eventual failure detection service, which we shall refer to as $\Diamond\mathcal{P}^{\odot}$. A component using the service subscribes to updates for a particular `Address` (a concept we shall treat as opaque for now) via a ⟨WATCH | *addr* : `Address`⟩ event. If and when *addr* becomes unreachable, an implementation of $\Diamond\mathcal{P}^{\odot}$ must trigger a ⟨SUSPECT | *addr* : `Address`⟩ event, which will travel along all connected channels, whether or not the component on the other end is actually watching this particular address. If *addr* becomes reachable again in the future $\Diamond\mathcal{P}^{\odot}$ must trigger a ⟨RESTORE | *addr* : `Address`⟩ event to revoke its earlier suspicion.

In the port specification of $\Diamond\mathcal{P}^{\odot}$ the WATCH event is declared as *request*, while both SUSPECT and RESTORE events are declared as *indication*. Assume that a component F is an implementation of $\Diamond\mathcal{P}^{\odot}$, then $\Diamond\mathcal{P}^{\odot}$ must occur in F's specification as *provided*, as providing a port is what it means to be an "implementation" of a port. With $\Diamond\mathcal{P}^{\odot}$ in provided position, F can behave as described above, handling WATCH events and triggering SUSPECT and RESTORE events as necessary. Now assume another component C must keep track of a set of reachable addresses $A = \{a_1, a_2, a_3\}$. In its specification, it will *require* $\Diamond\mathcal{P}^{\odot}$ and some time after starting it will trigger events ⟨WATCH | a_i⟩ for each address $a_i \in A$ on $\Diamond\mathcal{P}^{\odot}$, which it can do with $\Diamond\mathcal{P}^{\odot}$ in required position. Assuming it is connected with F via a channel on $\Diamond\mathcal{P}^{\odot}$, F will then keep C informed about the status of the $a_i \in A$, triggering SUSPECT and RESTORE events as required, which C, in turn, will handle.

Channels connecting ports decouple components, such that event delivery and processing is asynchronous. They provide FIFO order and *exactly-once* (per connected consumer component) delivery.

2.2.2 Kompics Java

The reference implementation for the Kompics component model is the *Kompics framework* [99, 7, 8] — which we will call *Kompics Java* in this dissertation, to differentiate from other implementations described herein — a Java library originally written by Cosmin Arad, but now an open-source project with a number of contributers including the author of this dissertation.

In this section we describe implementation concerns addressed by Kompics Java, but faced by any implementation of the Kompics component model.

2.2.2.1 Component Scheduling

In Kompics Java, incoming events are added to a port queue of outstanding events, but only if a handler subscription for them exists (see below). Components are only scheduled if there are outstanding events queued on one or more of their ports. A component is guaranteed to be scheduled and executed by a single thread at a time and thus has exclusive access to its internal state without the need for further synchronisation. Different components, however, may be scheduled in parallel on a thread pool, in order to exploit the parallelism expressed in a message-passing program. When a component is scheduled it handles one event at a time, until either there are no more events queued at its ports or a configurable maximum number $\eta_{\max}$ of events to be handled is reached. After the component has finished handling events, it will be placed at the end of the FIFO queue of components waiting to be scheduled, if it still has outstanding events. Tuning $\eta_{\max}$ enables developers to pick a tradeoff between increased throughput, where higher values maximise cache reuse through fewer component context switches, and fairness; that is, avoiding starvation of components with fewer queued events.

2.2.2.2 Handler Execution

As events in Kompics are not addressed to components in any way, but are instead published on all connected channels, the same event can be received by many components. The components themselves decide which events to handle and which to ignore by *subscribing* (event-)*handler*s on their declared ports. Note that ignored messages are silently dropped, which is necessitated by the channel broadcasting model. That is to say, as opposed to Actor systems, in Kompics it is often completely correct to simply ignore a large number of events arriving at a particular port. In order to avoid unnecessary load on the scheduler, a Kompics component is only scheduled *after* checking that at least one subscription exists for the incoming event. Matching of an event e of type E to a handler h that is declared to handle events of type E_h is based on Java's subtyping relationship <:, such that h matches e iff $E <: E_h$, in other words h matches E_h and all its subtypes.

2.2.2.3 Lifecycle

In order to support dynamic creation and destruction of components, as well as consistent channel reconnection semantics and fault handling, Kompics Java provides a lifecycle model that every component follows. An overview of this model can be seen in figure 2.3. All components start in the PASSIVE state, when they are first created. In this state only the START lifecycle event is handled and all other events are queued up for later. START events can be triggered on a child component's control port explicitly, or implicitly as part of the top-level `Kompics.createAndStart` call, which creates a new Kompics system. A common confusion for new Kompics users is, that components created as part of a hierarchy in a `Kompics.createAndStart` call are started implicitly, while components created later at runtime require explicit

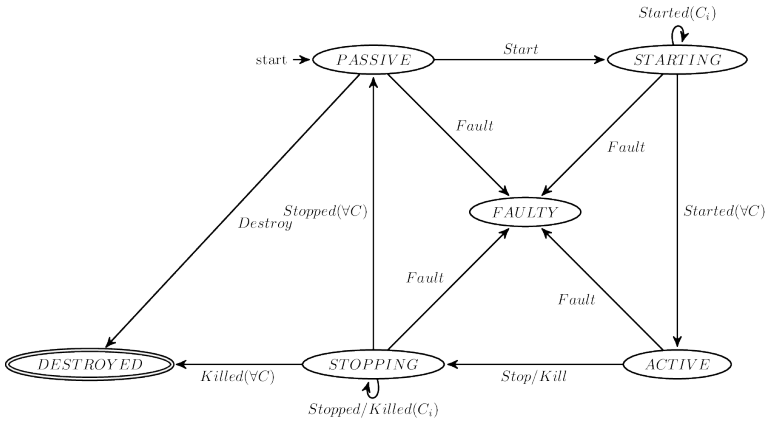


Figure 2.3: State transition diagram of the Kompics component lifecycle. *Image taken from the Kompics documentation [99].*

START events. Independently of the source of the START event, once it arrives at a component c, c will move to the STARTING state and forward the START event to all its children (if any). As in the PASSIVE state, only a single lifecycle event is handled in the STARTING state. Before moving to the ACTIVE state, c waits for all its children to respond with a STARTED event. As the STARTED events from the last child arrives, c moves to the ACTIVE state and sends a STARTED event for itself to its parent. An equivalent procedure is followed when a component needs to be stopped. In this way, a consistent lifecycle state for a whole supervision sub-tree is guaranteed. Kompics Java supports two different kinds of stopping a component: The use of a STOP event instructs the system to "pause" a subtree of components, returning them to the initial PASSIVE state. This procedure is particularly convenient for consistently changing channel connections of existing components, ensuring a consistent cut in the event processing. Alternatively, a component sub-tree can be destroyed via the KILL event, which prevents it from being restarted and eventually frees it up for garbage collection (assuming no other outstanding references).

From every state except DESTROYED, an exception thrown during the execution of an event handler will immediately move a component and all its children into the FAULTY state. After the sub-tree has been marked as faulty, the *fault handler* of the sub-tree's parent component (if any, otherwise the system's fault handler) at `ComponentDefinition.handleFault(Fault)` is consulted, in order to resolve the fault. Kompics Java provides four different possible actions that can be taken upon the fault of a child component:

Escalate A component can escalate the fault to its own parent, thereby causing itself and all its other children to be marked as faulty as well.

Resolved The custom fault handler code can mark the fault as resolved, to declare

that no further action is necessary by the Kompics system. This makes sense if the fault handler implemented custom restarting of child components, for example.

Ignore If a fault is marked as ignored, the faulty sub-tree is immediately moved into the PASSIVE state, and a START event is sent to resume processing.

Destroy Finally, the whole sub-tree can simply be destroyed, cleaning up the possibly inconsistent state.

As fault-handling also applies to a whole component sub-tree, and not just to a single faulty component, Kompics continues to ensure consistent lifecycle within a sub-tree.

Example The same example from the section 2.1.1.1 can be seen in Kompics Java in listing 2.3. Note how we define a `BestEffortBroadcast` port with the two required events, which we then implement in the `BasicBroadcast` component by marking it as *provided*. We also assume there is a `PerfectPointToPointLink` port abstraction in scope, together with some `Address` type it requires to send messages over the network. We make use of that `PerfectPointToPointLink` by marking it as *required*. Note also the two explicit handler subscription to `beb` and `pl`, which are often forgotten by programmers.

```
1  package basicbroadcast;
2
3  import se.sics.kompics.ComponentDefinition;
4  import se.sics.kompics.KompicsEvent;
5  import se.sics.kompics.PortType;
6
7  public class BestEffortBroadcast extends PortType {
8    public static class Broadcast implements KompicsEvent {
9      public final KompicsEvent msg;
10     public Broadcast(KompicsEvent msg) {
11       this.msg = msg;
12     }
13   }
14
15   public static class Deliver implements KompicsEvent {
16     public final KompicsEvent msg;
17     public final Address from;
18     public Deliver(KompicsEvent msg, Address from) {
19       this.msg = msg;
20       this.from = from;
21     }
22   }
23
24   {
25     request(Broadcast.class);
26     indication(Deliver.class);
27   }
28 }
29
30
31 public class BasicBroadcast extends ComponentDefinition {
32
33   Positive<PerfectPointToPointLink> pl = requires(PerfectPointToPointLink.class);
34   Negative<BestEffortBroadcast> beb = provides(BestEffortBroadcast.class);
35
36   private final List<Address> peers;
37
38   public BasicBroadcast(Init init) {
39     this.peers = init.peers;
40
41     subscribe(broadcastHandler, beb);
42     subscribe(deliverhandler, pl);
43   }
44   public static class Init extends se.sics.kompics.Init<BasicBroadcast> {
45     public final List<Address> peers;
46     public BasicBroadcast(List<Address> peers) {
47       this.peers = peers;
48     }
49   }
50
51   Handler<BestEffortBroadcast.Broadcast> broadcastHandler = new
     ↪  Handler<BestEffortBroadcast.Broadcast>() {
52     @Override
53     public void handle(BestEffortBroadcast.Broadcast broadcast) {
54       for (Address peer : peers) {
55         trigger(new PerfectPointToPointLink.Send(peer, broadcast.msg), pl);
56       }
57     }
58   };
59
60   Handler<PerfectPointToPointLink.Deliver> deliverHandler = new
     ↪  Handler<PerfectPointToPointLink.Deliver>() {
61     @Override
62     public void handle(PerfectPointToPointLink.Deliver deliver) {
63       trigger(new BestEffortBroadcast.Deliver(deliver.msg, deliver.from), beb);
64     }
65   };
66 }
```

Listing 2.3: Basic Broadcast in Kompics Java.

CHAPTER

3

From Algorithm to Implementation

In this chapter, we describe a Scala DSL for Kompics that is designed to model abstract algorithm descriptions closely, and thus reduce mental transfer overhead for students and developers.

Kompics Java has been used for programming exercises and projects in distributed systems education at KTH for many years, as its abstractions closely match the ones found in common textbook descriptions of distributed algorithms, in particular from "Introduction to Reliable and Secure Distributed Programming" [16].

To give an example, consider algorithm 2: It describes the implementation of a $(1, N)$ regular register abstraction, that is a distributed read/write-register with a single writer and multiple readers and "regular" semantics (see [16, p. 141] for detailed properties). The algorithm relies on three other abstractions to fulfil its purpose: A best effort broadcast, a perfect point to point link, and a perfect failure detector.

It can easily be seen in algorithm 2 how the **Implements** and **Uses** blocks map to *required* and *provided* port positions respectively. Furthermore, **upon event** blocks clearly map directly to *event handlers*, and **trigger**s are present in both models. In fact, the only[1] construct in algorithm 2 that maps poorly into Kompics Java is the **upon**-<condition>-block at the very end, which needs to be implemented using checks after every update to any of the two referenced sets.

A sketch of a Kompics Java implementation of algorithm 2, which ignores some subtleties of how to inject sender information into the `WriteBEBDeliver` and `Ack` messages for brevity, can be seen in listings 3.1 and 3.2. Apart from noting the general verbosity inherent in a Java-embedded DSL, a few points of contention deserve particular attention.

Separate Handler Creation and Subscription All the very important (and easily forgotten) `subscribe` commands for the event handlers, listing 3.1 lines 54-58, are quite far away in the code from the handler declarations themselves. This distance increases the mental overhead to manage subscriptions, especially when maintaining existing code. A forgotten handler subscription is very difficult to debug, as no errors are thrown typically, but rather some required behaviour silently does not happen, often preventing the algorithm from eventually terminating or possibly causing timeouts somewhere completely different in a program.

Of course, the distance between handler declaration and subscription could be somewhat reduced by either declaring handlers in the constructor, like we do for subscriptions, or putting both handler declarations and subscriptions into an *instance initializer* block (see the Java Specification [41, p. 272]). However, this approach is not *always* feasible, as losing the reference to a handler prevents it from being unsubscribed later. The ability to unsubscribe handlers, however, is necessary in components where behaviour — which is primarily defined by handler subscriptions — must change dynamically at runtime. It should also be noted

[1] In the original Kompics Java implementation by Cosmin Arad [7, 8] ⟨INIT⟩ events actually existed and behaved as expected in algorithm 2, being handled in an event handler. However, they were moved into simple class constructors in a later release, as they complicated the implementation and lifecycle model unnecessarily.

Algorithm 2 Read-One Write-All (*adapted from [16, p. 144]*)

Implements:
(1, *N*)-RegularRegister, **instance** *onrr*.

Uses:
BestEffortBroadcast, **instance** *beb*.
PerfectPointToPointLink, **instance** *pl*.
PerfectFailureDetector, **instance** *P*.

1: **upon event** ⟨ INIT ⟩ **do**
2: $val := \bot$
3: $correct := \Pi$
4: $writeset := \emptyset$
5: **upon event** ⟨ P, CRASH | p ⟩ **do**
6: $correct := correct \setminus \{p\}$
7: **upon event** ⟨ $onrr$, READ ⟩ **do**
8: **trigger** ⟨ $onrr$, READRETURN | val ⟩
9: **upon event** ⟨ $onrr$, WRITE | v ⟩ **do**
10: **trigger** ⟨ beb, BROADCAST | [WRITE, v] ⟩
11: **upon event** ⟨ beb, DELIVER | p, [WRITE, v] ⟩ **do**
12: $val := v$
13: **trigger** ⟨ pl, SEND | p, ACK ⟩
14: **upon event** ⟨ pl, DELIVER | p, ACK ⟩ **do**
15: $writeset := writeset \cup \{p\}$
16: **upon** $correct \subseteq writeset$ **do**
17: $writeset := \emptyset$
18: **trigger** ⟨ $onrr$, WRITERETURN ⟩

that subscribing a freshly created handler, while not *always* desired, is usually the default case. Despite this fact, Kompics Java's syntax optimises for the less common case of subscribing handlers later; dynamically at component runtime.

Subclassing instead of Wrapping Both the broadcast of the "write" message and the direct message for the acknowledgement, which in algorithm 2 were simply tagged with atomic markers WRITE and ACK, respectively, have been converted into subclasses of `BestEffortBroadcast.Deliver` and `PerfectPointToPointLink.Deliver`, respectively. This conversion, in fact, assumes implementations of both BestEffortBroadcast and PerfectPointToPointLink abstractions that deliver the content of the SEND event directly, rather than wrapping into an additional DELIVER event. This assumption is incompatible, for example, with the BestEffortBroadcast implementation we gave in listing 2.3 as part of section 2.2.2. It is, however, necessary in a Kompics environment, where multiple higher-level abstractions make use of a

```
1  package registers;
2
3  import se.sics.kompics.ComponentDefinition;
4  import se.sics.kompics.KompicsEvent;
5  import se.sics.kompics.PortType;
6
7  public class OneNRegularRegister extends PortType {
8    public static class Read implements KompicsEvent {
9      public static final Read EVENT = new Read();
10   }
11   public static class ReadReturn implements KompicsEvent {
12     public final Object v;
13     public ReadReturn(Object v) {
14       this.v = v;
15     }
16   }
17
18   public static class Write implements KompicsEvent {
19     public final Object v;
20     public Write(Object v) {
21       this.v = v;
22     }
23   }
24   public static class WriteReturn implements KompicsEvent {
25     public static final WriteReturn EVENT = new WriteReturn();
26   }
27
28   {
29     request(Read.class);
30     request(Write.class);
31     indication(ReadReturn.class);
32     indication(WriteReturn.class);
33   }
34 }
35
36
37 public class ReadOneWriteAll extends ComponentDefinition {
38
39   Negative<OneNRegularRegister> onrr = provides(OneNRegularRegister.class);
40
41   Positive<BestEffortBroadcast> beb = requires(BestEffortBroadcast.class);
42   Positive<PerfectPointToPointLink> pl = requires(PerfectPointToPointLink.class);
43   Positive<PerfectFailureDetector> pfd = requires(PerfectFailureDetector.class);
44
45   private HashSet<Address> correct;
46   private Object val = null;
47   private HashSet<Address> writeset;
48
49   public ReadOneWriteAll(Init init) {
50     this.correct = new HashSet(init.peers);
51     this.writeset = new HashSet();
52
53     subscribe(crashHandler, pfd);
54     subscribe(readHandler, onrr);
55     subscribe(writeHandler, onrr)
56     subscribe(bebDeliverhandler, beb);
57     subscribe(plDeliverhandler, pl);
58   }
59   public static class Init extends se.sics.kompics.Init<ReadOneWriteAll> {
60     public final List<Address> peers;
61     public BasicBroadcast(List<Address> peers) {
62       this.peers = peers;
63     }
64   }
65
```

Listing 3.1: Read-One Write-All in Kompics Java (*part 1*).

```
66      @Override
67      public void handle(PerfectFailureDetector.Crash crash) {
68        correct.remove(crash.p);
69        checkCondition();
70      }
71    };
72
73    Handler<OneNRegularRegister.Read> readHandler = new Handler<OneNRegularRegister.Read>() {
74      @Override
75      public void handle(OneNRegularRegister.Read read) {
76        trigger(new OneNRegularRegister.ReadReturn(val), onrr);
77      }
78    };
79
80    Handler<OneNRegularRegister.Write> writeHandler = new Handler<OneNRegularRegister.Write>() {
81      @Override
82      public void handle(OneNRegularRegister.Write write) {
83        trigger(new BestEffortBroadcast.Broadcast(new WriteBEBDeliver(write.v)), beb);
84      }
85    };
86
87    Handler<WriteBEBDeliver> bebDeliverHandler = new Handler<WriteBEBDeliver>() {
88      @Override
89      public void handle(WriteBEBDeliver deliver) {
90        this.val = deliver.v;
91        trigger(new PerfectPointToPointLink.Send(deliver.from, Ack.EVENT), pl);
92      }
93    };
94
95    Handler<Ack> plDeliverHandler = new Handler<Ack>() {
96      @Override
97      public void handle(Ack ack) {
98        this.writeset.add(ack.from);
99        checkCondition();
100     }
101   };
102
103   private void checkCondition() {
104     if (writeset.containsAll(correct)) {
105       writeset.clear();
106       trigger(OneNRegularRegister.WriteReturn.EVENT, onrr);
107     }
108   }
109
110   private static class WriteBEBDeliver extends BestEffortBroadcast.Deliver {
111     public final Object v;
112     public WriteBEBDeliver(Object v) {
113       this.v = v;
114     }
115   }
116
117   private static class Ack extends PerfectPointToPointLink.Deliver {
118     public static final Ack EVENT = new Ack();
119   }
120 }
```

Listing 3.2: Read-One Write-All in Kompics Java (*part 2*).

BasicBroadcast component.

To understand why this is the case, consider a minimal example, where two different components c_1 and c_2 rely on the BestEffortBroadcast abstractions provided by our BasicBroadcast component c_{BB}. When c_{BB} receives a $\langle pl, \text{DELIVER} \mid p, m\rangle$ event, if it is implemented as in algorithm 1, it will rewrap m into a $\langle beb, \text{DELIVER} \mid p, m\rangle$ event and trigger it on the *beb* port. As both c_1 and c_2 are connected to that port via channels, Kompics will attempt to deliver the event to both components, checking first for compatible handlers, as described in section 2.2.2.2. Since both c_1 and c_2 expect events of the form $\langle beb, \text{DELIVER} \mid p, m\rangle$, they will both have subscriptions for them, and will thus be scheduled for execution. Now assume that c_1 is a regular register and actually only cares about $\langle beb, \text{DELIVER} \mid p, [\text{WRITE}, v]\rangle$ events, while c_2 is perhaps an implementation of All-Ack Uniform Reliable Broadcast [16, p. 83] and thus is only interested in $\langle beb, \text{DELIVER} \mid p, [\text{DATA}, s, m]\rangle$ events. If the distinguishing tag, a Java class corresponding to WRITE or DATA, is wrapped inside the DELIVER event, then the components can only decide whether or not an event is actually of interest to them, after being scheduled and executed, for example, via an `instanceof` check. However, this late check stands in contradiction to Kompics' policy of only scheduling components for events they really are handling. The only way[2] to achieve the desired behaviour in Kompics Java then, is to make use of its support for checking subscriptions for subtypes of the event type delivered by an abstraction, such as BestEffortBroadcast. Extending the `BestEffortBroadcast.Deliver` event, in combination with an implementation that avoids wrapping as described above, allows the Kompics Java implementation to select early on the type of `WriteBEBDelivery` and `DataBEBDelivery` subclasses, when checking c_1 and c_2 for handler subscriptions, and thus preventing unnecessary scheduling of the unaffected component.

While this pattern of subclassing instead of wrapping is *functional*, both in the case of BestEffortBroadcast and PerfectPointToPointLink, as well as other similar messaging abstractions, it is both *verbose* and rather *unintuitive*, as was just made evident by the half a page or so of text needed to explain it in the first place. The ability to employ a form of pattern matching early during subscription-checking would improve the readability of Kompics Java code significantly, but Java's complete lack of pattern matching support, makes this goal essentially unattainable.

As a matter of fact, the author of this dissertation has attempted to provide a Java solution to this *very* common issue, by adding a new `PatternMatchingHandler` class to the Kompics Java implementation, which, when paired with an event implementing the `PatternExtractor` interface, allows the subscription-checking to be extended to the extracted field or fields [99, section "Message Matching Handlers"]. However, while this approach is more ergonomic than the one described above, it still falls significantly short of the readability provided by the pattern matching notation used in the algorithmic specification.

[2] Later releases of Kompics Java have introduced a limited mechanism for pattern matching incoming events, that could also be used to achieve this.

While the ergonomics issues of Kompics Java described above may be no more than a constant nuisance to someone accustomed to working with Kompics Java, they present real hurdles to students, who are both new to distributed algorithms and Kompics alike. Apart from causing students to waste many hours of work on issues not directly related to their learning goals, such as finding missing handler subscriptions or figuring out the subclassing instead of tagging pattern described above, these issues also transitively cause a number of unnecessary emails and questions to the teaching assistant, thereby inducing motivation in the latter to address them.

In order to address the three issues outlined above, that is (Kompics) Java's verbosity, inconvenient handler subscription, and the lack of pattern matching, we provide a new implementation of the Kompics component model, embedded as a DSL in the Scala language [85] — a language much better suited for embedding DSLs than Java —, while maintaining interoperability with Kompics components written in Kompics Java.

A Kompics Scala implementation of algorithm 2 can be seen in listing 3.3 and serves as a teaser for the rest of this chapter. Contrasting it with the Kompics Java implementation in listings 3.1 and 3.2 at a glance, it should become apparent that Kompics Scala not only saves lines of code, but is also much easier to read and maps more intuitively to the pseudocode from algorithm 2.

The remainder of this chapter will describe the new DSL in detail, using excerpts from listing 3.3 as examples, as well as explain how pattern matching interacts with Kompics' policy to schedule only if necessary. Finally, we will investigate the performance cost of the new implementation over the simpler Kompics Java one.

3.1 The Kompics Scala DSL

Kompics Scala is an implementation of the Kompics component model in the Scala language [85]. It exploits Scala's many features that are convenient for writing DSLs, such as *operator overloading*, *infix notation* for methods, and *block notation* for function arguments, as well as the ability to create true singleton *objects* and *symbol literals*, and most importantly *pattern matching*.

Kompics Scala, however, is not a complete from-scratch reimplementation of the Kompics component model, but rather a DSL layer on top of the Kompics Java implementations. The exact implementation differences will be discussed in section 3.2, but for now it is sufficient to note that, while Kompics Java implementations live in the `se.sics.kompics` namespace, language constructs and implementations unique to Kompics Scala can be found in the `se.sics.kompics.sl` namespace. For brevity, we will omit the `se.sics.kompics`-prefix in the remainder of this section, and simply write `C` to denote a class or interface from Kompics Java and `sl.C` to denote a construct from Kompics Scala.

```
package registers;

import se.sics.kompics.sl._;

object OneNRegularRegister extends Port {
  case class Write(v: Any) extends KompicsEvent
  case object Read extends KompicsEvent
  case object WriteReturn extends KompicsEvent
  case class ReadReturn(v: Option[Any]) extends KompicsEvent

  request[Write];
  request(Read);
  indication(WriteReturn);
  indication[ReadReturn];
}

class ReadOneWriteAll(init: Init[ReadOneWriteAll]) extends ComponentDefinition {
  import OneNRegularRegister._;

  val onrr = provides(OneNRegularRegister);

  val beb = requires(BestEffortBroadcast);
  val pl = requires(PerfectPointToPointLink);
  val pfd = requires[PerfectFailureDetector];

  private val Init(peers: List[Address]) = init;

  private var correct: Set[Address] = Set(peers:_*);
  private var value: Option[Any] = None;
  private var writeSet: Set[Address] = Set.empty;

  pfd uponEvent {
    case crash: PerfectFailureDetector.Crash => {
      correct -= crash.p;
      checkCondition();
    }
  }

  onrr uponEvent {
    case Read => {
      trigger (ReadReturn(value) -> onrr);
    }
    case Write(v) => {
      trigger (BestEffortBroadcast.Broadcast(('write, v)) -> beb);
    }
  }

  beb uponEvent {
    case BestEffortBroadcast.Deliver(from, ('write, v)) => {
      value = Some(v);
      trigger (PerfectPointToPointLink.Send(from, 'ack) -> pl);
    }
  }

  pl uponEvent {
    case PerfectPointToPointLink.Deliver(from, 'ack) => {
      writeSet += from;
      checkCondition();
    }
  }

  private def checkCondition(): Unit = {
    if (correct subsetOf writeSet) {
      writeSet = Set.empty;
      trigger (WriteReturn -> onrr);
    }
  }
}
```

Listing 3.3: Read-One Write-All in Kompics Scala.

```
5  object OneNRegularRegister extends Port {
6    case class Write(v: Any) extends KompicsEvent
7    case object Read extends KompicsEvent
8    case object WriteReturn extends KompicsEvent
9    case class ReadReturn(v: Option[Any]) extends KompicsEvent
10
11   request[Write];
12   request(Read);
13   indication(WriteReturn);
14   indication[ReadReturn];
15 }
```

Listing 3.4: Event and port declarations in Kompics Scala.

Furthermore, note that a few classes and static fields from Kompics Java are re-exported in Kompics Scala, with the intention that an `import se.sics.kompics.sl._` statement should often be sufficient to have everything required for a simple Kompics program in scope. In particular, the re-exports include a type alias for the the `KompicsEvent` interface, as well as both type aliases and field re-exports for the lifecycle events `Start`, `Started`, `Stop`, `Stopped`, `Kill`, and `Killed` and their associated singleton instance field `event`. The empty init event in `Init.NONE` is also re-exported as `sl.Init.NONE` in order to avoid confusion about which of the two may be in scope, however it is often more convenient to use `sl.Init.none[C]` instead, to pass an empty init event to a component of type `C`. This is more convenient, because that invocation performs correct type coercion to satisfy the signature of the *create* method in `def create[C <: ComponentDefinition: TypeTag](init: Init[C]): Component`, which requires that the generic parameter of the init event matches the type of the component definition it is being applied to.

3.1.1 Events and Ports

In a manner equivalent to Kompics Java, any Scala *class* or *object*, as well as *case class* and *case object*, can be used as an **event**, as long as it implements/extends the `KompicsEvent` marker interface/trait. In order to interact properly with Scala's pattern matching facilities, case classes and case objects are recommended, as the Scala compiler automatically generates many convenience methods for them, such as the `unapply` method used in patterns, for example.

Port Type definitions should now be done in an *object* extending `sl.Port`, as port types are meant to be singletons. Reflecting the distinction between an *object* and a *type* (as defined by a class), there are now two different signatures to mark events as *request* or *indication* on a port. For an event `E`, that is defined as a *class* or *case class*, the syntax `request[E]` and `indication[E]` is used, while for a an event *E*, that is defined as an *object* or *case object*, the syntax `request(E)` and `indication(E)` must be used.

```
17 class ReadOneWriteAll(init: Init[ReadOneWriteAll]) extends
   ↪ ComponentDefinition {
18   import OneNRegularRegister._;
19
20   val onrr = provides(OneNRegularRegister);
21
22   val beb = requires(BestEffortBroadcast);
23   val pl = requires(PerfectPointToPointLink);
24   val pfd = requires[PerfectFailureDetector];
25
26   private val Init(peers: List[Address]) = init;
27
28   private var correct: Set[Address] = Set(peers:_*);
29   private var value: Option[Any] = None;
30   private var writeSet: Set[Address] = Set.empty;
31
```

Listing 3.5: Component definitions with port positions in Kompics Scala.

While this duplication might seem confusing, it is usually enforced by the compiler and thus not a significant concern from a learning and ergonomics perspective. The alternative — more consistent — syntax `request[E.type]` and `indication[E.type]` is also available for object events, but not any more readable in the author's opinion, and thus not recommended.

Internally, there is little difference between the two variants of each method, as both simply acquire an instance of the Java `Class` representing the event, and pass it to the Kompics Java implementation of `PortType.request` or `PortType.indication` respectively. In the case of an *object* the `Class` instance can simply be obtained via Java's `getClass()` method, while the method expecting a *type* makes use of a Scala feature called a "TypeTag", which is a mechanism for the Scala compiler to pass type information into a runtime instance. A `Type` instance acquired from such a TypeTag can then be converted to a Java `Class` instance via Scala's runtime reflection facilities.

In listing 3.4 Kompics Scala event and port definitions similar to those in the Kompics Java listing 3.1 can be seen. Note how both `Read` and `WriteReturn` use the recommended parenthesised *object* syntax, while `Write` and `ReadReturn`, by virtue of requiring parameters and thus being declared as *case classes*, use the square bracket syntax for *types*.

3.1.2 Component Definitions

Components in Kompics Scala should be defined as classes — indicating their ability to be instantiated multiple times — extending the `sl.ComponentDefinition`

abstract class, which itself extends Kompics Java's `ComponentDefinition` class, but employs a different `ComponentCore` as explained in section 3.2.2.

Port Declarations happen via the `provides` and `requires` methods, and, thanks to Scala's local *type inference*, the verbose type annotations for the fields, as seen in Kompics Java, are now superfluous. As with the event direction declarations in section 3.1.1, the port position declaration methods come in *object* and *TypeTag* variants. However, as Kompics Scala ports should always be objects, in a pure Kompics Scala project only that form should be necessary. The TypeTag variant is used when requiring or providing a port type defined in Kompics Java, such as the `pfd` port in listing 3.5 on line 24. The port position declarations on lines 20, 22, and 23 all assume an implementation in Kompics Scala.

Init events are handled similar to Kompics Java, in that any instance of a subtype of `Init[C]` may be used to instantiate a component of type `C` via the appropriate class constructor. In addition, Kompics Scala provides a generic `case class sl.Init[C <: sl.ComponentDefinition](params: Any*)` extending `Init[C]`, that can be used for pattern matching initialisation arguments as shown in listing 3.5 on line 26[3], for example. This approach is not as (type) safe as using a custom init class for every component definition, but it is often faster to write and more readable. Note that the `peers` field declared on line 26 is marked as immutable by the use of the `val`-keyword, while the fields in lines 28-30 are mutable, as indication by the `var` keyword.

3.1.3 Event Handlers

By far the largest deviation from Kompics Java, is the new syntax for defining event handlers, which is designed around Scala's pattern matching facilities. The general format for handler declarations is:

```
1 aPort uponEvent {
2   case Event1Pattern => { /* body1 */ }
3   /* ... */
4   case EventNPattern => { /* bodyN */ }
5 }
```

The beginning of the expression is taken up by a reference to the port the new handler is supposed to be subscribed to, which happens implicitly at this position, making the most common case the default one. The whole expression also returns a reference to the declared handler, which can be assigned to a field, in case later unsubscription is necessary. Furthermore, it is still possible to create handlers without immediately subscribing them, by using the following form instead, which can then later be subscribed to a port via `subscribe(myHandler, aPort)` as before:

[3] Note that the generic `Address` type parameter of the `List` class is unchecked by Scala in this pattern.

```
1 val myHandler = handler {
2   case Event1Pattern => { /* body1 */ }
3   /* ... */
4   case EventNPattern => { /* bodyN */ }
5 };
```

However, the `uponEvent`-form is strongly recommended wherever applicable, as it completely avoids the common issue of forgetting to subscribe a handler to a port.

Event Patterns can be any patterns supported by the Scala language, though only patterns that can actually match an event type declared on the port being subscribed to have any chance of succeeding, of course. Whether or not a pattern can possibly succeed, however, is not checked by the compiler or the runtime, and no "dead code" warnings will be issued.

It is also important to note, that pattern matching happens *before* the component is scheduled to handle the event being matched, and is thus not guaranteed to have exclusive access to the component's internal state. Because of this, using *guard*-patterns that access (mutable) component state is explicitly (thread) *unsafe*. Consider, for example, a component with a mutable local field `var m: Boolean`, and a pattern of the form `case x if m => ...`,. This code is *not* thread safe, because *m* would be evaluated on a sender component, while it could also be in the process of concurrently being mutated by its owner component. Note that this would not be a problem, if *m* were *immutable*, instead. This particular behaviour may seem slightly unintuitive to a programmer, because it looks like everything in the pattern matching statement is part of the (supposedly thread safe) body of the component. But consider that these patterns are evaluated to make a decision, whether or not to schedule a component in the first place, and thus it must be possible to make such a choice without scheduling the component first. Adding any coordination primitive to pattern checking would have severe scalability impacts on the performance of the implementation. We will discuss subscription-checking and handler execution internals again in more detail in section 3.2.2.

Listing 3.6 shows three examples of possible event patterns with pure Scala events, that is all events are case classes or case objects. In line 40 a simple `Read` object is matched, which carries no additional information, while on line 43 a case class with a field is matched, and a reference called `v` to the field is created, which is then available in the body of the handler. Nested patterns are, of course, also possible, as shown on line 48, where we match the `Deliver` event from the BestEffortBroadcast abstraction, but only if its content has the form `('write, v)`, where `'write` is a *literal symbol* and `from` and `v` are bound names. This pattern perfectly implements the pseudocode from algorithm 2 line 11. Two alternative implementations, that are also possible for this purpose, are 1) the use of a `case object Write` instead of the literal symbol (or "atom") `'write`, or 2) the declaration of a `case class Write(v:Any)`, which can then replace the `('write, v)` tuple. As is typical for Scala, the choice between multiple equivalent solutions is up to the developer's (aesthetic) preference, mostly.

```
39  onrr uponEvent {
40    case Read => {
41      trigger (ReadReturn(value) -> onrr);
42    }
43    case Write(v) => {
44      trigger (BestEffortBroadcast.Broadcast(('write, v)) -> beb);
45    }
46  }
47
48  beb uponEvent {
49    case BestEffortBroadcast.Deliver(from, ('write, v)) => {
50      value = Some(v);
51      trigger (PerfectPointToPointLink.Send(from, 'ack) -> pl);
52    }
53  }
```

Listing 3.6: Event handlers for native Scala events in Kompics Scala.

```
32  pfd uponEvent {
33    case crash: PerfectFailureDetector.Crash => {
34      correct -= crash.p;
35      checkCondition();
36    }
37  }
```

Listing 3.7: An event handler for matching Kompics Java events in Kompics Scala.

As events declared in Kompics Java lack the convenient pattern extraction features that Scala's case classes provide, patterns matching them must usually rely on simple type matching of the form `case e: EventType => ...`, which matches only events of type `EventType` and assigns them the name `e` in the body of the handler. An example of such a pattern can be seen in listing 3.7 on line 33, where we assume that the `PerfectFailureDetector` port and all its events are defined in Kompics Java and we wish to reuse the existing implementation.

3.1.4 Statements and Expressions

Trigger statements were not a big syntactical issue in Kompics Java, but nevertheless have received a slight "face lift" in Kompics Scala, for the purpose of improving readability when nested events are created directly in the a `trigger` expression. The new syntax clearly separates the event from the port, it is being triggered on, by interposing an arrow symbol (`->`) in between them, pointing from event

```
1  package registers;
2
3  import se.sics.kompics.sl._;
4
5  object Main extends App {
6    Kompics.createAndStart[SetupComponent]();
7    Kompics.waitForTermination();
8  }
9  class SetupComponent() extends ComponentDefinition {
10   val peers = cfg.getValue[List[Address]]("config.peers");
11   val selfAddress = cfg.getValue[Address]("config.self");
12
13   val networkLink = create[TCPPerfectLink](Init(selfAddress));
14   val broadcast = create[BasicBroadcast](Init(peers));
15   val failureDetector = create[PerfectFailureDetector](new
      ↪ PerfectFailureDetector.Init(peers));
16   val register = create[ReadOneWriteAll](Init(peers));
17
18   connect(PerfectPointToPointLink)(networkLink -> broadcast);
19   connect(PerfectPointToPointLink)(networkLink -> failureDetector);
20   connect(PerfectPointToPointLink)(networkLink -> register);
21   connect(BestEffortBroadcast)(broadcast -> register);
22   connect[PerfectFailureDetector](failureDetector -> register);
23 }
```

Listing 3.8: Component and connection setup in Kompics Scala.

to port. That is to say, the trigger syntax is of the form `trigger (event -> port)`. Not only does this form improve readability, but it is also intended to evoke a "trigger event onto port"-way of reading, as is familiar from mathematical function definitions, e.g. $\mathbb{N} \to \mathbb{R}$, or from key-value mappings `Map(1 -> 1.0, 2 -> 2.0, ...)`, which are logically similar. In fact, the implementation does make use of Scala's implicit tuple creation syntax, which is also used for maps, among other places. This means that an equivalent syntax of the form `trigger((event, port))` (note the double parentheses) is also allowed, but is generally not recommended as it reduces readability, even compared to Kompics Java.

Component Creation The expression to create a new component of type `C` has the form `create[C](initForC)`, where `initForC` is a valid init event for `C`, that is it must be a subtype of `Init[C]`. The implementation uses TypeTags again, in the same manner as described above. Examples of creating both Kompics Scala and Kompics Java components can be seen in listing 3.8 in lines 13-16, as well as creation of the root component on line 6, which uses `Kompics.createAndStart` instead, implying the implicit sending of a `Start` event.

Channels are created using either of the following two syntactic variants:

```
1 connect(Port)(providingComponent -> requiringComponent)
2 connect[Port](providingComponent -> requiringComponent)
```

The first one is used for Kompics Scala object ports, while the second one is used for Kompics Java ports. Similar to the trigger syntax, this is meant to be read as "connect, on port `Port`, `providingComponent` to `requiringComponent`". However, despite the arrow being one-way, this expression always creates a bidirectional channel with the correct event-directionality, as was described in section 2.2.1. In fact, Kompics Scala has no special syntax currently for one-way channels, as they were a later addition to Kompics Java that is rarely used anyway. If they are desired, Kompics Scala users can simply fall back onto the Kompics Java mechanism to create them, for example writing something like `connect(c1.getPositive(...), c2.getNegative(...), Channel.ONE_WAY_POS)`.

In addition to the form taking two components, it is also possible to pass one or two ports to `connect` instead. In that case the specification of the port type is elided, as it can be inferred from the type signature of the given port or ports. Using ports instead of components as parameters is more convenient when connecting a child component to a port of its parent, for example, a mechanism we refer to as "port chaining". Port chaining allows the creation of nested abstractions in Kompics, which are a very powerful compositional feature previously described by Cosmin Arad [8, p. 22].

An example of how to connect all the dependencies of the `ReadOneWriteAll` component was already seen in listing 3.8 in lines 18-22.

Configuration and Logging A very shallow DSL for using Kompics Java's built-in configuration and logging support is also provided, consisting of nothing more than two "memoised"[4] wrappers `cfg` and `log`, which provide Scala proxy methods to access `config` and `logger` respectively. The `cfg` memo simply provides TypeTag variants of `readValue` `getValue` methods to load a configuration at a particular key and convert it to a desired target type. An example of using `cfg` can be seen in listing 3.8 on lines 10 and 11, where it is used to acquire peer- and self-addresses. The `log` memo simply wraps Kompics Java's `org.slf4j.Logger` into a more convenient `com.typesafe.scalalogging.Logger` instance, which allows, for example, Scala's *string interpolation* to be used.

3.1.5 Simulation Support

Kompics Java also comes with a discrete event simulator framework and an eDSL to describe stochastic processes, both were previously described by Cosmin Arad [5, 8]. A Scala DSL for this framework was also implemented by the author of this thesis, and is used regularly by our students. However, as we are not discussing simulation

[4] "Memoised" means lazily created only when used for the first time, and then cached for later uses.

anywhere else in this dissertation, we are omitting the concrete DSL for simulation support at this point, and would simply like to note that it indeed exists. It can be found found on Github at `https://github.com/kompics/kompics-scala/tree/master/simulator`.

3.2 Kompics Scala Implementation

Kompics Scala is implemented as a Scala library and is published on Bintray at `https://bintray.com/kompics/Maven/kompics-scala`, from where it can be included in other Scala projects as a dependency, for example, using SBT with the following addition to a `build.sbt` file:

```
resolvers += Resolver.jcenterRepo
resolvers += Resolver.bintrayRepo("kompics", "Maven")

libraryDependencies += "se.sics.kompics" %% "kompics-scala" % "1.1.+"
```

The dependency is available for Scala `2.11` and `2.12` at the time of writing, with support for Scala `2.13` in preparation.

Kompics Scala is an open-source project, the sources of which can be found on Github at `https://github.com/kompics/kompics-scala`.

3.2.1 Kompics Java Interoperability

Kompics Scala was designed with keeping interoperability with Kompics Java in mind, so that existing projects could be ported over piece by piece, instead of requiring complete rewrites up front. To that end Kompics Scala reuses much of Kompics Java's code, either directly or by using syntactic sugar as described in the previous section. Interoperability between Java and Scala code in general comes in two forms: Either one is trying to use code written in Java from a Scala codebase, or, conversely, one is trying to use code written in Scala from a Java codebase. Both approaches are supported with Kompics Scala and Kompics Java, but to a different degree of convenience.

3.2.1.1 Java in Scala

The more common case of using existing Kompics Java components, events, and ports in a Kompics Scala project is also the case with better support. Generally, all scenarios of combining Kompics Scala and Kompics Java components are supported from Kompics Scala, and we have already shown multiple examples of how to do so, for example, using the `PerfectFailureDetector` abstraction in listing 3.3 or setting up the component implementing it in listing 3.8.

Apart from Scala's built-in support for interacting with Java by virtue of both languages compiling to Java virtual machine (JVM) bytecode, a few additions to

Kompics Scala's code were necessary to make this interoperability as smooth as possible, namely:

1) Two wrapper classes `sl.NegativeWrapper` and `sl.PositiveWrapper`, which implement the Scala `sl.AnyPort` trait, as well as the `sl.NegativePort` and `sl.PositivePort` traits, respectively, for instances of Kompics Java's `PortCore`. This is necessary for the `uponEvent` syntax as well as versions of `connect` with one or two ports to work with Kompics Java ports.

2) A number of implicit and explicit conversions for Kompics Java types and other types used in the Kompics Java API, such as `java.util.Optional`, to equivalent Scala types.

Additionally, a not insignificant change to the Kompics Java codebase was necessary to accommodate the new Kompics Scala implementation of components with pattern matching handlers. The details of the Scala implementation will be described in section 3.2.2, but we will note at this point, that this required the separation of the original `Component` class from Kompics Java, which contained all the logic for handler execution and lifecycle management into a cross-language reusable `ComponentCore` class and a Kompics Java specific `JavaComponent` class. This allowed for a separate implementation of a `ScalaComponent` class extending `ComponentCore` for Kompics Scala. For the same reason and in the same manner, the Kompics Java `Port` class has been split up into `PortCore` and `JavaPort`, and an implementation of `ScalaPort` has been added, that supports subscriptions-checking based on Kompics Scala's pattern matching handlers.

3.2.1.2 Scala in Java

When designing and implementing Kompics Scala, we have made no special accommodations whatsoever to make it easy to use components, ports, or events implemented in it from Kompics Java. However, as with all Scala code, doing so is generally possible, albeit with some caveats. We will begin this section by describing how Kompics Scala definitions can be used from Kompics Java, and afterwards we discuss limitations of this approach.

To make our explanations as concrete as possible, assume that we wish to use the `OneNRegularRegister` port and its events, which were shown in listing 3.4, from a Kompics Java component. We could, for example, design a parent component that invokes operations on the register, by sending it `Read` and `Write` events, and handling `ReadReturn` and `WriteReturn` events. Such an implementation can be seen in listing 3.9.

Paths and Names The use of the many dollar-signs in type names and field names results from the way the Scala compiler translates objects, case classes, and case objects into JVM byte code. It is crucial here, to write or elide the dollar signs in the correct places, in particular in the handler signatures used for event matching at

```
1  class RegisterParent extends ComponentDefinition {
2    Positive<OneNRegularRegister$> onrr =
     ↪  requires(OneNRegularRegister$.class);
3
4    Handler<OneNRegularRegister$.ReadReturn> readHandler = new
     ↪  Handler<OneNRegularRegister$.ReadReturn>(){
5      @Override
6      public void handle(OneNRegularRegister$.ReadReturn event) {
7        Object v = event.v().get(); // v is read value
8        trigger(OneNRegularRegister$.Read$.MODULE$, onrr); // read the
         ↪  register
9      }
10   };
11
12   Handler<OneNRegularRegister$.WriteReturn$> writeHandler = new
     ↪  Handler<OneNRegularRegister$.WriteReturn$>() {
13     @Override
14     public void handle(OneNRegularRegister$.WriteReturn$ event) {
15       trigger(OneNRegularRegister$.Write$.apply(5), onrr); // write 5
16     }
17   };
18 }
```

Listing 3.9: Using a Kompics Scala port from Kompics Java.

runtime. For example, writing `Handler<OneNRegularRegister$.WriteReturn>` instead of `Handler<OneNRegularRegister$.WriteReturn$>` may not generally lead to a compile-time error, as the Scala compiler will in fact generate *both* classes for a case object like `WriteReturn`, but with different content. In this particular case, however, it *would* lead to a compile-time error, because `OneNRegularRegister$.WriteReturn$` does not implement the `KompicsEvent` interface, which is an upper type bound for the generic type parameter of the `Handler<>` type. Even though a compile-time error is produced, the message of the error may be somewhat misleading in this case, as it suggests that `WriteReturn` does not implementm`KompicsEvent`, which it does, of course. Furthermore, the same applies the other way around for `OneNRegularRegister$.ReadReturn$` and `OneNRegularRegister$.ReadReturn`, which is a case class instead of a case object.

Instance Creation When it comes to instance creation, a Scala (case) object's singleton instance can be accessed via the `MODULE$` field in Java, as seen on line 8 in listing 3.9. The equivalent of creating a case class instance via the apply syntactic sugar, such as `Write(5)` in Scala, can be achieved in Java by desugaring to `Write$.apply(5)`, as seen in line 15.

Instances of Kompics Scala components can generally be created as usual,

since they are just simple classes and typically do not have a companion object. However, the creation of `Init<>` instances follows the same rules as for events above. If the Scala component expects a generic `sl.Init` instance, instead of a custom case class, the construction would require the programmer to fill an instance of `scala.collection.immutable.Seq<Object>` from the Scala standard library[5] with the correct arguments in the correct order first. This instance must then be passed to the static `sl.Init.apply` method to create an `sl.Init` instance, which can finally be passed as usual to the `create(...)` method.

Channels and Ports Java *port instances* can easily be created from Scala *port types* as shown on line 2 in listing 3.9, with the caveat about naming described above. Since the created port instances are in fact Kompics Java instances, they can be connected via channels without any issue.

Limitations There are two major limitations when using Kompics Scala definitions from Kompics Java:

1) It is not possible to subscribe a Kompics Java handler to a Kompics Scala port instance, as the instance expects pattern matching functions, which are not syntactically available in Java. It may be possible to work around this, by creating a Java implementation of `scala.PartialFunction[Any, Unit]` manually, but the author of this dissertation has never attempted this.

2) Much of the convenience in using Java from Scala comes from the availability of *implicit conversions* in Scala, which are, of course, not available in Java. Thus any custom classes created to wrap Kompics Scala implementations in Kompics Java accessible interfaces, for example, would have to be created manually when needed, instead of relying on the compiler as we do when using Java from Scala. Of course, this is "merely" an inconvenience.

3.2.2 Subscription-checking and Handler Execution

In this section, we will describe in detail how Kompics Scala performs subscription-checking and handler execution. Before we do so, however, we will first introduce some new notational shorthands, and then take a step back and expand on the mechanism employed in Kompics Java, that was already alluded to in section 2.2.2.2.

3.2.2.1 Notation

In order to make our description in the next sections more concise, we introduce some additional notation for port instances and handler matching. To do so, we will repeatedly use the following very simple example setup, which is also depicted

[5] This clearly implies the need for having the Scala standard library available at runtime when using Kompics Scala from Kompics Java.

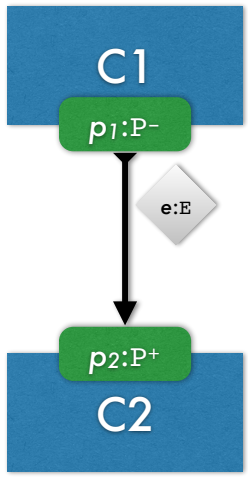


Figure 3.1: Visualisation of the examples used to explain subscription-checking: C_1 and C_2 are components, p_1 and p_2 instances of port type P connected via a channel, and e is an event of type E.

in figure 3.1: Consider the case where we have two components C_1 and C_2. C_1 has a single *required* port instance p_1, which is of type P, while C_2 also has a port instance p_2 of the same type, but its instance is *provided*. In our example, C_1 wants to send an event e of type E to C_2, by using their respective port instances of type P. The subject we are interested in for the next few sections is 1) whether this is legal with respect to the Kompics model (see [8, appx. B]) and 2) what is the concrete mechanism employed to check this and then select and execute appropriate handlers.

As concurrency concerns will become important in our discussion of mechanisms, we will use the term "the *sending thread*" for the thread C_1 is executing on when it triggers e, and similarly term the thread C_2 uses to handle e (if it does so), "the *receiving thread*". Note that it can, of course, happen in a real execution that both threads are the same, but for the sake of making the example more general, we will assume that they are indeed different threads.

We introduce the following new notation:

Port Directions are indicated by writing $\texttt{P}^-$ for a *required* port of type P and similarly $\texttt{P}^+$ for a *provided* port. Thus, we can simply write $p_1 : \texttt{P}^-$ to indicate the fact that p_1 is a required port instance of port type P. We will also write $\texttt{P}^?$ as a placeholder for either a $\texttt{P}^-$ or a $\texttt{P}^+$, when the context allows us to make more general statements valid for either port direction.

Event Directions were introduced in section 2.2.1, and we add here a shorthand for indicating the direction an event may pass through a port: For any event type ϵ and

port instance p, we write $e \in out(p)$ to indicate that e is *outgoing* on p, and similarly $e \in in(p)$ to indicate it is *incoming*. We will use the same notation to generalise from a concrete port instance p of a port type $\mathtt{P}$ to all instances of $\mathtt{P}$ of the same direction, by writing $e \in out(\mathtt{P}^-)$, for example. More formally, we make the following definitions: Let $e : \mathtt{E}$ be an event of type $\mathtt{E}$, I and R two sets of event types, and $\mathtt{P} = (I, R)$ a port type with indications I and requests R. Then we write:

$$e \in out(\mathtt{P}^+) \quad \text{iff } \exists\, \mathtt{E}' \in I \text{ s.t. } \mathtt{E} <: \mathtt{E}' \tag{3.1}$$

$$e \in out(\mathtt{P}^-) \quad \text{iff } \exists\, \mathtt{E}' \in R \text{ s.t. } \mathtt{E} <: \mathtt{E}' \tag{3.2}$$

$$e \in in(\mathtt{P}^+) \quad \text{iff } \exists\, \mathtt{E}' \in R \text{ s.t. } \mathtt{E} <: \mathtt{E}' \tag{3.3}$$

$$e \in in(\mathtt{P}^-) \quad \text{iff } \exists\, \mathtt{E}' \in I \text{ s.t. } \mathtt{E} <: \mathtt{E}' \tag{3.4}$$

$$e \in in(p) \text{ or } e \in out(p) \quad \text{iff } p : \mathtt{P}^? \text{ and } e \in in(\mathtt{P}^?) \text{ or } e \in out(\mathtt{P}^?) \tag{3.5}$$

Handlers have very different syntax and semantics in Kompics Scala than they have in Kompics Java. In order to be able to compare the two, we would like a common way of expressing handler types for both models. Conveniently, Kompics Java's usage of (sub-)type checking is actually a special case of Kompics Scala's pattern matching for handlers. This allows reuse a notation that can express all of Kompics Scala, and simply specialise it for Kompics Java. We are thus going to use the following abstract notation for handler types: For any *pattern* $\mathcal{P}$, the notation $\mathcal{H}\langle\mathcal{P}\rangle$ describes the class of all possible handler implementations which handle all events matched by $\mathcal{P}$. The term *matching* here has the intuitive meaning of producing a boolean TRUE value when applied to a concrete event instance. We will ignore the name bindings, that usually happen as part of programming language pattern matching facilities, for now. In addition to all patterns allowed by the Scala language, there are two special cases we are interested in:

1) Handlers of the form $\mathcal{H}\langle \_ : \mathtt{E}\rangle$ match any event $e : \mathtt{E}$ for some event type $\mathtt{E}$. These handlers correspond to the semantics used by Kompact, which we will introduce later in chapter 8.

2) Let $\mathtt{E}$ and $\mathtt{E}'$ be two event types. Handlers of the form $\mathcal{H}\langle \_ <: \mathtt{E}\rangle$ match any event $e : \mathtt{E}'$ such that $\mathtt{E}' <: \mathtt{E}$. These handlers correspond to the semantics used by Kompics Java.

Note that this handler notation does not describe the *behaviour* of a handler, but only the set of events it will handle. As there can be an almost infinite variety of possible behaviours for the same event set, we write $h : \mathcal{H}\langle\mathcal{P}\rangle$ to describe a concrete handler instance h for the handler class $\mathcal{H}\langle\mathcal{P}\rangle$.

3.2.2.2 Rules

The rules that describe which event-trigger operations and which handler subscriptions are *legal* are described previously by Cosmin Arad [8]. We will rephrase them

in this section using the notation established above for convenience. Additionally, we also consider the question of which event-trigger operations are *relevant* for a component. Events are *relevant* if they are going to be handled by a component. Clearly, all events that are illegal can not be relevant, but the opposite is not true. An event that is ignored by a component is considered irrelevant, even if it is legal.

Legal Operations Our example event e can legally pass from C_1 to C_2 via p_1 and p_2 (refer to figure 3.1), if the following conditions are met:

$$e \in out(p_1) \tag{3.6}$$

$$e \in in(p_2) \tag{3.7}$$

As the connection of two port instances via a channel is allowed in Kompics only if both instances are of the same port type, but with opposite directionality, it is sufficient to verify equation 3.6, as 3.7 follows automatically.

Proof. To see why this is the case, consider the following: Assume that $e \in out(p_1)$ (eq. 3.6) and, without loss of generality, assume $p_1 : \mathtt{P}^-$. The rules for channel connection require that $p_2 : \mathtt{P}^+$. Let $\mathtt{E}'$ be an event type, which is a request of port type $\mathtt{P}$ and a supertype of the type of e, by equations 3.5 and 3.2. Then it follows from applying equation 3.3 followed again by equation 3.5 that $e \in in(p_2)$ (eq. 3.7). □

Relevant Events A legal event e incoming at p_2 is only relevant, if at least one handler $h : \mathcal{H}\langle\mathcal{P}\rangle$ is subscribed to p_2, such that its pattern $\mathcal{P}$ matches e. Formally, let S be the set of all handlers subscribed to p_2, then an event e is *relevant* if and only if

$$\exists h \in S \text{ and } \exists \mathcal{P} \text{ s.t. } h : \mathcal{H}\langle\mathcal{P}\rangle \text{ and } \mathcal{P}(e) = \textsc{true} \tag{3.8}$$

When we handle an event, we must execute all subscribed handlers that match it. We will call the collection of all these handlers the *set of relevant handlers* for an event e and a port p. Clearly this must be a subset of the set S_p of all handlers subscribed to p, and so we will use the — perhaps familiar — "limited-to" notation: $S_p|_e$. For any event e and port p, the *set of relevant handlers* is defined as:

$$S_p|_e = \left\{ h \in S_p \mid \exists \mathcal{P} \text{ s.t. } h : \mathcal{H}\langle\mathcal{P}\rangle \text{ and } \mathcal{P}(e) = \textsc{true} \right\} \tag{3.9}$$

Given equations 3.8 and 3.9, it is clear that the statement "e is relevant" is equivalent to $S_p|_e \neq \emptyset$, and this serves as the motivation for the terminology.

3.2.2.3 Kompics Java

Sending Side The implementation of Kompics Java will perform the following steps on the *sending thread*, when C_1 triggers e on p_1:

1) Verify equation 3.6.

2) For each channel c connected to p_1, pass e to the port instance on the other end of c. In our example we only have a single channel, where the port instance is p_2.

3) Forward e along all channels connected to p_2 (repeat from the previous step while possible). In our example there are no internal connections.

4) Check if e is *relevant* by trying equation 3.8 on each subscribed handler $h \in S_{p_2}$ until it either succeeds or no more handlers remain. In the latter case e is irrelevant and the following steps are *not* executed.

5) If e is *relevant*, enqueue it on p_2's internal port queue.

6) If C_2 (the parent component of p_2) is not already scheduled to be executed, scheduled it now.

Note that for performance reasons handlers of the same type are grouped on a port, such that the worst case runtime of the subscription-check, which occurs when no match is found, grows linearly in the number of different *subtypes* of all the *incoming* types on the port that have handlers subscribed to them.

Receiving Side Once C_2 is executed on the *receiving thread*, it will dequeue e from p_2[6] and then compute the set of relevant handlers, $S_{p_2}|_e$. This is achieved using the same approach as described above for checking whether e is relevant, but without stopping on the first success, and instead collecting all successful matches. Assuming $S_{p_2}|_e$ is not empty, the `h.handle(e)` method is then invoked for each handler $h \in S_{p_2}|_e$. No assumptions on the order of handler execution should be made, as no guarantees are given by the model or the implementation, but typically they will be invoked in subscription order. As `h.handle(e)` is executed on the *receiving thread*, and components are guaranteed to be only scheduled on a single thread at a time, access to the internal state of C_2 is (thread) safe.

Note that due to dynamic handler subscription and unsubscription it can technically happen for $S_{p_2}|_e$ to be empty at this point, despite only relevant e being queued for execution. This circumstance occurs only if all handlers matching e have been unsubscribed by another event, which was handled *after* e was enqueued, but *before* e was dequeued again. If this occurs e is simply ignored, and we only wasted some cycles scheduling C_2 unnecessarily. However, the semantics for such components, where the set of handled event types changes at runtime, are very difficult to understand for developers. It is thus strongly recommended to keep the set of handled event types constant at runtime, and only use dynamic subscription to change the *behaviours* of the handlers for each type.

[6] For simplicity we will ignore fairness concerns over multiple ports, and event batching here, and assume e is the only event currently queued.

Event Matching In Kompics Java, any handler h has the form $h : \mathcal{H}\langle \_ <: \mathtt{E}_h \rangle$, where $\mathtt{E}_h$ corresponds to the generic type parameter of the `Handler<>` class. Thus, in order to evaluate the pattern of h, the Kompics Java implementations concretely executes `h.eventType.isInstance(e)`, where `h.eventType` is a stored reference to the Java `Class<>` instance corresponding to $\mathtt{E}_h$.

Handler Subscription Technically, a handler $h : \mathcal{H}\langle \_ <: \mathtt{E}_h \rangle$ in Kompics Java is only *legal* on a port p_2 iff $\mathtt{E}_h$ is *incoming* on p_2. However, violating this rule would have no observable effect at runtime, since handler subscriptions are only checked *after* event legality has been verified. Thus no illegal event could ever be handled by such an illegally subscribed handler, and the behaviour of the handler would simply be dead code. Nevertheless, Kompics Java's implementation does verify the legality of handler subscription at the moment the handler is being subscribed, as illegally subscribed handlers always indicate a logic mistake by the programmer, which should be caught as early as possible.

3.2.2.4 Kompics Scala

The Kompics Scala implementation proceeds in much the same way as Kompics Java when C_1 triggers the event e on its port instance p_1. However, the differences in handler internals require some deviations in how exactly relevance is checked and how relevant handlers are then executed, which we will describe in this section.

While a Kompics Java handler only consists of a single function body together with a single event type it is interested in, the body of a handler in Kompics Scala is a `match`-statement, which itself can contain a number of patterns mapping to a particular body to be executed for each pattern. Thus, not only are we checking in linear time through all subscribed handlers S_{p_2}, but additionally, we must test against each pattern in the body of each handler. Furthermore, during step 4 on the sending side, we do not in fact want to execute the body of any pattern within a handler immediately when it is determined to match. Instead, we would prefer to get a "match or no-match"-answer as quickly as possible and then execute the body associated with the matched pattern later on the *receiving thread*.

The fact that we need to defer body execution and the cost of linearly checking through a number of (potentially large) Scala patterns, leaves us with two possible approaches for an efficient implementation:

1) We could try to get a "match or no-match"-answer as quickly as possible, to reduce work on the *sending thread*, at the cost of having to perform up to the whole procedure again on the *receiving thread*, if it so happened that the only matching pattern was the last one to be checked.

2) We could try to avoid duplicating work, by performing a full match on the *sending thread* already and remembering all bodies that were matched, including all variables that were bound during the pattern's evaluation.

Quick Check From the perspective of Scala, a handler $h : \mathcal{H}\langle\mathcal{P}\rangle$ is nothing more than an implementation of the `PartialFunction` trait with the signature `KompicsEvent => Unit`. This trait provides a method `isDefinedAt(e)`, which returns `true` if e is part of the handler's domain, i.e. there exists at least one matching pattern (or the function is actually total, of course). As this method perfectly captures our notion of $\mathcal{P}(e) = \text{TRUE}$, we can use it to evaluate equation 3.8.

While this method is faster than `PartialFunction.apply(e)`, which evaluates the body of the matched pattern, and can sometimes return early when a match is found, it does not produce the matched handler body in a form that we could use to evaluate it later on the *receiving thread*. However, we could simply proceed the same way Kompics Java does from step 5, enqueue e and schedule the component. Once C_2 is scheduled, we can run `PartialFunction.apply(e)` on the *receiving thread* against all subscribed handlers to evaluate all matching bodies.

This approach shares the disadvantage of the Kompics Java implementation, that is, its complexity, since in the worst case it may have to try through $n-1$ handlers twice in vain, to find the one matching handler at the very end of the list. As Kompics Scala must check linearly through handlers, not just types, and each individual handler check can be complex, depending on the size of the pattern(s) used, the problem of double matching is significantly exacerbated in Kompics Scala compared to Kompics Java.

Avoiding Double Matching An alternative implementation would be to skip the `isDefinedAt(e)` step completely, and immediately call `apply(e)` on all subscribed handlers. Since we do not actually want execute the matching bodies on the *sending thread*, but only to record them, we would have to modify the handler's signature to be `KompicsEvent => (() => Unit)`. That is, instead of evaluating the body, we return a closure containing the body as well as all bound variables. We thus construct the set of relevant handlers in a way equivalent to equation 3.9 and use our previous observation, that $S_{p_2}|_e \neq \emptyset$ implies "e is relevant", in order to perform step 4. In step 5 we then enqueue the event together with all such closures we created on the port, and then schedule the component.

When C_2 is scheduled on the *receiving thread*, the stored closures are dequeued and invoked one by one, guaranteeing (thread) safe access to C_2's internal state, without having to perform another check against e. Note that, *if* Kompics Scala is supposed to have exactly the same semantics as Kompics Java, we would need to check if the handler that produced the closure is actually still subscribed at this point. If this is not a concern, the check can be elided.

Either way, this approach avoids the potentially expensive double matching, but at the cost of creating a number of closures, storing all of them on a concurrent queue temporarily, and then later deallocating them after they have been executed. This puts significant additional stress on Java's memory allocation and garbage collection mechanisms.

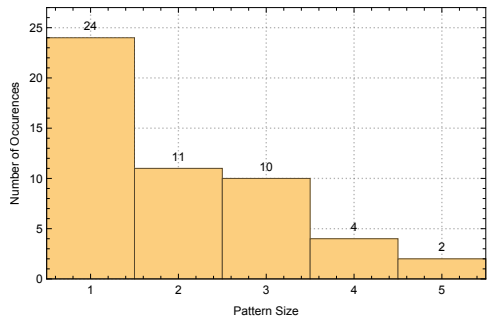


Figure 3.2: Frequency distribution of Kompics Scala pattern sizes in the MPP benchmark suite.

Which of these two alternatives performs better in practice depends on the relative cost of pattern matching, compared to the cost of creating, storing, and deallocating closures. Microbenchmarks give a conservative approximation of the crossover point, where the second approach becomes more efficient. It appears to happen at patterns of a width or depth of three. Based on this information, for example, a pattern like `MyEvent(a, Some(b: Int), c: None)`, which is both width and depth 3, should be about equal in performance for either implementation option, while smaller patterns would favour the first option, and larger pattern the second option.

When we originally implemented and presented Kompics Scala [61], we made the assumption that patterns of size 3 and larger would be fairly common. We thus implemented the closure-based second option, in an attempt to avoid the costly double matching overhead. Over the last few years of using Kompics Scala in education, however, it has become clear that patterns larger than size three are actually fairly rare.

Consider for example the frequency distribution of pattern sizes in the benchmarking suite we describe in chapter 6, shown in figure 3.2. Average and median pattern size is only 2 and the largest pattern used is only of size 5. In fact, it turns out that almost half, 47 % to be precise, of patterns used are very small, only matching a single object, class, or field. In this particular project, only 31 % of all handlers benefit from the closure-based implementation, and only 12 % are expected to actually perform better with it, than with the simpler quick-check implementation. And this analysis is based on an estimate, which we already established to be conservative, in the sense that it likely underestimates the crossover point. While these results only cover a single project, we have found that they are fairly representative of the code our students tend to write in Kompics Scala.

For these reasons, we have implemented a new version of Kompics Scala, which we will refer to as Kompics Scala `2.x` in the future, which uses the quick-check based approach to handler subscription. The DSL described in section 3.1 refers to this newer version Kompics Scala. It only differs in from the Kompics Scala `1.x` DSL,

Algorithm	Java	Scala	Reduction Factor
Basic Broadcast	66 LOC	28 LOC	2.4×
ROWA $(1, N)$ Regular Register	121 LOC	68 LOC	1.8×
RIWCM (N, N) Atomic Register	118 LOC	90 LOC	1.3×
Increasing Timeout EPFD	109 LOC	72 LOC	1.5×
Sequence Paxos	712 LOC	181 LOC	3.9×

Table 3.1: Code maintainability comparison based on LOC between Kompics Java and Kompics Scala.

described previously in [61], by the type signature of handlers, and the deprecation of the additional `handle` keyword, which was used to create of the required closure, instead of evaluating a handler body associated with a pattern.

3.3 Evaluation

Having provided a new implementation for Kompics in the form of a DSL embedded in Scala, we must now turn to judge how well we did with regards to our goal of improving (student) experience when working with Kompics and what price we are paying for any such improvements gained, if any. From the descriptions and worst case runtime analysis in section 3.2.2 it should be fairly clear that for the closure-based solution, Kompics Scala `1.x`, we expect a performance degradation compared to the Kompics Java implementation under most circumstances, since a) complex patterns are more expensive to match than simple types and b) simple patterns still incur the overhead of the closure creation, since we decided to optimise for complex patterns. However, for the quick-check solution, Kompics Scala `2.x`, the relationship is not as clear, as the two mechanisms are very similar now. We perform a simple benchmark experiment to test these expectations and gain some insights into the magnitude of the differences here. A more thorough analysis can be found as part of chapter 6, where we compare the performance of a number of state-of-the-art message-passing systems, including all Kompics versions. Student experience, on the other hand, is difficult to measure, and we have to rely on self-reporting and some meta information gained from various course results here.

Code Maintainability What can generally be said, however, is that the amount of code to maintain is reduced — sometimes drastically — by Kompics Scala. This is partly because of its more concise DSL, but also partly because the surrounding code in Scala is simply less verbose than the equivalent Java code. A common measure for code maintainability is the number of lines of code (LOC), and table 3.1 shows size reduction comparisons between example implementations of the following common algorithms between Kompics Java and Kompics Scala:

- The *Basic Broadcast* implementation of the *Best Effort Broadcast* abstraction, as shown earlier in algorithm 1 and listing 2.3
- The *Read-One Write-All* implementation of the $(1, N)$ *Regular Register* abstraction, as shown earlier in algorithm 2 and implementations 3.3, 3.1, and 3.2
- The *Read-Impose Write-Consult-Majority* algorithm for an (N, N) *Atomic Register* from [16, p. 168-169]
- The *Increasing Timeout* algorithm for implementing an *eventually perfect failure detector* $\Diamond\mathcal{P}$ from [16, p. 55]
- The *Sequence Paxos* algorithm for distributed *sequence consensus* we teach as part of our distributed systems course.

It can clearly be seen that on those algorithms alone, Kompics Scala saves between 20% and 75% of source code lines compared to Kompics Java.

3.3.1 Performance

In order to test handler matching performance, we replicated an old Akka experiment [82] that is essentially an implementation of *Throughput Ping Pong*. Two components (or actors) C_1, C_2 are set up, such that every PING message from c_1 is answered by a PONG message from c_2, until a configured maximum of PING-PONG-pairs $n_{\max}$ is reached, at which point a latch is triggered and the total execution time T is recorded. A value for the average throughput of the run is obtained by $\frac{2n_{\max}}{T}$. If, however, C_1 would be idle while waiting for a response from C_2, we would be measuring a form of round-trip time (RTT), rather than throughput. To avoid this, a number n_{batch} of messages is sent by C_1 before waiting for a PONG the first time. This ensures that handler matching always operates under load and the frameworks' internal batching mechanisms can kick in (see section 2.2.2.1). We also want to see how performance scales over multiple central processing unit (CPU) cores, and thus will launch a number n_{pairs} of independent "Pinger"-"Ponger"-pairs. Since more pairs send more messages, the formula for the message throughput changes to $\frac{2n_{\max}n_{\text{pairs}}}{T}$.

This experiment was originally implemented in Kompics Java, Kompics Scala `1.x`, and also Akka (Scala) as a reference for another Scala based framework and the result were presented in [61]. In order to also compare our new implementation of Kompics Scala `2.x`, we reimplemented the experiment as part of the MPP benchmark suite (chapter 6). We will report the newer experiment, as it supersedes previous results.

Setup The experiment was performed on Amazon's AWS platform, using a single `c5.metal` instance with 2019 specifications: 2 Intel® Xeon® Platinum 8124M CPUs

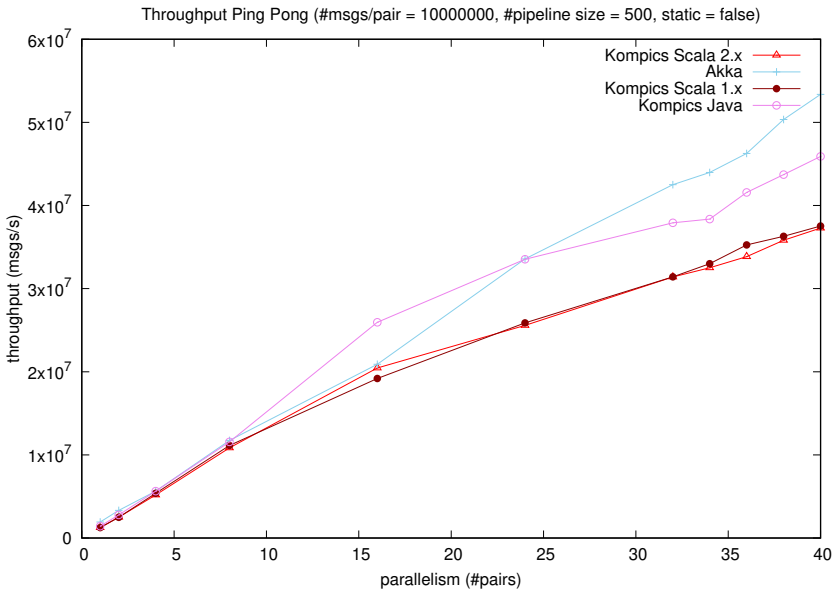


Figure 3.3: Throughput in number of messages for increasing n_{pairs} for Akka (Scala), Kompics Scala 1.x and 2.x, as well as Kompics Java.

at 2.9 GHz with 192 GB of main memory. More details on the environment can be found in section 6.4.1.1.

Akka's dispatcher throughput and Kompics' $\eta_{\max}$ were set to 50, which Nordwall [82] determined to be a good tradeoff between fairness and throughput in this kind of experiment. $n_{\max}$ was set to ten million PING-PONG-pairs, $n_{\text{batch}} = 500$, and n_{pairs} was varied from 1 to 40 as part of the benchmark.

Observations As expected, the results in figure 3.3 show that Kompics Java slightly outperforms both Kompics Scala versions in this simple microbenchmark, achieving around 1.2 × the performance at $n_{\text{pairs}} = 40$. It is also not a surprise that Akka's simpler always-schedule model, where almost all of the work happens on the *receiving thread*, significantly outperforms all Kompics implementations, especially at higher parallelism. Akka peaks at about 53.3 million messages per second (msgs/s), while Kompics Java peaks at 45.9 million msgs/s, and the two Kompics Scala versionss peak at 37.5 and 37.2 million msgs/s.

It is interesting to note that in this very simple benchmark, which *should* favour Kompics Scala `2.x`, no significant differences between the two Kompics Scala implementations can be observed. This indicates that there exists another factor beyond handler matching efficiency, which dominates this benchmark. We will see later in chapter 6 that there are, however, tasks on which Kompics Scala `2.x`

Year	Number of student groups using Kompics Java	Kompics Scala
2017	20 (77 %)	6 (23 %)
2018	4 (31 %)	9 (69 %)
2019	1 (8 %)	11 (92 %)

Table 3.2: Development of Kompics Java and Scala usage in on-premise courses.

significantly outperforms Kompics Scala `1.x`, and sometimes even Kompics Java.

While these results indicate a slight scalability issue for Kompics Scala, this is not a huge issue for its use in an educational context. In this context, code is typically run on fewer cores, and with less load, and the concrete performance and scalability are not crucial, unless they are cripplingly poor. Based on these results, we expected no performance issues with Kompics Scala's use for education, and indeed we have observed none over the last three years of usage.

3.3.2 Education

Kompics Scala has been used at KTH both for online courses and on-premise since 2017.

Online Courses In 2017 we used Kompics Scala exclusively to design and run a sequence of two online courses for the *edX* platform, which are similar in material to the on-premise version described above. They had, however, the additional requirement that all the grading had to happen in an automated manner, as manually grading course work for their ~ 5000 enrolled students would simply have been intractable. To achieve this, we designed a series of seven programming exercises ranging from a simple *Game of Life* [25] example, that had nothing to do with distributed algorithms at all, to a rather involved Sequence Paxos exercise. The exercises were implemented as Apache Zeppelin[7] notebooks with built-in validation of algorithm properties using the Kompics Simulator framework [8, 6, 99]. Student response to the course was overwhelmingly positive, and what issues did arise were mostly related to the validation framework, a task that is very difficult to make general enough, and not to Kompics Scala DSL.

On-Premise Courses We have been using Kompics Scala alongside Kompics Java since 2017 in an on-premise course on distributed algorithms. The coursework involves the development of a project consisting of a key-value store using the algorithms taught in the course. The choice of implementation language is left up to the students, with project templates provided in both Java and Scala (since 2018). Many students in the course have, in fact, never programmed Scala before and

[7] `https://zeppelin.apache.org/`

face the additional challenge of learning both a new programming language and a new framework at the same time as learning the more theoretical material of the course. However, despite this, table 3.2 shows that the number of students choosing Kompics Scala has steadily increased over the last three years, not least due to the introduction of the programming exercises from the online course, which are only available in Scala, first optional in 2018 and then mandatory in 2019. The ability to test the correctness of their implementation in the exercise environment, and then simply copy&paste them into the project code, has convinced many students to tackle the additional challenge of learning Scala.

Informal interviews conducted after the course showed that Kompics Scala users had a much easier time translating algorithms into practical code, especially since the introduction of the programming exercises.

3.4 Related Work

While Kompics Scala's application is not limited to educational use, that is its area of focus, and thus other languages and frameworks being used for teaching distributed algorithms are of most interest.

The *Tempo language* [75] is a formal modelling and verification language used at MIT by Nancy Lynch. It is based around a timed extension of the well-known *Input/Output Automata formalism* [73, 74], which we also use as a formal model for teaching. While based around a similar formalism, as a modelling language Tempo is not suited for writing distributed applications, and has thus a very different focus from Kompics Scala.

The *Go language* [40], with its lightweight threads and typed channel abstractions, provides an implementation inspired by CSP model [50, 89]. It is a good fit for many practical distributed systems problems, and used for teaching them at Princeton.

Actor model implementations like *Akka* and *Erlang* (see section 2.1) are, of course, also particularly suitable for education in distributed systems, especially when the goal is more in efficient systems development than following strict formalisms. In fact, Erlang is used in a more basic distributed systems course at KTH as well.
One important difference to note is that, as opposed to Akka, where any subtype of the `Any` top type is a valid message, in Kompics Scala (and Kompics Java) only types extending the `KompicsEvent` interface/trait are allowed to be used as events, thus limiting events to classes, objects, case classes and case objects. While this is a seemingly unnecessarily limitation, it enforces the notion that *values*, such as numbers or strings, should always be tagged with their intended meaning to form *events*, and thus improves readability of the resulting code, if applied properly.

The *Oz language* [68, 105] is also used in distributed algorithms education at the Catholic University of Louvain. It provides the notion of distributed dataflow variables, which form a very powerful model for distributed computation, which can, for example, easily implement the actor model, making its model more general. However, Oz' lack of static typing makes it more difficult to express the Kompics

model, and causes a similar concern about readability as we described above for the actor model.

CHAPTER

4

Preventing Bugs at Compile Time

In this chapter, we describe a transpiler from a language implementation of the Kompics component model, called Kola, to Java code, reusing the existing Java implementation of Kompics at runtime. We classify common types of errors in Kompics and formalise the typing rules used to avoid. We then show how this language can statically prevent, or at least warn about, a number of these common Kompics mistakes.

In the previous chapter 3 we have seen how a well-chosen DSL can improve code maintainability and reduce simple "transcription" errors, introduced when converting algorithm pseudocode into actual code in a real programming language. This approach, however, does nothing to *prevent* even obvious errors, like sending a message or an event to an actor or a component that does not handle such events.

Consider, for example, the two Actor model snippets shown in listing 4.1: Both actors expect a *ping* message, but are consequently being sent a *pong* message instead. Not only does this clearly misbehaving code not result in a compile-time error in either implementation, but in fact is does not result in a runtime exception either. Even worse, in the Erlang listing 4.1a, the `pong` message is going to remain in the mailbox of `Actor1` forever, and any subsequent such messages would slowly start filling up the mailbox and leaking memory.

Now consider a similar example in the two Kompics implementations, given in listings 4.2 and 4.3, which do suffer from some additional boilerplate compared to the Actor implementations, but also exhibit an important difference in behaviour: They *fail* at runtime with an exception at lines 25 (in 4.2) and 21 (4.3), because that trigger statement is invalid. As we defined PING as a *request* and PONG as an *indication* on `PingPongPort`, the Kompics runtime library knows that $\text{PONG} \notin \mathit{out}(\texttt{PingPongPort}^{-})$ and refuses to process the trigger. Not only does Kompics fail with an exception, instead of silently misbehaving, it also tells the developer exactly *where* and *why* the issue occurred. If we had made the opposite mistake of subscribing to `Pong` in `Component1` instead, we would have gotten an exception during the subscribe invocation there, since the runtime knows that $\text{PONG} \notin \mathit{in}(\texttt{PingPongPort}^{+})$ either.

But, frustratingly, we have to run the program to find out that we made these errors, even though all the necessary information to detect them is actually available at compile time already. Neither the Kompics model nor any of its implementations allow for dynamic changes in the events carried on a port, and thus detecting whether or not a trigger or subscription is valid should really be done at compile time, rather than deferring until the execution of the code happens to reach the point where the mistake occurs. In large codebases, reaching a rare edge case containing the error could conceivably happen hundreds of hours into a distributed deployment, potentially causing subsystems to fail and requiring at lot of developer effort to find the appropriate log file with the original root exception and trace it back to the actual piece of code that caused the issue. But while the information is available at compile time, it is not expressed in a manner that is understood by either Java's or Scala's type system. There simply is no rule correlating an event type `E` with its direction on a port `P` and that port's position on a component, expressed in Java's or Scala's type system as part of the respective DSLs.

In this chapter we will explore common errors and issues when using Kompics Java, in particular, that can be detected and potentially prevented at compile time. We will then give typing rules that capture those errors and describe a system called

```erlang
Actor1 = spawn(fun() ->
  receive
    ping ->
      io:format("Got a ping!~n")
  end
).
Actor1 ! pong
```

(a) Erlang.

```scala
case object Ping;
case object Pong;
class Actor1 extends Actor {
  override def receive = {
    case Ping =>
      println("Got a ping!")
  }
}
val actor1 =
 ↪  sys.actorOf(Props[Actor1]);
actor1 ! Pong
```

(b) Akka (Scala).

Listing 4.1: Two Actor programs that compile and run, but do not behave as intended.

```java
class Ping extends KompicsEvent { /* ... */ }
class Pong extends KompicsEvent { /* ... */ }
class PingPongPort extends PortType {{
  request(Ping.class);
  indication(Pong.class);
}}
class Component1 extends ComponentDefinition {
  Negative<PingPongPort> ppp = provides(PingPongPort.class);
  { subscribe(pingHandler, ppp); }
  Handler<Ping> pingHandler = new Handler<Ping>() {
    @Override
    public void handle(Ping event) {
      System.out.println("Got a Ping!");
    }
  };
}
class Component2 extends ComponentDefinition {
  Positive<PingPongPort> ppp = requires(PingPongPort.class);
  { subscribe(startHandler, control); }
  Handler<Start> startHandler = new handler<Start>() {
    @Override
    public void handle(Start event) {
      trigger(new Pong(), ppp);
    }
  };
}
```

Listing 4.2: A failing program in Kompics Java.

```
3  case object Ping extends KompicsEvent;
4  case object Pong extends KompicsEvent;
5  object PingPongPort extends Port {
6    request(Ping);
7    indication(Pong);
8  }
9  class Component1 extends ComponentDefinition {
10   val ppp = provides(PingPongPort);
11   ppp uponEvent {
12     case Ping => handle {
13       println("Got a Ping!");
14     }
15   }
16 }
17 class Component2 extends ComponentDefinition {
18   val ppp = requires(PingPongPort);
19   ctrl uponEvent {
20     case _: Start => handle {
21       trigger (Pong -> ppp);
22     }
23   }
24 }
```

Listing 4.3: A failing program in Kompics Scala.

Kola[1] extending the Java language, that implements those typing rules. Finally, we will investigate compile-time overheads introduced by Kola and discuss its applicability.

4.1 Common Mistakes in Kompics Java

There are two classes of mistakes, that we must differentiate:

1) There are Kompics component model *violations*, which are always incorrect. We will call those *errors*. Errors result in immediate system termination at runtime in Kompics Java or Scala.

2) And then there are common *issues*, that are often wrong, but sometimes intended and do not violate the Kompics component model. Those we will refer to as *warnings*, based on how a compiler would treat them. Warnings do not cause either Kompics Java or Scala to throw an exception, but may lead to unintended behaviours exhibited at runtime.

[1] for **Ko**mpics **la**nguage

```
3  public class PortP extends PortType {{
4    indication(EventA.class);
5    request(EventB.class);
6  }}
7  public class ComponentC extends ComponentDefinition {
8    Positive<PortP> pp = requires(PortP.class);
9    Handler bHandler = new Handler<EventB>(){
10     public void handle(EventB event) {
11       trigger(new EventA(), pp); // error #3
12     }
13   };
14   { subscribe(bHandler, pp); }// error #4
15 }
16 public class ComponentD extends ComponentDefinition {
17   Handler<Start> startHandler = new Handler<Start>(){
18       public void handle(Start event) {
19           // Do something
20       }
21   }; // warning #1
22 }
23 public class ParentComponent extends ComponentDefinition {{
24   Component cc = create(ComponentC.class, Init.NONE);
25   Component cd = create(ComponentD.class, Init.NONE);
26   connect(cc.provided(PortP.class), // error #2
27           cd.required(PortP.class)); // error #1
28   Component cc2 = create(ComponentC.class, Init.NONE); // warning #2
29 }}
```

Listing 4.4: Pedagogical example with common Kompics mistakes the Java compiler cannot detect.

Listing 4.4 shows a rather construed example of all the mistakes described in the next two paragraphs, all of which could be prevented using information available at compile time.

Errors

1) *Connecting a component on a port it does not require or provide.*
   Clearly, if a component c does not declare a port p of type `P`, then c cannot be connected on a port of type `P` via a channel.
   An example of this is seen in listing 4.4 on line 27, where `cd` of type `ComponentD` gets connected on a port of type `PortP`, but it can clearly be seen in lines 16-22, that `ComponentD` does not declare any ports, certainly not one of type `PortP`.

2) *Connecting the ports of two components the wrong way around.*
   Even if a component c *does* declare a port $p : \mathtt{P}^D$ of type `P` in position D, a connection may be attempted using p in the opposite position $\overline{D}$.
   In listing 4.4 on line 26, for example, a connection attempt to a *provided* instance

of `PortP` is made on component `cc` of type `ComponentC`, but, as seen in line 8, `ComponentC` *requires* a port of type `PortP` and does not provide it.

3) *Triggering events on ports which do not carry them (in that direction).*
We have already seen an example of attempting to trigger an event $e : \mathtt{E}$ on a port $p : \mathtt{P}^D$ of type `P` and position D, where it was *not* the case that $\exists_{E \in out(\mathtt{P}^D)} \mathtt{E} <: E$, in the previous section in listings 4.2 and 4.3.
Another example of this can be seen in listing 4.4 on line 11, where we attempt to trigger an event of type `EventA` on port `pp`, which is in required position and of type `PortP` (line 8). But, as seen on line 4, `EventA` is an *indication* on `PortP`, but only *requests* are *outgoing* on a *required* instance of `PortP`.

4) *Subscribing an event handler to a port for an event that is not carried on that port (in that direction).*
Also previously alluded to, it is illegal to subscribe a handler $h : \mathcal{H}\langle \_ <: \mathtt{E} \rangle$ to a port $p : \mathtt{P}^D$ of type `P` and position D, if it is *not* the case that $\exists_{E \in in(\mathtt{P}^D)} \mathtt{E} <: E$. In general, a handler $\mathcal{H}\langle P \rangle$ pattern P must be verifiable for some member of $in(\mathtt{P}^D)$ to be legal, but this is a much more difficult problem to decide without a concrete description of possible patterns, than the subtyping relationship.
Listing 4.4 shows an example of this on line 14, where `bHandler` is subscribed to port `pp`, which is of type `PortP`. But `bHandler` handles events of type `EventB`, which are *requests* on `PortP`, yet `pp` is a *required* port, which has only *indications* as *incoming*.

Warnings

1) *Creating an event handler without subscribing it to a port.*
As already described at the beginning of chapter 3, forgetting to subscribe an event handler is one of the most common bugs in Kompics Java code, both by novices and experienced Kompics programmers. For example, in listing 4.4 the `startHandler` (lines 17-21) of `ComponentD` is never subscribed to anything, clearly making it "dead code". While this kind of mistake is commonly avoided in Kompics Scala, due to is subscribe-by-default `uponEvent`-syntax, it is not prevented altogether, as unsubscribed handlers can still be created via the `handler`-syntax, as described in section 3.1.
The reason is, of course, that, while in the vast majority of cases an unsubscribed handler constitutes a bug, in certain cases it is absolutely necessary to be able to declare handlers without immediately subscribing them. For example, in a Kompics component that implements a finite state machine (FSM), with different behaviours in different states, it may be perfectly reasonable to dynamically subscribe and unsubscribe handlers at runtime. While this means that *some* subscribe statement for the handler in question must exist, it may not easy to track whether or not this is the case, as handlers for each state may be grouped in a collection and subscription happens over all members of the collection upon a state change.

Thus, truly verifying whether or not a handler is ever subscribed, can not be done with certainty, and at best a "dead code"-like warning could be issued to alert the developer to a potential problem.

2) *Creating components without connecting all their required ports.*
While this is not the most common issue in practice, due to typically low distance in the source code between component creation and port connections, it can happen on occasion, particularly in large components with many children. Like warning 1 this bug usually manifests as silently lost events, and can be difficult to track down.
This happens in listing 4.4 in line 28, as `cc2` is lacking any connections, in particular one for its required port `pp` of type `PortP`.
The same issue could be raised for *provided* ports as well, but typically components function correctly without any connections to their provided ports, but fail if required services are missing. While not connecting any provided ports is probably a bug, or at the very least "dead code", leaving some of them unconnected can happen occasionally in components where some provided services can be considered *optional*. This is typically not true for *required* ports, where every declared instance is typically needed for correct functionality.
However, once again we can not make this judgement for all possible codebases, as, indeed, it can happen that required ports are connected late and via indirections, such as collections, similar to the FSM-case for handlers.

4.2 Typing Rules

In order to describe the rules we want to enforce statically, we are going to provide typing rules for a subset of Kompics kernel language provided in [8, appendix A]. We are also going to base our notation heavily on Pierce's formalisation of *Featherweight Java* [86, 53].

Notation In this section, we will use the following notational conventions: Metavariables `T` range over *types* of the host language (for example Java) and may refer to *primitives* or *classes*, for example. They are considered opaque for the purposes of this treatment. If we need to specify that something is *not* a primitive in the host language, we use the metavariable `C` instead. Metavariables $\mathtt{C}$ range over *component definition* names and `P` over *port names*, and for *event names* `E` is used. D ranges over *event direction* declarations, while P ranges over *port position* declarations, and H over *handler* declarations. K is a *constructor* like in [86], but without supertype fields, in the case of component definitions. S are *statements* in either Kompics or the host language. We assume that all statements type to `Unit`. Note that this type system is *nominal*, i.e. types are compared based on their name, not their structure, with the exception of *handler references* and *port positions*, which are indeed *structural*.
Γ is a *typing context*, mapping variables to types, written as a sequence of mappings

$x_1 : \mathtt{T}_1, \ldots, x_n : \mathtt{T}_n$, for which we will also use the shorthand $\overline{x} : \overline{\mathtt{T}}$. A typing statement for a term has the form $\Gamma \vdash \mathtt{t} : \mathtt{T}$, which is read as "in the environment Γ, term $\mathtt{t}$ has the type $\mathtt{T}$". In order to be able to look up declarations from context, we will have mappings in the environment, which we assume do not change during typing, and are thus independent of the typing context. All port definitions are stored in a *port table* PT, and all component definitions are stored in a *component definition table* CDT. Similarly, event definitions are stored in an *event table* ET. The name set of these tables can be accessed via the dom-function, e.g. dom(ET) is the set of all event names.
We will judge declarations to be well formed, by marking them as OK, or OK in X to note that they are well formed within the body of X, for example a component definition $\mathcal{C}$.
The function $\mathit{fields}(\mathtt{C}|\mathtt{E}|\mathcal{C})$ is defined as in [86, 53] to give a sequence of type-fieldname-pairs (including superclass fields) for a class $\mathtt{C}$, but is also extended to work in the same manner for components and events. In a similar manner, $\mathit{handlers}(\mathcal{C})$ gives a set of all handler definitions in a component $\mathcal{C}$, and $\mathit{ports}(\mathcal{C})$ gives a set of all port position declarations in $\mathcal{C}$.

4.2.1 Strict Typing

We will first describe a simpler strict variant of the typing rules for Kompics, which does not consider event subtyping, and then add event subtyping in the next section. We give this separate variant, because we are going to use it later in chapter 8 to embed Kompics into a language that has a slightly different approach to subtyping than Java: The Rust language.

In figure 4.1 we can see that singleton events are always well formed, while other events are well formed if all their fields are either host types or other events. Indication and requests are well formed if they occur within port types and are over events. Port position declarations are well formed within component definitions if they are over ports. Handler definitions are well formed if they occur within component definitions, and are over an event type. Furthmore, all the statements in their body must type correctly given the event and component definition context. Similarly a component definition is well formed, if it has a constructor with arguments for all its fields, the statements within the constructor are typable, and all their port positions, handlers, and methods are well formed.

The typing rules for statements and expressions are provided in figure 4.2. The type of a handler is given using the same notation as in section 3.2.2 $\mathcal{H}\langle \_ : \mathtt{E} \rangle$ as long as $\mathtt{E}$ an event type and the handler occurs within a component definition. Similarly, for port positions occurring within component definitions, a port $\mathtt{P}$ reference declared as *provided* has type $\mathtt{P}^+$, while a *required* reference has type $\mathtt{P}^-$, as before. We also reuse our *in*, *out* functions in the typing rules for subscription and trigger.
As described in the previous section on error 4 a subscription is typable if its handler matches an event type $\mathtt{E}$ and that type is actually *incoming* on the port reference

$$\frac{}{\texttt{event E}\ OK}\ \text{T-SingletonEvent}$$

$$\frac{\tau_i = \texttt{T} \quad \text{or} \quad \tau_i = \texttt{E} \in \text{dom(ET)} \quad \text{for all } \tau_i \in \overline{\tau}}{\texttt{event E\{}\overline{\tau}\ \overline{x}\texttt{\}}\ OK}\ \text{T-Event}$$

$$\frac{\texttt{E} \in \text{dom(ET)} \qquad \texttt{P} \in \text{dom(PT)}}{\texttt{request E}\ OK\ \text{in P}}\ \text{T-Request}$$

$$\frac{\texttt{E} \in \text{dom(ET)} \qquad \texttt{P} \in \text{dom(PT)}}{\texttt{indication E}\ OK\ \text{in P}}\ \text{T-Indication} \qquad \frac{\overline{D}\ OK\ \text{in P}}{\texttt{port P\{}\overline{D}\texttt{\}}\ OK}\ \text{T-Port}$$

$$\frac{\texttt{P} \in \text{dom(PT)} \qquad \mathcal{C} \in \text{dom(CDT)}}{\texttt{requires P}\ p\ OK\ \text{in}\ \mathcal{C}}\ \text{T-Requires}$$

$$\frac{\texttt{P} \in \text{dom(PT)} \qquad \mathcal{C} \in \text{dom(CDT)}}{\texttt{provides P}\ p\ OK\ \text{in}\ \mathcal{C}}\ \text{T-Provides}$$

$$\frac{\texttt{E} \in \text{dom(ET)} \qquad \mathcal{C} \in \text{dom(CDT)} \qquad e\texttt{:E,this:}\mathcal{C} \vdash \overline{S}\ \texttt{: Unit}}{\texttt{handler}\ h\texttt{(E}\ e\texttt{)\ \{}\ \overline{S}\ \texttt{\}}\ \ OK\ \text{in}\ \mathcal{C}}\ \text{T-Handler}$$

$$\frac{\begin{array}{l} K = \texttt{constructor (}\overline{C}\ \overline{f}\texttt{)\ \{ this.}\overline{f}\ \texttt{= }\overline{f}\texttt{;}\overline{S}\texttt{\}} \\ \overline{f}\texttt{:}\overline{C}\texttt{,this:}\mathcal{C} \vdash \overline{S}\ \texttt{: Unit} \qquad \overline{P}, \overline{H}, \overline{M}\ OK\ \text{in}\ \mathcal{C} \end{array}}{\texttt{component}\ \mathcal{C}\texttt{\{}\overline{P}\texttt{;}\overline{T}\ \overline{f}\texttt{;}K\ \overline{H}\ \overline{M}\texttt{\}}\ OK}\ \text{T-ComponentDefinition}$$

Figure 4.1: Strict typing rules for the Kompics kernel language.

being subscribed to (we use $\texttt{P}^D$ to abstract over port position). Similarly, triggers are typable if their event is of type E, which is actually *outgoing* on the port being triggered on, as described previously in error 3.
New events and component can be created by passing appropriate arguments (allowing subtypes) to their respective constructors.
Finally, connects can only be typed if both ports are of the same type P but in opposite positions. This covers errors 1 and 2, as the port terms can not be typed if no port reference exists on a component. Note that we arbitrarily decided that the provided port comes first, like we did in Kompics Scala, but in this case both options would actually be legal, so an equivalent statement in the opposite direction may be assumed if needed. Further note, that these typing rules assume that the

$$\frac{\Gamma \vdash \mathtt{t} : \mathcal{C} \qquad \mathit{handlers}(\mathcal{C}) = \mathtt{handler}\ \overline{\mathtt{h}}(\overline{\mathtt{E}}\ \overline{e})\{\ \ldots\ \}}{\Gamma \vdash \mathtt{t}.h_i : \mathcal{H}\langle \_ : \mathtt{E_i} \rangle}\ \text{T-HandlerRef}$$

$$\frac{\Gamma \vdash \mathtt{t} : \mathcal{C} \qquad \mathit{ports}(\mathcal{C}) = \{\ldots, \mathtt{provides}\ \mathtt{P_i}\ p_i\,;, \ldots,\}}{\Gamma \vdash \mathtt{t}.p_i : \mathtt{P_i^+}}\ \text{T-ProvidedRef}$$

$$\frac{\Gamma \vdash \mathtt{t} : \mathcal{C} \qquad \mathit{ports}(\mathcal{C}) = \{\ldots, \mathtt{requires}\ \mathtt{P_i}\ p_i\,;, \ldots\}}{\Gamma \vdash \mathtt{t}.p_i : \mathtt{P_i^-}}\ \text{T-RequiredRef}$$

$$\frac{\Gamma \vdash \mathtt{t}_h : \mathcal{H}\langle \_ : \mathtt{E} \rangle \qquad \Gamma \vdash \mathtt{t}_p : \mathtt{P}^D \qquad \mathtt{E} \in \mathit{in}(\mathtt{P}^D)}{\Gamma \vdash \mathtt{subscribe(t}_h\mathtt{,t}_p\mathtt{)} : \mathtt{Unit}}\ \text{T-Subscribe}$$

$$\frac{\mathtt{E} \in \mathrm{dom}(\mathrm{ET}) \qquad \mathit{fields}(\mathtt{E}) = \overline{\tau}\ \overline{f} \qquad \Gamma \vdash \overline{\mathtt{t}} : \overline{\tau'} \qquad \overline{\tau'} <: \overline{\tau}}{\Gamma \vdash \mathtt{E(}\overline{\mathtt{t}}\mathtt{)} : \mathtt{E}}\ \text{T-NewEvent}$$

$$\frac{\Gamma \vdash \mathtt{t}_e : \mathtt{E} \qquad \Gamma \vdash \mathtt{t}_p : \mathtt{P}^D \qquad \mathtt{E} \in \mathit{out}(\mathtt{P}^D)}{\Gamma \vdash \mathtt{trigger\ t}_e\ \mathtt{on\ t}_p : \mathtt{Unit}}\ \text{T-Trigger}$$

$$\frac{\mathcal{C} \in \mathrm{dom}(\mathrm{CDT}) \qquad \mathit{fields}(\mathcal{C}) = \overline{\mathtt{T}}\ \overline{f} \qquad \Gamma \vdash \overline{\mathtt{t}} : \overline{\mathtt{T}'} \qquad \overline{\mathtt{T}'} <: \overline{\mathtt{T}}}{\Gamma \vdash \mathtt{create}\ \mathcal{C}\mathtt{(}\overline{\mathtt{t}}\mathtt{)} : \mathcal{C}}\ \text{T-Create}$$

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{P}^+ \qquad \Gamma \vdash \mathtt{t}_1 : \mathtt{P}^-}{\Gamma \vdash \mathtt{connect\ t}_0\ \mathtt{to\ t}_1 : \mathtt{Unit}}\ \text{T-Connect}$$

Figure 4.2: Strict typing rules for Kompics statements and expressions.

connect statement actually has access to the port instances on the components it is trying to connect (or rather, the rule does not specify how to access them, but field access from [86] is the only described syntax available). This is of course not true in either Kompics Java or Kompics Scala, where component instances are different objects than component definitions, and fields can not be directly accessed from one to the other for (thread) safety reasons. However, this treatment makes the typing rules easier to describe, and the rule itself would not change in the case of Kompics Java/Scala, just more syntax and additional rules for accessing port instances would be required, which we elide for brevity.

$$\frac{\begin{array}{c}\mathtt{E}' \in \mathrm{dom}(\mathrm{ET}) \\ \mathit{fields}(\mathtt{E}') = \overline{\tau'}\ \overline{x'} \qquad K = \mathtt{E}(\overline{\tau'}\ \overline{x'}, \overline{\tau}\ \overline{x})\{\mathtt{super(}\overline{x'}\mathtt{);\ this.}\overline{x} = \overline{x}\mathtt{;}\} \\ \tau_i = \mathtt{T} \text{ or } \tau_i = \mathtt{E} \in \mathrm{dom}(\mathrm{ET}) \text{ for all } \tau_i \in \overline{\tau} \\ \tau'_j = \mathtt{T} \text{ or } \tau'_j = \mathtt{E} \in \mathrm{dom}(\mathrm{ET}) \text{ for all } \tau'_j \in \overline{\tau'}\end{array}}{\mathtt{event\ E\ extends\ E'}\ \{\overline{\tau}\ \overline{x}; K\}\ \mathit{OK}}\ \text{T-Event}$$

$$\frac{\mathrm{ET}(\mathtt{E}) = \mathtt{event\ E\ extends\ E'\ \{\ ...\ \ \}}}{\mathtt{E} <: \mathtt{E}'}\ \text{T-SubEvent}$$

$$\frac{\Gamma \vdash \mathtt{t} : \mathcal{C} \qquad \mathit{handlers}(\mathcal{C}) = \mathtt{handler}\ \overline{\mathtt{h}}(\overline{\mathtt{E}}\ \overline{e})\mathtt{\{\ ...\ \ \}}}{\Gamma \vdash \mathtt{t}.h_i : \mathcal{H}\langle \_ <: \mathtt{E_i} \rangle}\ \text{T-HandlerRef}$$

$$\frac{\Gamma \vdash \mathtt{t}_h : \mathcal{H}\langle \_ <: \mathtt{E} \rangle \qquad \Gamma \vdash \mathtt{t}_p : \mathtt{P}^D \qquad \mathtt{E}' \in \mathit{in}(\mathtt{P}^D) \qquad \mathtt{E} <: \mathtt{E}'}{\Gamma \vdash \mathtt{subscribe(t}_h\mathtt{,t}_p\mathtt{)} : \mathtt{Unit}}\ \text{T-Subscribe}$$

$$\frac{\Gamma \vdash \mathtt{t}_e : \mathtt{E} \qquad \Gamma \vdash \mathtt{t}_p : \mathtt{P}^D \qquad \mathtt{E}' \in \mathit{out}(\mathtt{P}^D) \qquad \mathtt{E} <: \mathtt{E}'}{\Gamma \vdash \mathtt{trigger\ t}_e\ \mathtt{on\ t}_p : \mathtt{Unit}}\ \text{T-Trigger}$$

Figure 4.3: Typing rules for Kompics with event subtyping.

4.2.2 Subtyping

In order to type Kompics Java we must alter the rules from the previous section to allow extending events and allowing subtypes in subscriptions and triggers.

Figure 4.3 shows all the rules that were changed from the previous section. Events may now extend other events, and we use the same notation for "superevent" constructors as in [86] for superclasses. Extending an event introduces a subtyping relationship, as it does for classes.

Handlers have the same requirements as before, but they are typed to $\mathcal{H}\langle \_ <: \mathtt{E} \rangle$ instead of $\mathcal{H}\langle \_ : \mathtt{E} \rangle$ now.

Finally, both subscribe and trigger rules allow an additional event type $\mathtt{E}'$ to be *incoming* or *outgoing*, respectively, as long as the actual event type observed is a subtype of $\mathtt{E}'$.

4.3 Typed Kompics

Given the typing rules established in the previous section, we must now consider how to design and implement a typed Kompics DSL. In general, DSLs are separated

into those that are *embedded* in a host language, and *external* ones, that have their own compiler (or interpreter). This fundamental implementation choice, of course, also affects the ability and method to express typing rules.

4.3.1 Embedded DSL

In an eDSL we can take advantage of existing host language features, constructs, and types, as well as tooling including its compiler with its existing type checker. On the other hand, we are limited to the host language's syntax and type constraints, that can be expressed in the host language's type system. We must also reformulate our typing rules such that they can expressed in the host language's type system. This approach is well suited to host languages that have powerful expressive type systems, such as Scala (or Dotty [4]), for example. If the host language is predetermined due to other constraints, we may have to make some compromises in the DSL, in order to be able to embed it.

We will show an example of how to embed a slightly more limited version of the Kompics model in the Rust language [79, 102] in chapter 8. In this version of the Kompics model, each port can only have a single type for each direction, that is, one request and one indication type. It uses the strict typing rules from section 4.2.1, with that limitation applied.

4.3.2 External DSL

In an external DSL we have the freedom to choose any syntax and implement any typing rules we like, but the onus of developing tools like a parser, a type checker, and a backend that compiles to something executable lies on us. This effort may be limited for small DSLs, with a very narrow focus, but for larger languages, or language that require many additional features to be practically useful, this can be a major undertaking.

Perhaps a compromise, then, is to *extend* an existing language, instead of starting over from scratch. If the new language constructs can, after type checking, be compiled to something that can actually be expressed in the extended language, then the compiler can be limited to an source-to-source (S2S) compiler from the DSL to the target language.

This is the approach taken by *Kola*, the *Kompics language* implementation, based on the typing rules from section 4.2.2. Kola, which is described in detail in the rest of this chapter, is an extension of the Java language, which adds Kompics constructs like *events*, *component definitions*, and *handlers*, among others.

4.4 The Kola DSL

Kola is an extension of the Java language with Kompics constructs. The now familiar example of the *Read-One Write-All Atomic Register* algorithm (algorithm 2) can be seen in Kola in listing 4.5. It's compiler `kolac` performs type checking of

```
package registers;

port OneNRegularRegister {
  public static event Write(Object v)
  public static event Read
  public static event WriteReturn
  public static event ReadReturn(Object v)

  request {Write, Read}
  indication {WriteReturn, ReadReturn}
}

componentdef ReadOneWriteAll {

  provides OneNRegularRegister onrr;

  requires BestEffortBroadcast beb;
  requires PerfectPointToPointLink pl;
  requires PerfectFailureDetector pdf;

  private List<Address> peers;
  private HashSet<Address> correct;
  private Object val = null;
  private HashSet<Address> writeset;

  init(List<Address> peers) {
    this.peers = peers;
    this.correct = new HashSet<Address>(peers);
    this.writeset = new HashSet<Address>();
  }

  handle crashHandler => pfd : PerfectFailureDetector.Crash crash {
    correct.remove(crash.p);
    checkCondition();
  }

  handle readHandler => onrr : OneNRegularRegister.Read read {
    !trigger new OneNRegularRegister.ReadReturn(val) => onrr;
  }

  handle writeHandler => onrr : OneNRegularRegister.Write write {
    !trigger new BestEffortBroadcast.Broadcast(new WriteBEBDeliver(write.v)) => beb;
  }

  handle bebDeliverHandler => beb : WriteBEBDeliver deliver {
    this.val = deliver.v;
    !trigger new PerfectPointToPointLink.Send(deliver.from, Ack.event) => pl;
  }

  handle plDeliverHandler => pl : Ack ack {
    this.writeset.add(ack.from);
    checkCondition();
  }

  private void checkCondition() {
    if (writeset.containsAll(correct)) {
      writeset.clear();
      !trigger OneNRegularRegister.WriteReturn.event => onrr;
    }
  }

  private static event WriteBEBDeliver(Object v) extends BestEffortBroadcast.Deliver

  private static event Ack extends PerfectPointToPointLink.Deliver
}
```

Listing 4.5: Read-One Write-All in Kola.

the new constructs according to the rules laid out in section 4.2.2, in addition to partially type checking Java constructs as necessary, before generating Java source code that is then passed to the Java compiler `javac` for proper Java type checking and generation of JVM bytecode.

In this section we will describe the new grammatical constructs in Kola, as well as the Java source they are compiled to by `kolac`. In the following examples, Java compilation targets are pointed to by a ⇓ from Kola sources. A formal grammar for the new constructs in Kola can be found in appendix A, while the full grammar of Kola can be found in SableCC [37] format at `https://github.com/kompics/kola/blob/master/src/main/sablecc/kola-0.1.7.sablecc`.

4.4.1 Keywords and Tokens

Kola introduces the following new keywords and tokens:

```
handler, handle, port, component, componentdef, !subscribe, !unsubscribe,
!connect, !disconnect, init, !trigger, requires, provides, indication,
request, event, =>
```

The keywords that correspond to Kompics statements, i.e. `!subscribe`, `!unsubscribe`, `!connect`, `!disconnect`, and `!trigger`, have been prefixed with an exclamation mark, in order to avoid confusing the parser when parsing the equivalent method names from the Kompics framework. This was done to maintain backwards-compatibility with existing Kompics Java code.
For the same reason, a few special rules have been introduced to reinterpret keywords as identifiers in certain contexts, such as method invocations and field references. One such instance, for example, is access to the singleton instance of an event via the `MyEvent.event` field, as seen in listing 4.5 on line 46.

The choice of `!` as a prefix was meant to evoke Erlang's and Akka's message sending "tell" operator, but it is also reminiscent of Rust's *function-like macros* (or macros defined via `macro_rules!`).

4.4.2 Events

Kola provides a new syntax for declaring *events*, that can be seen in listing 4.6 and is heavily inspired by Scala's *case classes* and *case objects* [84, chapter 5]. The syntax is meant as a way to reduce Java's verbosity and the boilerplate code required to write these common, immutable data classes. Unlike Scala, however, the choice between a "class-style" event and an "object-style" event is made automatically, based on provided fields (or lack thereof). That is, if an argument list is provided after the event's name, a "class-style" *event*, like `Write` in listing 4.6, is generated, and otherwise a "object-style" *singleton event* like `Read` is generated. Also unlike Scala, a singleton event can not be accessed just by its name, but the single instance is instead accessed via the `Read.event` field, similar to how `Read.class` would work in Java.

```
4   public static event Write(Object v)
5   public static event Read
```

(a) Kola

⇓

```
1  public static class Write implements KompicsEvent {
2    public final Object v;
3    public Write(Object v) {
4      this.v = v;
5    }
6  }
7  public static class Read implements KompicsEvent {
8    public static final Read event = new Read();
9    private Read(){}
10 }
```

(b) Java

Listing 4.6: Event declarations in Kola.

The generated Java code contains automatically generated `public final` fields for each of the event arguments provided, or a single `public static final <Type> event` field for a singleton event. The `KompicsEvent` interface is also automatically implemented for all events.

Kola does not provide a *mutable* annotation for event fields, and thus only immutable events are generated. Due to implicit parallelization and the ability for some messages to cross a network in Kompics, mutable messages are strongly discouraged, as they can lead to unexpected behaviours. If a mutable field is truly needed, events can either be generated the traditional way in pure Java, or alternatively the mutability can be hidden behind a container type, similar to an `Arc<Mutex<T>>` in Rust [102], for example.

4.4.3 Port Types

A port type in Kompics essentially describes which event belongs to which directional group. The Kola syntax for these assignments allows grouping and is borrowed from Java's array initialisers [41, section 10.6]. In the example in listing 4.7 it can be seen, that the `.class` suffix, necessary in normal Java port declarations, is omitted in Kola, as is the instance initialisation block.

More important than the new slightly improved convenience of the new syntax, is the addition of explicit abstract syntax tree (AST) nodes for port declarations, which makes it easier for Kola's compiler to recognise them. This is crucial during the type checking state, as all the typing rules that relate to the errors described in section 4.1 involve port types, as described in section 4.2.

```
port OneNRegularRegister {
  request { Write, Read }
  indication { WriteReturn, ReadReturn }
}
```

(a) Kola

⇓

```
public class OneNRegularRegister {
  {
    indication(WriteReturn.class);
    indication(ReadReturn.class);
    request(Write.class);
    request(Read.class);
  }
}
```

(b) Java

Listing 4.7: A port declaration in Kola.

4.4.4 Component Definitions

Component definitions are the templates for components, in much the same way that classes are templates for objects. They define the *ports*, *handlers*, *child components*, and *internal state* that a component instance of a specific type will have.

The component definition syntax, introduced by the new `componentdef` keyword as seen in listings 4.8, 4.9, 4.10, and 4.11, allows everything that a normal class definition would allow in its body, plus additional Kola declarations of *init blocks*, *ports positions*, *child components*, and *handlers*, as well as setup-statements, such as `!connect` and `!subscribe`, for example.

Init Blocks Since only instances of `Init<C>` may be passed to `create(...)` calls in Kompics Java, passing constructor arguments tends to be extremely verbose, as it always involves the creation of new subclass of `Init<C>` for each component `C`.

Kola avoids this verbosity with a new construct called an *init block*, which acts similar to a Java constructor, but automatically generates the required subclass of `Init<C>`.

As can be seen in listing 4.8, an init block compiles to two constructors and an `AutowireInit` class for the appropriate arity (e.g., `AutowireInit1` for a single argument), extending `Init<C>` for the appropiate type of `C`. The public constructor takes the `AutowireInit` instance passed by the Kompics runtime during instance creation, and passes each parameter as a separate field to the other constructor, which also contains the contents of the Kola init block.

```
componentdef ReadOneWriteAll {
  private List<Address> peers;
  private HashSet<Address> correct;
  init(List<Address> peers) {
    this.peers = peers;
    this.correct = new HashSet<Address>(peers);
  }
}
```

(a) Kola

⇓

```
public class ReadOneWriteAll extends ComponentDefinition {
  private List<Address> peers;
  private HashSet<Address> correct;
  ReadOneWriteAll(List<Address> peers) {
    this.peers = peers;
    this.correct = new HashSet<Address>(peers);
  }
  public ReadOneWriteAll(final AutowireInit1 autowireInit) {
    this(autowireInit.peers);
  }
  public static class AutowireInit1 extends Init<ReadOneWriteAll> {
    public final List<Address> peers;
    public AutowireInit1(List<Address> peers) {
      this.peers = peers;
    }
  }
}
```

(b) Java

Listing 4.8: Component definition with init block in Kola.

Port Positions Most components provide and require a number of named ports of a specific port type. In Java these are simply variables that are initialised once the component is loaded, but in Kola they are treated as a special type of field by the compiler, identified by the `provides` and `requires` keywords, as seen in listing 4.9.

The compilation of these fields is very straightforward, as it essentially avoids the explicit description of the *port position* type (i.e., `Negative<P>` or `Positive<P>`). As in *port type* declarations, the `.class` suffix is also avoided.

As it was the case with *port type* declarations, the read benefit is not so much the boilerplate code saved, but the ability for the compiler to identify AST nodes that correspond to port positions, which is important for type checking later.

```
componentdef ReadOneWriteAll {
  provides OneNRegularRegister onrr;
  requires BestEffortBroadcast beb;
}
```

(a) Kola

⇓

```
public class ReadOneWriteAll extends ComponentDefinition {
  protected final Negative<OneNRegularRegister> onrr =
  ↪ provides(OneNRegularRegister.class);
  protected final Positive<BestEffortBroadcast> beb =
  ↪ requires(BestEffortBroadcast.class);
}
```

(b) Java

Listing 4.9: Port position declarations in Kola.

```
componentdef ExampleParent {
  component StaticExampleChild("Argument") staticChild;
  component DynamicExampleChild dynamicChild;
}
```

(a) Kola

⇓

```
public class Example extends ComponentDefinition {
  protected final Component staticChild = create(
    StaticExampleChild.class,
    new AutowireInit1("Argument"));
  protected Component dynamicChild;
}
```

(b) Java

Listing 4.10: Child component declarations in Kola.

Child Components Components in Kompics form a supervision hierarchy, similar to Erlang actors. A child component can be created *dynamically* at runtime, for example in the body of an event handler, or *statically* at component creation time. Kola only deals with static creation at this time, as shown in listing 4.10, while dynamic creation works as before in the Java version, i.e. using the `create(...)` method of the parent component definition.

Omission of the component initialisation arguments, as in listing 4.10 on line 3, indicates a dynamic child. In this case, only a child component field is created, but the component is not initialised or started automatically, leaving it prepared for dynamic instantiation.
If the argument list is empty, the component is initialised with a no-argument constructor, that is using using `Init.NONE`.
If arguments are provided, the Kola compiler will attempt to find a matching `AutowireInit` class (cf. *Init Blocks* above) for this component and of the appropriate arity, and use that for initialisation. Kola does not currently support the use of static child component initialisation for components declared in Java with custom init events.

Handlers Within a component, handlers are used to process incoming events. Handlers are subscribed to ports, which can be done *statically* or *dynamically*, as was the case with child component creation. However, in this case Kola supports both, as shown in listings 4.11a and 4.11b, respectively.

A `handler` declaration simply declares a special type of field that holds a handler object, that is an implementation of the `Handler<E>` interface from the Kompics framework. Of course, a handler does not actually do anything unless it is subscribed to a matching port, which can be done with the `!subscribe` statement and undone with the `!unsubscribe` statement. Kola allows these statements directly in the component definition body, and will move them into the component's instance initialiser during compilation. Since forgetting to subscribe a handler to a port is one of the most common mistakes in Kompics in practice, as described in section 4.1 as warning 1, Kola also defines a `handle` declaration, which does handler creation and subscription in one step. Like in Kompics Scala (cf. section 3.1), this should be the default approach of creating handlers in Kola. Using the declared field name, the handler can still be unsubscribed later, if necessary.

The astute reader may have already noticed the extra method created in listing 4.11c on lines 13-15, which contains the actual content of the `handle` or `handler` block. The reason that Kola compiles to that form instead of simply doing the work in the body of the handler's `public void handle(Read read) { ... }` method, has to do with accessing the parent's fields via `this`. Kola suggests the idea that handler blocks are like a special kind of function on the body of the component, which implies that any component field `f` can be accessed via `this.f`. However, if we move the body of the Kola handler into an anonymous Java `Handler<E>` instance,`this` refers to *that* instance, while the parent `ComponentC` field must be accessed via

```
componentdef ReadOneWriteAll {
  provides OneNRegularRegister onrr;
  private Object val = null;
  handle readHandler => onrr : Read read {
    !trigger new ReadReturn(val) => onrr;
  }
}
```

(a) Kola implicit subscription.

```
componentdef ReadOneWriteAll {
  provides OneNRegularRegister onrr;
  private Object val = null;
  handler readHandler : Read read {
    !trigger new ReadReturn(val) => onrr;
  }
  !subscribe readHander => onrr;
}
```

(b) Kola explicit subscription.

⇓

```
public class ReadOneWriteAll extends ComponentDefinition {
  protected final Negative<OneNRegularRegister> onrr =
  ↪  provides(OneNRegularRegister.class);
  private Object val = null;
  protected final Handler<Read> readHandler = new Handler<Read>() {
    @Override
    public void handle(Read read) {
      ReadOneWriteAll.this.readHandlerMethod(read);
    }
  };
  {
    subscribe(readHandler, onrr);
  }
  private final void readHandlerMethod(Read read) {
    trigger(new ReadReturn(val), onrr);
  }
}
```

(c) Java

Listing 4.11: Handler declarations in Kola.

`ComponentC.this.f`. As this would be very unintuitive in Kola, the code is compiled to a private method on `ComponentC` directly, instead, and simply invoked using the fully specified syntax from within the handler's body.

Channels are created using the `!connect providedPort => requiredPort` statement, and destroyed using the `!disconnect => providedPort => requiredPort` statement. Like in Kompic Scala, both statements take the *provided* port to the left of the arrow and the *required* port to the right.

Generally, this is compiled in the obvious manner by simply replacing the keyword and arrow with the equivalent binary method from the Kompics framework, i.e. either `connect(providedPort, requiredPort)` or `disconnect(providedPort, requiredPort)`. However, if the type of either of the expressions is not a port position type, but `Component`, then, instead of using the expression directly, a `.getPositive(PortType.class)` or `.getNegative(PortType.class)` is suffixed to the expression. While the `PortType` can technically be inferred in the case of at least one of the expressions being of port position type, Kola requires an explicit annotation in all cases for clarity.
In any case, Kola's type checker will verify that the connection is sound, before generating any Java code.

4.4.5 Trigger

Kola also adds a `!trigger someEvent => somePort;` statement, mostly for consistency with the other new statements, but also to point out to the compiler that it should verify the T-Trigger rule from section 4.2.2.

4.4.6 Other

As another small extension to Java, Kola allows multiple top-level type declarations in a single file, similar to Scala [84, chapter 9]. This makes it possible to easily keep a component or a port type together with all its associated events for easy reference, without having to resort to static inner classes.

Additionally, Kola keeps all classes in `se.sics.kompics` permanently in scope, just like Java does with `java.lang`. This allows developers to omit the repetitive Kompics imports in the header of Kola files. The necessary imports for the Java source files are automatically generated by the compiler.

4.5 Kola Implementation

The Kola compiler, `kolac`, is a S2S compiler — also called a *transpiler* — that takes Kola source files, which end in `.kola`, and produces Java source files.

In order to make the inclusion Kola files in existing Java projects more convenient, there is also a Apache Maven plugin, that allows Kola projects to be integrated into the Java development cycle.

4.5.1 The Kola Compiler

`kolac` consists of three major parts: A lexer and parser frontend, an abstract syntax tree (AST) analyser/transformer component, and finally a Java source generator and writer.

Lexing and Parsing Both the Lexer and the Parser are automatically generated by the SableCC parser generator [37] from a grammar file in SableCC's format, which can be found in its entirety at `http://github.com/kompics/kola/blob/master/src/main/sablecc/kola-0.1.7.sablecc`. This grammar is based on an older Java 1.7 grammar provided on the SableCC website at `http://www.sablecc.org/java1.7/`. It was extended with the Kola specific rules described in appendix A. Additionally, transformation rules from concrete syntax tree (CST) to abstract syntax tree (AST) were added, as they are supported in newer SableCC versions.

Lexing and parsing is done on multiple source files in the *source path* in parallel using a `java.util.concurrent.ExecutorService` with a thread pool, since this stage is trivially parallelisable. If no errors occur during lexing and parsing, the AST analysis stage is invoked for all source files in sequence to avoid concurrency issues with type resolution.

AST Analysis and Transformation A single-pass depth-first traversal is used to analyse the AST, and transform it into a Java syntax tree (JST). The format used for the JST is a modified version of Sun's JCodeModel, which can be found at `http://github.com/Bathtor/JCodeModel`.
Where names cannot be immediately resolved, markers are inserted into the JST, to be resolved later after the AST→JST pass has finished, and all source files have been analysed. During this pass the new Kola grammar structures are resolved into Java structures, as described in section 4.4. Additionally, the typing rules from section 4.2, and in particular the subtyping variant from section 4.2.2, are checked. Additionally, the analysis stage tries to figure out whether all handlers are subscribed to ports, and if all components have their required ports connected. This behaviour attempts to implement the semantics of warnings 1 and 2 from section 4.1 as best as possible, but as already described there, perfect accuracy is typically not possible. Thus, even if violations of this type are found, the only consequence is a warning issued by the compiler, while the compilation itself proceeds as expected.

Java Source Generation This stage simply uses JCodeModel's source generation facilities and a file writer, to generate the right directory structure and Java source files for the JST in the *output directory*. During this stage it is also attempted to resolve previously unresolved names in a lazy manner, and if necessary errors are thrown where this is still not possible, as name resolution is required for correct `import`-statement generation.

Algorithm	Java	Scala	Kola
Basic Broadcast	66 LOC	28 LOC	31 LOC
ROWA $(1, N)$ Regular Register	121 LOC	68 LOC	65 LOC
RIWCM (N, N) Atomic Register	118 LOC	90 LOC	111 LOC
Increasing Timeout EPFD	109 LOC	72 LOC	67 LOC

Table 4.1: Code maintainability comparison based on LOC between Kompics Java and Kompics Scala and Kola.

4.5.2 Maven Plugin

The Kola plugin for Apache Maven wraps the actual `kolac` executable. It collects the correct `CLASSPATH` elements, and then replaces the normal Maven `sourceDirectory`(s) with the `outputDirectory` from `kolac`. Finally, it compiles all the Kola and Java files in the `inputDirectory` with `kolac`. It is run in the `generate-sources` phase of the Maven build lifecycle[2], which causes Maven to run `javac` on the generated Java source files afterwards.

The sources for the Kola Maven plugin can be found at `https://github.com/kompics/kola-maven-plugin`.

4.6 Evaluation

Having provided a new compiler with a type checker that improves the compile-time safety guarantees of Kompics code, as well as a new DSL for Kompics specific constructs in Kola, we must now ask ourselves what we have gained and what we have paid. Clearly, an additional compiler in the toolchain does not come for free in terms of compile time. On the other hand, a new DSL offers the possibility of readability improvements, as we have seen in section 3.3 for Kompics Scala. Finally, we will look at limitations of the approach imposed by the requirement to be compatible with Java.

4.6.1 Code Maintainability

Like in section 3.3, we will use the code maintainability measure lines of code (LOC) as a proxy for code readability. To this end, we extended table 3.1 with a new column for Kola implementations in table 4.1[3]. As we would expect, Kola code is consistently more compact than equivalent Kompics Java code. However, more interesting is that Kola code is sometimes also more compact than Kompics Scala code, despite all of Java's verbosity still being present in Kola outside Kompics specific constructs.

[2]Refer to `https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html` for more details of Maven's build lifecycle.

[3]Sequence Paxos is omitted due to time constraints.

This fact lets us reinterpret some of the numbers from table 3.1, as it gives us a better idea which improvements come from the respective Kompics DSLs, and which are simply cases of Scala being less verbose than Java. For example, in both the regular register algorithm and the failure detector, Kola code is shorter than equivalent Kompics Scala code, because Kompics-specific constructs are shorter in Kola and the algorithms have fairly small handler bodies. Conversely, in particular the atomic register algorithm suffers greatly from Java's verbosity in finding the maximum by a custom comparator in a set, and Kompics Scala is aided greatly by being able to express this in a single line using the `maxBy` collection library method.

To summarise, while Kola can lead to more compact code and improve readability in situations where Kompics constructs are dominant, Kompics Scala leverages Scala's conciseness, which often leads to more compact code overall. Furthermore, Kompics Scala's pattern matching capabilities are missing in Kola, making it a worse fit for situations where formal algorithms must be translated into real code.

4.6.2 Overhead

Two types of overhead are of interest for us: Compile time overhead, introduced by our additional Kola compiler, and run time overhead, which could occur if the Kola compiler produces less efficient code, compared to hand written Kompics code. The experiments in this section were run on a MacBook Pro with a 2.7GHz quad-core Intel Core i7 (L2 Cache 256KB per core, L3 Cache 8MB) and 16GB of DDR3 memory, as well as a solid-state drive (SSD). The JVM used was Oracle Java HotSpot(TM) 64-Bit Server VM 18.9 build `11.0.2+9-LTS` for MacOS.

Compile-time Overhead While it is clear that there has to be *some* compile-time overhead, since a certain amount of additional, and partially redundant, work is being done, we have to investigate exactly how much overhead is introduced, and how it scales with program size.

In order to do so, we have measured the compile time for small, comparable Kompics Java and Kola projects using the Kola Maven plugin, instead of measuring raw compiler time. The values for Kola include both the `kolac` and `javac` execution times, while Kompics Java only measured `javac`. In order to adjust for overhead introduced by using Maven, we additionally compiled a completely empty project with the same setup.

The two programs used were 1) a minimal *Hello World* implementation, and 2) a slightly larger *Throughput Ping Pong* example.The source code for both examples in Kola and in Kompics Java can be found at `https://github.com/kompics/kola-examples`.

Table 4.2 shows the results of both programs, averaged over 25 measurements each. Additionally, it shows compiling both programs together in the same compilation unit, and also compiling all four (Kola and Java) programs together through `kolac` (as `javac` can't handle Kola files, of course).

	code length		raw time		adjusted time		
Project	Java	Kola	Java	Kola	Java	Kola	Factor
Hello World	33 LOC	14 LOC	2.014 s	2.556 s	0.171 s	0.713	4.2×
Ping Pong	124 LOC	62 LOC	2.041 s	2.709 s	0.197 s	0.865 s	4.4×
Both Together	157 LOC	76LOC	2.078 s	2.754 s	0.234 s	0.910 s	3.9×
All Together	–	233LOC	–	2.797 s	–	0.954 s	–

Table 4.2: Average compile time in seconds as reported by Maven and adjusted for a average Maven overhead of 1.844 s.

As can be seen easily in the adjusted columns, the Kola compiler adds about three times Java's compilation effort again. While that does seem expensive, it must be considered that the Kola compiler does similar work to the Java compiler and is certainly a little less optimised. Additionally, the Kola compiler also produces more intermediate output, and calculates a number of software maintainability metrics during the normal compilation process.However, passing through Java files, as is done in the last row, adds very little overhead.

Run-time Overhead As opposed to compile time overhead, it is not clear whether the Kola compiler introduces any noticeable inefficiencies into the generated code. As the only performance critical aspects that are different in Kola are handler definitions, any introduced overhead should be reflected in a loss of raw event handling throughput.

To measure this, we used the *Ping Pong* example to collect event throughput statistics, which we used to compare the performance of the Kompics Java and the Kola implementations. These experiments were run using the Kompics scheduler with a thread pool of size $n = 2$ and an event batch size of $\eta_{\max} = 50$. The JVM used default settings, as memory management was not an issue, since no new objects were being created during the measurement phase. We measured 25 consecutive cold-start runs, with a length of 200 million events each (100 million ping-pong-pairs).

The Java version reached an average of $5.721\,99 \times 10^6$ events per second, with a sample standard deviation of 286 912. The Kola version showed an average of $5.625\,86 \times 10^6$ events per second, with a sample standard deviation of 342 940. Using Student's t-distribution at a sample size of 25+25, the 95% confidence interval for the mean difference between the two samples is (−83 849,276 109), which is statistically insignificant at this level.

The tilt of the difference and the mean difference value of 96 130 suggest that perhaps Kompics Java is a little faster. This might be caused by the introduction of the additional method call in handlers compiled from Kola to Java, as described at the end of section 4.4.4. While it is expected that this call would eventually be eliminated by the just-in-time (JIT) compiler, it may have an effect here, as the experiment runs are cold-start.

4.6.3 Limitations

While Kola does provide an implementation of the typing rules from section 4.2.2, which are designed to address the errors described in section 4.1, its design goal to be compatible with Kompics Java and the Java language, in general, introduces some limitations on the degree to which errors 1-4 can be detected even in Kola. In particular, Java's feature to perform almost arbitrary dynamic casts — indirectly via a common supertype like `Object` — can lead to important type information getting lost.

Consider, for example, a kind of meta-component, such as a framework for implementing FSM-like components. In such a scenario it may very well be necessary for the implementation to add both handlers and ports into a large meta-collection, that maps every FSM-state *S* to the specific handler subscriptions that should be active during *S*. By virtue of this collection having to work over multiple port and handler types, it is impossible to statically type this scenario with a type than can be used to check typing rule T-Subscribe, for example. Since generics are removed at runtime in Java and previously mentioned arbitrary casts are allowed, writing such a meta-component is possible in Kompics Java, but not in Kola. However, it may be necessary to write such a component in large projects that rely heavily on a FSM abstraction to organise their code.

In such a circumstance Kola must interact defensively with the pure Java code used to implement the FSM-component, and may thus not be able to detect an instance of error 4. Similar examples can be constructed for errors 1-3.

It is also important to note that the typing rules that Kola implements only verify that triggers and subscriptions are legal on a port. In no way do they guarantee that every triggered event is actually handled by all or even any receivers. Doing so would be completely opposed to the reason that Kompics allows subtyping in the first place; that is, avoiding to handle certain messages that are not interesting to a particular algorithm, as was explained in the introduction to chapter 3.

4.7 Related Work

While Akka[69] on Scala and Java endeavours to bring the Actor model to a statically typed language, as described in section 2.1.1.2, the types of messages that are being sent to specific actors are usually not checked. However, recently *Akka Typed* [71] has become officially supported, after multiple previous attempts to introduce type-safety in Akka at that level, such as:

1) *Typed Actors* are an implementation of the *Active Objects* pattern on top of Akka, effectively hiding the message-passing model below intercepted method invocations. This approach, however, tends to be very unintuitive and has never reached widespread application among Akka users.

2) *TAkka* [46] supports statically typed messages, and actor behaviours.

Akka Typed decouples *behaviour* specifications and the actors that exhibit such behaviour. This is very similar to the Kompics model, replacing channels and events with addressed messages.

Implementations with a similar design to Akka Typed exist in other languages as well, such as Rust's *Actix* [96] or *Riker* [88] frameworks.

Effpi [90] is a recent approach to statically verify message-passing protocols, based on type-level model checking, provided by the powerful Dotty compiler [4].

The *Go language* [40] is a statically typed language that uses statically typed channels for concurrency, thus limiting which messages can be sent to a particular receiver.

Orleans [101] is an actor implementation in C♯ for the .NET framework. It allows specification of actor behaviours using interfaces, which actors then have to implement, in a manner similar to *Typed Actors* above. Orleans' style, however, is reminiscent of asynchronous RPCs rather than message passing.

CHAPTER

5

Unit Testing Message-Passing Systems

In this chapter, we describe a methodology and associated DSL for unit testing message-passing systems, and show a prototype implementation of it for Kompics Java.

Having spent the last chapters developing approaches to avoid and prevent bugs in Kompics code, we must now recognise that *safe* code is not always the same as *correct* code. Consider the example of the *best-effort broadcast* abstraction, as given in [16, p. 75]. In addition to defining the meaning of the BROADCAST and DELIVER events, the abstraction also specifies three properties that any implementation has to fulfil:

Validity If a correct process broadcasts a message m, then every correct process eventually delivers m.

No Duplication No message is delivered more than once.

No Creation If a process delivers a message m with sender s, then m was previously broadcast by process s.

Consider now listing 5.1 as an implementation of the best-effort broadcast abstraction. Clearly, the subscriptions are legal, and as there are no triggers, they do not violate any typing rules, either. Furthermore, all declared handlers are subscribed, and all declared ports have subscribed handlers. The implementation is perfectly safe, yet does not exhibit the desired behaviour — or any behaviour at all, really. In particular, it only violates the **Validity** property of the abstraction, because that property is a *liveness property*, while the other two are *safety properties* (cf. [16, section 2.1.3]).

Clearly, a mechanism is needed to verify the behaviours of components against some template of expected behaviours. The Kompics framework indeed provides a set of tools for doing so, as part of its *simulation* support [5, 8]. However, the tools provided by the simulation framework are more targeted at repeatable *integration tests*, where an ensemble of Kompics components, using the abstracted network and timer capabilities of the simulator, is being run together to verify some larger properties, such as, for example, linearisability of a key-value store. While the simulation framework can be used for verify single component behaviours as well, doing so is cumbersome in its DSL, and achieving any amount of detail on the execution can require significant additions to otherwise production code.

In this chapter we will explore a mechanism for verifying detailed properties of event sequences produced by a single component, which is the message-passing equivalent to *unit testing* [2], another form of automated software testing. Unit testing focuses on relatively small areas of code such as a single function, a single class in an object oriented language, or a single actor or component in a message-passing model. However, most message-passing models allow for the composition of abstractions in such a way that a single abstraction may hide a large hierarchy of components or actors, thus effectively presenting very large units. In this sense the notion of unit testing for these systems straggles the border between unit and integration testing. For these systems the description of the test for automation is not straight forward, as common unit testing frameworks like JUnit[98], for example, express their tests in an imperative manner, as is appropriate to the languages they

```
componentdef TrivialBroadcast {
  provides BestEffortBroadcast beb;
  requires PerfectPointToPointLink pl;

  handle broadcastHandler => beb : BestEffortBroadcast.Broadcast e {
    // ignore
  }

  handle deliverHandler => pl : PerfectPointToPointLink.Deliver e {
    // ignore
  }
}
```

Listing 5.1: Trivially safe, but incorrect implementation of Best Effort Broadcast.

are written in: Java, in this case. The behaviour of a message-passing component, however, is encoded in the sequence of messages it sends and receives, and also in its internal state. As internal state is never shared in message-passing frameworks, though, it is often desirable to test them in a black box manner, that is without inspecting the internal state, but only by observing the events entering and leaving the component.

One approach to testing message-passing systems is thus to record the sequence of messages the component has seen, and check whether this sequence fulfils certain properties that are required for it to be 'correct'. For example, such a property could be "no message is emitted twice". In this sense a property is a predicate over the message sequence, and a component is 'correct' if all its predicates evaluate to `true` for all sequences it can produce. This is a very formal way of unit testing message-passing systems, which forms the basis for model checking methods in software testing, where the checker generates both sequences and the code to verify the properties over the sequences. This approach is very powerful, but it requires a very good formal description of the problem to be solved by the code. While this description is usually available for well-known algorithms and abstractions, it is often not economical to find for the average programmer trying to get an interactive, practical system to work.

From the perspective of a programmer it is often easier to work with a more execution-oriented description of the desired behaviour, similar to the following: "If I send a message m_1 to the component I expect to see only a message m_2 leaving it, then giving it a message m_3 should cause emission of one or more m_4 messages". This approach represents a compromise between the formal approach of model checking and the imperative approach of traditional unit testing frameworks. The same sequence can be described more concisely, combining the expected message sequences going *in*to the component and those coming *out* of it, by stating that we expect to see the sequence $(in(m_1), out(m_2), in(m_3), out(m_4), (out(m_4))^*)$ (where * is

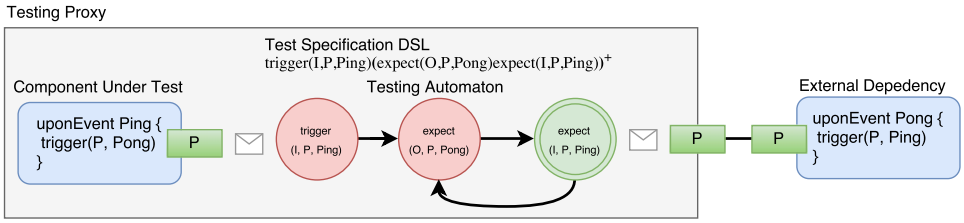


Figure 5.1: Overview of the KompicsTesting approach using the classic Ping-Pong Example.

the Kleene star indicating zero or more occurrences). If we call the combination of a message with its direction an *event*[1] and arbitrarily assign the letters a, b, c, d to the events in the previous sequence, we really want to recognise the sequences described by $abcdd^*$ and reject all other sequences. It can be seen that the set $A = \{a, b, c, d\}$ forms the alphabet of a "language", where sequences of events are the "strings of letters". In this case $abcdd^*$ is in fact a *regular expression* [60], thus describing a *regular language*. Such a language is recognised by a unique deterministic finite automaton (DFA), which can be generated from the expression itself [52].

We will begin by giving a more formal introduction to languages and automata, before describing our language-based approach to unit testing message-passing systems. We will then describe a general DSL to express test cases for execution sequences of messages and states in a manner that extends regular expressions to support message-passing requirements, such that they can be converted into a testing automaton. We show in a general way how to implement these automata, such that they are executed in lockstep with the component under test (CUT), either producing or matching the incoming messages and matching the outgoing messages. If desired by the test designer and supported by the target framework, the component's internal state can also be inspected at predefined points in the execution. Messages can be produced and handled by the CUT, the testing automaton, or other external components. Such a test is considered passed if the automaton ends the execution in an "accept" state. A high-level overview of our approach, as applied to the Kompics model, can be seen in figure 5.1.
After describing the general model, we will give a brief overview of a prototype implementation for Kompics Java.

[1] This definition differs slightly from the one in chapter 2, and will only be used in this chapter, for lack of a better terminology. We will use message to refer both to addressed messages and unaddressed Kompics events, in order to to abstract over the concept.

5.1 Background

As this work combines ideas from the very different fields of software testing, concurrent and distributed systems, and formal languages and automata theory, we will begin by giving a quick introduction for each, in order to clearly establish concepts and notations that will be used throughout the chapter.

5.1.1 Automated Software Testing

Automated software testing can be done on many levels, the most common being unit testing and integration testing [2]. Unit testing focuses on relatively small areas of code such as a single function, a single class in an object oriented language, or a single actor or component in a message-passing model. At this scale the behaviour of the code is well understood, to such detail that every corner case could be covered by a test if that is desired. Integration testing, on the other hand, focuses on larger scale constructs such as multiple modules interacting, or a (sub-)tree of actors in an actor system working together. At this scale complexity is typically so high already that only relatively broad requirements can be tested. Used together unit and integration testing typically complement each other and can increase confidence in the code base as a whole.

When designing tests, it is important to decide whether the logic of the tests assumes knowledge of the concrete implementation of the functionality being tested or not. The former is referred to as *white-box* testing, while the latter is called *black-box* testing [59]. For the purposes of this chapter, in the context of testing actor/component systems, we shall adopt a related, but more specific, definition of these terms: In *white-box* testing the tests have access to a actor's/component's internal state, while in *black-box* testing only the messages entering and leaving a actor/component may be observed.

5.1.2 Executions

Real-world message-passing systems typically follow one of the formal concurrent processing models in their design philosophy, as described in section 2.1. These models are sufficiently similar that our approach applies to all of them with minor variations. What is relevant from these models is that they execute in steps, where in each step one of a set of actions may be taken. For the purposes of this chapter, we assume that the possible actions are:

1) *receive* (recv(e)) a message,

2) *execute* (comp(c)) local computation, and

3) *send* (send(e)) a message.

For the time being we are not concerned with where messages come from, or where they go, only if they are going into the component (recv) or coming out of it (send).

Given this, we call a sequence of such actions, one per step, an *execution* (E) (at the CUT). For a given set of messages (written m_i) and computations (written c_i) an example execution might be $E = (\text{recv}(m_1), \text{comp}(c_1), \text{comp}(c_2), \text{send}(m_2))$. All of these steps can be regarded as *events* (e_i), each of which moves the CUT from one *state* (s_i) to the next. If multiple processes are involved, then all the individual process states together are considered a *configuration* of the system. This leads us to a notation for executions like $E = (e_1, s_1, e_2, s_2, \ldots)$, interleaving events and states. Either of these notations can be convenient in different circumstances and they will be used interchangeably throughout the rest of the chapter.

Most of the formal concurrent processing models also consider executions to be infinite, and fill up the suffix of the execution, where nothing meaningful occurs, with *nil* events that have no influence on the outcome. We shall mostly eschew that notion for this paper and simply end all executions in ellipses (i.e. '…') leaving the finiteness or lack of it undefined.

Many distributed abstractions can be described without referring to the internal state of participations actors/components. In such cases we can write an execution $E = (s_0, e_1, s_1, e_2, s_2, e_3, s_3, e_4, s_4, \ldots)$ simply as $T = (e_1, e_2, e_3, e_4, \ldots)$, omitting internal state, and call it a *trace* instead. For deterministic components, given an initial state s_0, a trace is sufficient to determine all intermediate states.

5.1.3 Formal Languages and Automata

Let Σ be any finite set of symbols, then the *closure* Σ^* denotes the set of all sequences over Σ. That is if $\Sigma = \{a, b\}$ then $\Sigma^* = \{a, b, ab, aab, abb, \ldots\}$, where aab is short for the sequence (a, a, b). We say that Σ is the *alphabet*, its elements are *letters*, and the elements of Σ^* are *words* or *strings*.

Any set $\mathrm{L} \subseteq \Sigma^*$ is a *language* over Σ. In effect L is nothing but an explicit enumeration of all possible words that fulfil certain criteria. Instead of enumerating all the words, one can try to describe the criteria they must fulfil, in order to be part of L. Such a description is called a *grammar* G and is often shorter than explicitly enumerating the words. When a grammar G perfectly describes a language L we say G generates L and write $L(G) = \mathrm{L}$.
More formally, Chomsky [22] defines a grammar as a 4-tuple $(N, \Sigma, P, \mathrm{S})$, where N is a set of *nonterminal* symbols and Σ is the alphabet of *terminal* symbols, such that $N \cap \Sigma = \emptyset$. P is a set of production rules of the form $(\Sigma \cup N)^* N (\Sigma \cup N)^* \to (\Sigma \cup N)^*$, and $\mathrm{S} \in N$ is a unique *start* symbol. Without going into unnecessary details, given this form any string in $L(G)$ can be generated using G by beginning with S and repeatedly expanding the right-hand side of a matching production rule in P until all nonterminals are consumed. If all possible such expansions are executed, all of $L(G)$ is generated.

While a grammar G generates a language $\mathrm{L} = L(G)$, a function $M : \Sigma^* \to \mathbb{B}$, defined

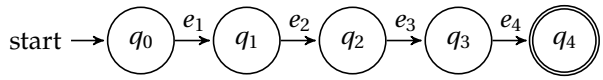


Figure 5.2: DFA recognising the sequence of events e_1 to e_4.

as follows, is called a *recogniser* of L:

$$M(w) = \begin{cases} \text{TRUE} & \text{if } w \in \text{L} \\ \text{FALSE} & \text{otherwise} \end{cases}$$

$\mathbb{B}$ here refers to the boolean set $\mathbb{B} = \{\text{TRUE}, \text{FALSE}\}$ or in this context sometimes also written as $\mathbb{B} = \{\text{ACCEPT}, \text{REJECT}\}$. For any such M, the language it recognises is written as $L(M)$. The description of M is called an *automaton* or an *abstract machine*.

If M can be described as a deterministic finite automaton (DFA), then $L(M)$ is called *regular* and a grammar G with $L(G) = L(M)$ is called a *regular grammar* and may be more concisely described by a *regular expression*, instead of the production rules form given above. There are more general classes of formal grammars than only regular, and their relationship with each other is described by the 4-level *Chomsky hierarchy*. In this model regular grammars are of type 3. Type-2 grammars are called *context-free* (CFG). They are recognised by *pushdown automata*, that is a DFA with access to an unbounded stack. *Context-sensitive* grammars are type-1, and are recognised by a *linear bounded automaton*, a bounded tape version of a *Turing machine*, which in its unbounded form recognises *unrestricted* type-0 grammars.
It should be noted that, given any grammar G, we can always find an automaton M such that $L(G) = L(M)$ [52].

5.2 The Language of Event-Streams

The question we must answer in this section is: How can we arrive at an executable testing automaton, given a programmer's description of what a correct execution (sequence) of a component is? Let us call this description the *specification S*. Recall the example from the introduction to this chapter and, to begin with, simplify it even further:

> *S* = "*If I send a message m_1 to the component, then I expect to see only a message m_2 leaving it. Afterwards, giving it a message m_3 should cause emission of one m_4 message.*"

In terms of an expected trace $T = (e_1, e_2, e_3, e_4, \ldots)$, this would mean $e_1 = \text{recv}(m_1), e_2 = \text{send}(m_2), e_3 = \text{recv}(m_3), e_4 = \text{send}(m_4)$. Note that this particular test does not refer at all to the internal state of the CUT. We will always use traces instead of executions, when we are talking about black-box testing only.

Intuitively, we might write this execution as a graph such as shown in figure 5.2, meaning that on encountering event e_i in the execution we either transition

Table 5.1: A transition table for the DFA in figure 5.2.

	e_1	e_2	e_3	e_4	nil/\$
q_0	q_1				
q_1		q_2			
q_2			q_3		
q_3				q_4	
q_4					q_4/a

to the state pointed at by the arrow labelled e_i or fail the test immediately, if no such arrow leaves the current state. In this example, the test would be considered "passed" if the execution reaches q_4 and no more non-*nil* events follow.

While an edge-labelled directed graph may be easy to read for a human, it is neither pleasant to write down for a computer to interpret nor is it straightforward to interpret computationally when written down in the formal manner: As a vertex set V and a labelled edge set E, given as follows, together with a start transition to q_0 and an accept set $A = \{q_4\}$:

$$V = \{q_0, q_1, q_2, q_3, q_4\}$$
$$E = \{(q_0, q_1, e_1), (q_1, q_2, e_2), (q_2, q_3, e_3), (q_3, q_4, e_4)\}$$

A common way of writing this graph for execution by a program would be as a transition table, such as the one shown in table 5.1, where \$ marks the end of the execution (if any) and *a* means "accept".

However, neither this table form nor the formal (V, E) form are easily readable for a human, and no one would want to write either by hand when writing a testing specification. We need a way to describe S, such that it is easily writeable and readable by a programmer, and yet sufficiently formal that we can generate a machine-interpretable form, like a transition table, from it automatically.

5.2.1 Grammars as Test Specifications

Both the graph given in figure 5.2 and the transitions in the equivalent transition table 5.1 are different ways of describing a DFA. From the fields of formal languages and especially parser generators, we know well how to construct a DFA given a regular grammar G [52]. If we approach the above problem from this perspective, we might define $G = (N, \Sigma, P, \mathrm{S})$ with:

$$N = \{\mathrm{S}\}$$
$$\Sigma = \{e_1, e_2, e_3, e_4\}$$
$$P = \{\mathrm{S} \rightarrow e_1 e_2 e_3 e_4\}$$

```
1 S      → Body+
2 Body   → expect e_i+ | Kleene | Union
3 Kleene → repeat Body+ end
4 Union  → either Body+ or Body+ end
```

Listing 5.1: An example CFG for the specification of regular tests.

While that is already fairly readable, in terms of regular grammars, we can write it even shorter as a regular expression $R = e_1 e_2 e_3 e_4$. In either case we are treating the events e_i as if they were symbols in a string, which is too naïve a view to be practical, as events contain messages and messages are complex structures in any but the most trivial of programs. We will deal with this mismatch properly in section 5.2.2. A more immediate issue concerns the representation of such a regular expression in actual code. While most popular languages already provide built-in or library support for writing regular expressions, they are usually meant for recognising strings of characters and not sequences of events. Thus they are typically written as strings themselves, with certain special symbols (e.g., `*|()`) encoding the operations, instead of being interpreted as literals (terminals) to match. But, as pointed out above, events are not actually symbols we could simply embed in a string, and thus a way is needed to describe the desired regular expression in a similarly readable format without using strings. A common way of creating such custom formal code, is to design a DSL that can be converted into the actual instructions to be executed. Listing 5.1 shows an example of such a DSL, given as production rules, where bold words are terminals and e^+ means one or more repetitions of e, the same as ee^* in a regular expression. The described DSL allows the description of tests for regular sequences, or *regular tests*.

From what has been described so far it might seem convenient to choose a standalone (external) implementation of such a DSL, so that it may be used with any message-passing framework, exploiting the general nature of the approach. However, in practice this would not be feasible as 1) it would make white-box testing all but impossible and 2) matching events in an implementation-independent manner would be very limiting, as will become clear in section 5.2.2. For these reasons, it is all but mandatory to embed the DSL in the same language used by the message-passing system to be tested. This is the approach we have chosen for our prototype implementation as described in section 5.3.

5.2.2 Matching Events

In regular expressions for character strings, matching a single character is merely a bit-wise comparison, potentially with the added challenge of variable length encodings. If we abstract away from this detail, given an alphabet Σ we simply have an *equivalence relation* $\sim_\Sigma \subseteq \Sigma^2$. We say a character $c \in \Sigma$ in the input sequence *matches* an expected character $a \in \Sigma$ iff $c \sim_\Sigma a$.

When dealing with events, given a set M of possible messages, we have $\Sigma = D \times M$, where $D = \{\text{send}, \text{recv}\}$. Note that we have been and will continue to write both (d, m) and $d(m)$ with $d \in D$ and $m \in M$ to mean the same thing, for example both (send, m) and $\text{send}(m)$ mean that m is an outgoing message. There is a fairly obvious way to the define equivalence $\sim_D$ for D, that is $\sim_D = \{(\text{send}, \text{send}), (\text{recv}, \text{recv})\}$. For our Σ tuples, the relation would simply be:

$$\sim_\Sigma = \{((d_1, m_1), (d_2, m_2)) \mid d_1 \sim_D d_2 \wedge m_1 \sim_M m_2\}$$

The definition of $\sim_M$, however, depends on the actual structure of messages, which depends on the implementation language. In Erlang, for example, such an equivalence relation would be trivially available via the expression matching mechanism [9, section 3.3.3], and the same is true in Scala, at least for *case classes* and *objects* [84, chapter 8], while in Java a correct relation would require the developer to write correct `public boolean equals(Object other)` methods [41, section 4.3.2] for each class used as or in a message.

Whatever the specific method to define message equivalence, it will most likely be too narrow to be practical for testing. For example, we might not care at all for the actual content of the fields, but only wish to match a certain type of messages. Or maybe we only care about the content of a specific field, such as an addressee or source. If we are only given a single equivalence relation for matching, we would be forced to write unions over all possible values of the irrelevant fields, an approach that is clearly not realistic. String regular expression implementations often provide a predefined set of character classes, such as digits, or whitespace character, to avoid these kinds of large unions. But due to the wide variety of possible messages, such a predefined approach is not feasible in our case either.

The solution then is to use an even more general approach of allowing the developer to provide a *matching function*, essentially a user-defined predicate, $\mu : M \to \mathbb{B}$ instead of a specific event e_i as "expected". An event $e = (d, m)$ from the input sequence is considered *matched* by a tuple (d_{exp}, μ) iff $d \sim_D d_{\text{exp}}$ and $\mu(m) = \text{TRUE}$.

This approach is very powerful, but it comes with an important caveat: The developer must ensure that any DFA generated from the testing specification never has two outgoing edges with overlapping labels. That is, if the two edges are labelled with (d_1, μ_1) and (d_2, μ_2), respectively, and $d_1 = d_2$, then the *match sets* of μ_1 and μ_2 must be disjoint, i.e. $\mathcal{M}_1 \cap \mathcal{M}_2 = \emptyset$. The match set for a matching function μ_i is defined as:

$$\mathcal{M}_i = \{m \in M \mid \mu_i(m) = \text{TRUE}\}$$

Note that any DFA node may have an arbitrary number of outgoing edges, as long as all the *match sets* of its edge labels are disjoint. Violating this restriction would render the test nondeterministic. Note that, although we know how to convert a nondeterministic finite automaton (NFA) into a DFA [52], this method can not be applied here, as the nondeterminism between the μ_1 and μ_2 edges is

not explicitly known when generating the automaton, since we treat the matching functions as black boxes.

For the sake of brevity, we will continue to write $e_i = (d_i, m_i)$ both in executions and in test specifications, with the implicit understanding that in the case of test specifications we actually mean $e_i = (d_i, \mu_i)$ with $m_i \in \mathcal{M}_i$.

5.2.3 Extensions of Regular Tests

Like regular languages being closed under concatenation, union, and Kleene star, so are regular test specifications. However, the question arises if this provides sufficient expressive power to describe tests that are practical.
Clearly, the example specification from the beginning of the chapter,

> *S = "If I send a message m_1 to the component, then I expect to see only a message m_2 leaving it. Afterwards, giving it a message m_3 should cause emission of one or more m_4 messages."*

can be expressed in this model, as we already showed that it can be written as a regular expression $R = e_1 e_2 e_3 e_4 e_4^*$.

Similarly, if we expect to see either 1 or exactly 4 e_4 events at the end of S, we could write $R = e_1 e_2 e_3 (e_4 | e_4 e_4 e_4 e_4)$. This repetition is somewhat tedious to write, though, so it is common to define a short hand for fixed size repetitions and write $R = e_1 e_2 e_3 (e_4 | e_4^4)$ instead. This is not necessarily just a notational shorthand, but could also be used to arrive at a more efficient implementation, using a context with a loop counter, instead of generating all the states of the full DFA.

Another convenient extension is to allow the specification of unordered expectations. That is, instead of writing $R = (e_1 e_2) | (e_2 e_1)$, we could write $R = \text{all}(e_1, e_2)$. Again, such an extension can also lead to a more efficient implementation, for example using bit-sets to track seen events, instead of generating the states for all the possible permutations. It is clear that many such regular extensions are possible, and those we considered are described in more detail in the technical report about our prototype implementation [104].

Regular tests alone are already very powerful and can capture many common scenarios to be tested. But consider a very small variation of S, where m_1 and m_2 have to agree on a randomly generated `id` field marking m_2 as being a *response* to m_1, in particular, and not just any other message that looks like m_2, except for the `id`. As the `id` field is randomly generated at runtime, we have no way of statically writing a matching function μ for m_2, that could actually capture the runtime `id` of m_1.

5.2.4 Beyond Regular Tests

The kind of *request-response pattern* described at the end of the previous section is very common in message-passing systems, as it forms a logical bridge between

method invocation and message-passing models. Consequently, no testing solution can be considered suitable that can not adequately capture such behaviour.

Stack-based Approach As a first attempt, we will keep track of all requests we have seen so far by putting them on top of a stack $\sigma : \texttt{Stack}[M]$ when they are observed. We extend the matching function to the signature $\mu : M^2 \to \mathbb{B}$ and pass the request message on top of σ as the second parameter. Then the matching function could handle extracting the `id` fields from both messages, comparing them and deciding whether or not they match. If they do, the request is removed from the stack and the state transition is successful. This approach describes a *pushdown automaton*, which, as explained in section 5.1.3, recognises context-free grammars (CFGs).

Thus we could write a test specification S_1, where matching happens on the id i, as:

> S_1 = *"Any sending of a request $m_i^?$ must be immediately answered by receiving a matching response $m_i^!$."*

We can even have multiple responses outstanding, such as in S_2, for example:

> S_2 = *"Any sending of a request $m_i^?$ must be answered by receiving a matching response $m_i^!$ in a last-in, first-out (LIFO) manner."*

Both S_1 and S_2 are captured by the following set of production rules (with the other parts of the grammar being implicit):

S_1 :	S_2 :	
$\mathrm{S} \to SS$	$\mathrm{S} \to SS$	
$SS \to \mathrm{send}(m_i^?)\,\mathrm{recv}(m_i^!)$	$SS \to \mathrm{send}(m_i^?)\,SS\,\mathrm{recv}(m_i^!)$	for any id i
$SS \to \varepsilon$	$SS \to \varepsilon$	

Clearly, the language described by S_1 is a subset of the language described by S_2, i.e. $L(S_1) \subset L(S_2)$.

Indexed Grammars Note that we are treating every m_i as a unique symbol in Σ, as we assume the matching is handled by our matching function $\mu : M^2 \to \mathbb{B}$. In this sense, we are hiding complexity away from the grammar. Instead, we could also define a *linear indexed grammar* [52], that takes the id $i : \texttt{Id}$ into account, by exposing the stack $\sigma : \texttt{Stack}[\texttt{Id}]$ to the grammar. Such an indexed grammar G is a 5-tuple $G = (N, \Sigma, F, P, \mathrm{S})$, where N, Σ, S are as before, and F is the set of *index symbols*. The production rules P look as before, except that nonterminals are now annotated with the current state of the stack $\sigma : \texttt{Stack}[F]$, by writing $A[\sigma]$ for $A \in N$, or $A[\texttt{f}\sigma]$ for

`f` $\in F$ to denote that `f` is currently on top of the stack. $A[]$ denotes a production with empty stack. In this system, with $F =$ `Id`, S_2 can be more properly expressed as:

$$
\begin{aligned}
&S_2: \\
\mathrm{S}[\sigma] &\rightarrow SS[\sigma] \\
SS[\sigma] &\rightarrow \mathrm{send}(m_i^?)SS[i\sigma] \\
SS[i\sigma] &\rightarrow \mathrm{recv}(m_i^!)SS[\sigma] \\
SS[] &\rightarrow \varepsilon
\end{aligned}
$$

As only a single nonterminal symbol receives the stack in this new formulation of S_2, it is a linear index grammar, which is a subset of the indexed grammars.

Cross-serial Dependencies Inconveniently, neither S_1 nor S_2 are particularly realistic specifications for programs using the request-response pattern. Instead, S_3 is the most common specification for request-response communication:

> $S =$ *"Any sending of a request $m_i^?$ must be answered by receiving a matching response $m_i^!$ (eventually)."*

This specification allows traces such as

$$T = (\mathrm{send}(m_1^?), \mathrm{send}(m_2^?), \mathrm{recv}(m_1^!), \mathrm{recv}(m_2^!), \ldots)$$

, which are common, for example, when queueing up multiple operations on another component to be processed (e.g., sent over the network).

This relationship between requests and responses is referred to as a *cross-serial dependency* in language research and it has been shown that languages with cross-serial dependencies are not *context-free*, but are generated by context-sensitive grammar (CSG) of type 1 [57]. Although some cross-serial dependencies are simple enough to be captured by indexed grammars, as described above, in our example this is not the case. However, if we allow lookups for ids at any place in the stack σ — making it more of an id set in the process — we can express the production rules for S_3 with a similar notation as the one used previously for the indexed version of S_2:

$$
\begin{aligned}
&S_3: \\
\mathrm{S}[\sigma] &\rightarrow SS[\sigma] \\
SS[\sigma] &\rightarrow \mathrm{send}(m_i^?)SS[i\sigma] \\
SS[\sigma_1 i \sigma_2] &\rightarrow \mathrm{recv}(m_i^!)SS[\sigma_1\sigma_2] \\
SS[] &\rightarrow \varepsilon
\end{aligned}
$$

For our implementation [104], we have opted to provide only specialised support for recognising these request-response patterns, instead of giving full CSG support.

The approach is similar to the S_3 grammar described above, In that the generated automaton keeps a set O of outstanding requests in a context, and marking a barrier (e.g., the end of a repeat block, or the end of the test) by which time O must be empty, or the test case is considered failed. This is discussed in section 5.3.3.

Increasing the expressive power further, O can be extended to keep a count of expected responses, which is decremented every time a matching response is received. Once the counter reaches 0, the entry is removed from O. This is equivalent to adding i multiple times to the id set in the grammar above. In this way the common pattern of broadcasting a request and waiting for n (e.g., a majority) responses can easily be handled.

5.2.5 Observation vs. Interaction

In addition to the passive matching primitives we have described so far, which come from language-recognition, the particular field of *testing* also requires a form active participation by the test itself, in order to cause the generation of an execution that is relevant to the properties to be tested.

Recall once again that the example specification S from the introduction begins with the words *"If I send a message m_1 to the component..."* and the given trace starts with $T = (recv(m_1), \ldots)$. There is a mismatch in that we say *"I send a message"* in S, but in T we are only looking at the result of a message having been sent. Throughout the text so far, we have silently assumed that when a message is supposed to be sent somewhere during the test execution, it will just happen and we can simply *observe* the result. Clearly reality does not conform to our whims in this manner, and instead we must find ways to cause these messages to be sent. One possible approach is to use or implement other components or actors, referred to as *external dependencies*, that communicate with the CUT in the expected manner, allowing us to intercept and observe the communication in the same way as we have done so far. Implementing such dependency components just for the sake of testing is usually referred to as *mocking* [76], as typically only the minimum behaviour necessary to facilitate the test is implemented. This approach is often preferable to using production components as dependencies, as it gives better control over the test behaviour and reduces the risk that a complex dependency causes the test to fail, or, even worse, to spuriously pass.

Even finer control over the execution can be achieved by, instead of mocking a whole component, mocking only very specific behaviour through insertions of messages at specific positions in the execution. This approach is a form of *interaction* and requires the testing automaton to be run *online* on the stream of events, instead of evaluating a pre-recorded sequence of events. This way of testing has both advantages and disadvantages, which will be discussed in section 5.2.6. Given such an online automaton, we may then introduce *active states*, that is states which send a message upon entry instead of (or in addition to) expecting one to allow a transition.

```
1 S      → Body+
2 Body   → Action | Kleene | Union | Loop
3 Kleene → repeat Body+ end
4 Union  → either Body+ or Body+ end
5 Loop   → repeat n body Body+ end
6 Action → expect e_i+ |unordered e_i+ |trigger m_i+
```

Listing 5.2: An example CFG for the specification of online tests.

To incorporate such active states into our test description, we use a new alphabet $\Sigma_A = \{\text{expect}\} \times (D \times M) \cup \{\text{trigger}\} \times M$ in place of the previous alphabet $\Sigma_E = D \times M$. Given this, we may write the test specification using behavioural mocking for S as:

$$\text{trigger}(m_1) \cdot \text{expect}(\text{send}(m_2)) \cdot \text{trigger}(m_3) \cdot \text{expect}(\text{send}(m_4))^*$$

Listing 5.2 shows an extension of the DSL from listing 5.1, incorporating actions and the regular extensions described in section 5.2.3.

All three approaches, external dependencies, component mocking, and behavioural mocking are valuable in a testing framework and are supported by our prototype implementation.

5.2.6 Online vs. Offline Recognition

Apart from enabling the valuable interactive behavioural mocking described in the previous section, online test execution also allows for test runs to be eagerly aborted as soon as the specification is violated. With long tests covering many repetitions of a sequence, for example, this can be very valuable, as it reduces the turnaround time between detecting and fixing bugs. Offline testing, on the other hand, would require the whole test to be run to the end in any case, as violations are only detected on the complete, recorded execution. Furthermore, a recorded execution for a large test case could potentially be *very* large in terms of storage space requirements. This is especially true if intermediate internal states are recorded for white-box testing (cf. section 5.2.7).

However, a difficult issue that occurs in online testing is that of deciding when the test should terminate. In a recorded sequence it is clear that when the end of the sequence is reached, abstractly denoted as a stop sign \$, and the testing DFA is in an accept state then the test is passed. For an online test a decision has to be made as to whether the test is passed as soon as any accept state is reached, or how long to wait after reaching such a state to ascertain that no (un-)expected events follow, which would fail the test or transition to another accept state. In fact, if the test should end on a Kleene star, such as our running example with $R = a_1 a_2 a_3 a_4 a_4^*$, the end of the test itself is ambiguous. Deciding to terminate on the first accept state would mean testing for $R' = a_1 a_2 a_3 a_4$ instead of R. This issue is not generally

```
1 S      → Body+
2 Body   → Action | Kleene | Union | Loop
3 Kleene → repeat Body+ end
4 Union  → either Body+ or Body+ end
5 Loop   → repeat n body Body+ end
6 Action → expect e_i+ |unordered e_i+ |trigger m_i+ |inspect α_i
```

Listing 5.3: An example CFG for the specification of white-box tests.

solvable for online testing in the way described here. A practical approach is to define some sufficiently long *timeout*, a decision that can only be made by the test developer, for tests to ensure termination at all, and to avoid writing tests that end in an ambiguous manner.

Another issue with interactive online testing concerns possible interleavings of messages at runtime. This emerges because message-passing systems are typically designed to be executed in a concurrent manner, which means that there is some scheduling mechanism for components or actors involved. However, tests that need to insert messages at specific positions into the event stream might not be able to tolerate other interleavings. The solution for this issue strongly depends on the implementation of the target framework with respect to scheduling. A sensible approach, if possible, is to execute the test in single-threaded mode and ensure that the CUT is always scheduled together with the testing automaton. This is especially crucial when mocking a real-time abstraction, for example, as this requires a very fine degree of control over the actual execution sequence.

5.2.7 Inspecting State

So far we have only concerned ourselves with black-box testing. In fact, in the offline model white-box testing would be impossible without additionally recording every intermediate state of the CUT. The introduction of online testing allows us to inspect the component's state at any point we desire, provided the implementation deals with the scheduling issue described in section 5.2.6 and respects the model assumptions that computation steps are atomic. Like the triggering of messages, inspecting component state is another interactive addition to our model.

On a conceptual level the internal state inspection would take the form of an assertion function $\alpha : \mathcal{S}_c \to \mathbb{B}$, taking the current internal state $s_i \in \mathcal{S}_c$ of the CUT c, such that $\alpha(s_i) = \text{TRUE}$ iff s_i conforms to the test specification. On a more practical level, it may be convenient to simply pass c itself to α and reuse existing imperative unit testing solutions, appropriate to the programming language in use, to decide whether or not c's internal state conforms to the specification.

To incorporate state inspection into our model, we again extend the alphabet Σ_A to $\Sigma_I = \{\text{expect}\} \times (D \times M) \cup \{\text{trigger}\} \times M \cup \{\text{inspect}\} \times \mathcal{A}_c$ where $\mathcal{A}_c$ is the set of all

```
1 S      → Body+
2 Body   → Action | Kleene | Union | Loop
3 Kleene → repeat Header* body Body+ end
4 Union  → either Body+ or Body+ end
5 Loop   → repeat n Header* body Body+ end
6 Header → allow e_i+ |disallow e_i+ |drop e_i+ |blockExpect e_i+
7 Action → expect e_i+ |expect unordered e_i+ end |trigger m_i+
8        | inspect α_i
```

Listing 5.4: The CFG for the test specification DSL of the prototype.

assertion functions $\alpha : \mathcal{S}_c \to \mathbb{B}$. An extended DSL incorporating these changes is shown in listing 5.3.

5.3 KompicsTesting

As part of his master's thesis [103], Ifeanyi Ubah has written an implementation of the model laid out in the previous section, which we will briefly describe here. A longer treatment of the implementation, called KompicsTesting, in particular can also be found in [104].

Kompics-specific Model Extensions In order to support Kompics in our model, we need to extend our alphabet once more, such that we can incorporate the concept of port types into the description of an event. This is semantically important, as triggering the right message on the wrong port would typically imply a different behaviour. The ports in question could be connected to different components, for example, or even if its the same component, they may have different handler subscriptions. This, of course, would cause a different handler implementation to be invoked. Thus, if P is the set of port types in a system, then the alphabet for a test specification is $\Sigma_P = \{\text{expect}\} \times (D \times P \times M) \cup \{\text{trigger}\} \times (P \times M) \cup \{\text{inspect}\} \times \mathcal{A}_c$.

5.3.1 Specification Builder

As indicated already in section 5.2.1 we opted for an eDSL for our prototype. As we are using Kompics Java as a target, we had to embed the DSL in the Java language, which is syntactically rather strict and verbose and thus not quite optimal for building DSLs. The most reasonable approach for eDSLs in Java tends to be the builder pattern [38]. The full DSL currently implemented is given in CFG form in listing 5.4, while listing 5.5 gives an example of how the builder pattern is used in

```
bd.repeat(5)
    .allow(e1, p1, in) // block header
  .body() // block body
    .expect(e2, p1, in)
    .expect(e3, p2, out)
  .end();
```

Listing 5.5: A test specification in Java created using a builder pattern.

Java to create an actual specification, in this case for the expression:

$$\begin{aligned} R = &(\text{recv}(p_1, e_1) \cdot \text{recv}(p_1, e_2) \cdot \text{send}(p_2, e_3) \\ &| \text{recv}(p_1, e_2) \cdot \text{recv}(p_1, e_1) \cdot \text{send}(p_2, e_3) \\ &| \text{recv}(p_1, e_2) \cdot \text{send}(p_2, e_3) \cdot \text{recv}(p_1, e_1) \\ &)^5 \end{aligned}$$

In this expression, the p_i are port instances and the e_i could be event instances, events types, or predicates matching events, which may be given either as a `java.util.function.Predicate` instance or as an equivalent *anonymous function*.

5.3.2 Runtime

The theoretical framework, described in section 5.2, assumes that events incoming to and outgoing from the CUT are not only observable by an implementation, but also interactive in the form of triggers and state inspection. KompicsTesting interacts with Kompics' core runtime using a debugging API, that notifies a supplied *tracer* instance of incoming and outgoing events for observation. The use of the *tracer* API incurs no runtime cost during a real deployment, as the *tracer* is supplied during component instantiation for tests to the runtime, not the user code. From the perspective of the user code the presence or absence of a *tracer* is not observable.

Event triggers from the testing framework are handled like any other events, with no special consideration required.

The generated automaton for a test specification is simulated on a dedicated thread separate from the thread pool on which Kompics components such as the CUT are scheduled. This allows the automaton to progress at its own pace, with the events observed by the *tracer* applied as input symbols. Synchronisation between the automaton and *tracer* is implemented using an event queue, where the *tracer* is the producer and the automaton's thread is the consumer of events. Only when an event is consumed from the queue can it be *handled*, that is forwarded to the recipient(s), as determined by the transitions of the current state.

The same mechanism also ensures that a state inspection predicate can be executed without risking concurrent access or unexpected interleavings. However, Kompics, like many other message-passing systems such as Akka, does not actually

satisfy the assumption of the computational model that computation steps are *atomic*. For performance reasons, events triggered during a handler execution are forwarded immediately, and not buffered until the handler completes. Kompics also allows multiple handlers to be invoked in response to a single event, thus extending the necessary buffering scope even further. It follows that, while it is possible to do state inspections in KompicsTesting, they do not happen at semantically intuitive places in the execution. A similar problem has been encountered by the creators of Setac [94], who chose to wait for a system to stabilise, i.e. stop processing messages, before attempting inspection. As they recognised, this might not ever happen in systems processing regular timeout events, and providing a timeout value, after which to inspect, results in an even less clear position of the inspect in the resulting schedule. For these reasons we have elected not to attempt such a "system state"-based solution.

5.3.3 Answering Requests

As mentioned in section 5.2.4 we provide specialised support for request-response patterns in our prototype, allowing reponses to instead be generated by the user on receiving a matched request and triggered at a specified point on some destination port.

The user provides a mapper function $\rho : M^? \rightarrow M^!$ for each request that returns a unique response if a provided request is matched. It is also possible to wrap ρ into a `java.util.concurrent.Future` instance, thus making the data closed over by the mapper explicitly available to later points in the scenario.

The start point of the pattern is marked by calling the `answerRequest(...)` method, passing it either a mapper or a future, the event type, and the port. The direction for this pattern is always `OUT` as incoming requests would be handled by the CUT and not mocked. Listing 5.2 demonstrates basic usages of this mechanism.

5.4 Example — A Chat Application

To illustrate the usage of KompicsTesting, consider the example, inspired by Veanes et al. [47], of a simplified distributed chat component, that is reactive and can deal with a dynamically changing number of clients. The chat component makes use of the *eventually perfect failure detector* (EPFD) and *best effort broadcast* (BEB) abstractions from [16], which we have described before. Both abstractions are modified for dynamic process sets, instead of the a-priori fixed set, as described in [16], and can be seen in listing 5.2 lines 2-9.

A client begins by entering the chat room, which should result in it being added to the BEB process set and the EPFD process set (lines 33-43). From that point the client should receive all messages broadcast by BEB (lines 44-46), and messages it sends should be received by all processes in BEB's set, including the client itself (lines 50-55). If a process crashes, as detected by the EPFD, it should be removed

```
// Ports
public class EPFD extends PortType {{
 request(Watch.class);
 request(Unwatch.class);
 indication(Suspect.class);
 indication(Restore.class);
}}
public class BestEffortBroadcast extends PortType {{
 request(Broadcast.class);
 request(Member.Add.class);
 request(Member.Drop.class);
 indication(Broadcast.Deliver.class);
}}
public class ChatPort extends PortType {{
 request(Enter.class);
 request(Send.class);
 indication(Receive.class);
 indication(Status.class);
}}
// Test Suite
public class ChatTest {
 UserId node1 = UserId.random();
 UserId node2 = UserId.random();
 @Test
 public void failureTest() {
 // Setup
 TestContext<SimpleChat> tc = TestContext.newInstance(SimpleChat.class);
 Positive<ChatPort> chatPort = tc.getComponentUnderTest().provided(ChatPort.class);
 Negative<BestEffortBroadcast> bebPort = ...;
 Negative<EPFD> fdPort = ...;
 // Messages
 WatchFuture watchF = new WatchFuture();
 Send msg1 = new Send(node2, "test1");
 Send msg2 = new Send(node1, "test2");
 // Schedule
 tc
  .allow(Status.class, chatPort, OUT) // ignore Status messages
  .body()
   .trigger(new Enter(node1), chatPort) // join node1
   .unordered() // make sure the join is forwarded to FD and BEB
    .expect(new Watch(node1), fdPort, OUT)
    .expect(new Member.Add(node1), bebPort, OUT)
   .end()
   .trigger(new Enter(node2), chatPort) // join node2
   .unordered() // make sure the join is forwarded to  FD and BEB
    .answerRequest(Watch.class, fdPort, watchF) // put Watch event in the context
    .expect(new Member.Add(node2), bebPort, OUT)
   .end()
   .trigger(new Broadcast.Deliver(msg1.recv()), bebPort)     // mock node2 having sent msg1
   .expect(msg1.recv(), chatPort, OUT) // check msg1 delivery
   .trigger(watchF.suspect(), fdPort) // mark node2 as crashed
   .expect(new Member.Drop(node2), bebPort, OUT) // check it gets dropped from BEB
   .trigger(msg2, chatPort) // send msg2
   .expect(new Broadcast(msg2.recv()), bebPort, OUT) // check it gets broadcast
   .trigger(new Broadcast.Deliver(msg2.recv()), bebPort) // mock self delivery
   .expect(msg2.recv(), chatPort, OUT); // check msg2 delivery
  assertTrue(tc.check()); // run the test case
}}
```

Listing 5.2: An example test for a simple Chat component with KompicsTesting and JUnit.

Table 5.2: Detailed classification results of all code permutations.

Classification	Count	% of Total
Total	411	100.00 %
Original	1	0.24 %
Not Tested	319	77.62 %
Tested	91	22.14 %
Classification	**Count**	**% of Tested**
Detected	58	63.70 %
Path Not Tested	27	29.70 %
Internal	4	4.40 %
Optimisation	2	2.20 %
Not Expressible	0	0.00 %

from everyone's BEB set (lines 47-49). Crashes should not impact message exchange on correct nodes. Additionally, the chat component also emits `Status` updates on every event, which might cause user interface (UI) updates, for example, but are not relevant for the test case and thus explicitly ignored (line 31).

In addition to the actual test scenario, listing 5.2 also includes the necessary setup code, such as creating the testing context for the CUT (line 21), acquiring references to its port instances (lines 22-24), and preparing messages (lines 27-28) and a request future context (line 26, cf. section 5.3.3) for later use in the scenario.

5.5 Evaluation

We evaluated the KompicsTesting prototype on a video-streaming open source project that had been migrating its unit tests to KompicsTesting for about 6 months. According to Jacoco [51], the project has 706 classes with 2117 methods and all its unit tests (not just component tests) combined give it a code coverage of 21 % of the total instructions and 12 % of the total branches.

To evaluate the prototype, we analysed the Kompics components in the project and for each occurrence of a `subscribe` or `trigger` call, we generated a permutation of that particular class where the statement was commented out. As `subscribe` and `trigger` are side-effecting from the compiler's point of view, this covers two very common error sources that can not be statically detected, and thus does not affect whether or not the code compiles.

By this method we generated a total of 411 permutations (including the original code). For each permutation (sequentially) we continued to re-compile the project with the altered class instead of the original file. We then ran its complete test suite, recorded whether it passed or failed, and additionally measured the execution time on a Intel Core i7 (Haswell) 3.4 GHz system with 16 GB of memory. If the test coverage for the components would have been perfect and there was no dead-code

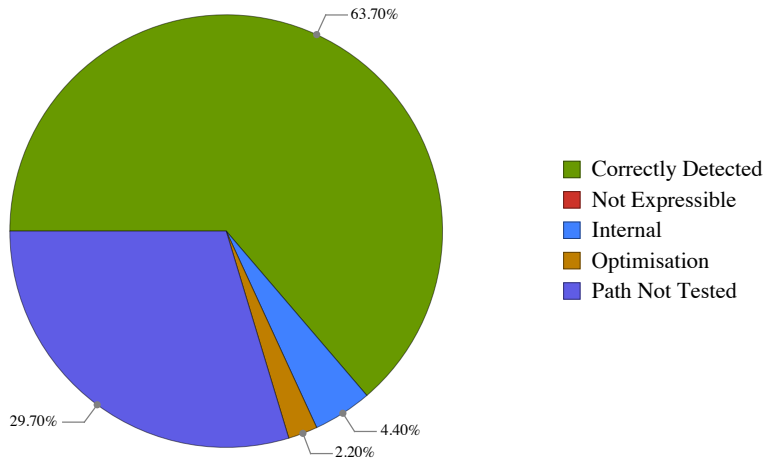


Figure 5.3: Overview of the relative composition of classifications of tested permutations.

in the tests, the expected outcome would have been that 410 of the 411 permutations, that is every permutation except the original run, would fail the tests, as each permutation introduces a bug.

Expressivity As already indicated by the low overall test coverage, this was, of course, not the case. The results show that only 91 of the 411 permutations are covered by unit tests at all (cf. table 5.2), which is surprisingly consistent with the overall coverage of 21 %. This indicates that the project has primarily, if not exclusively, KompicsTesting based unit tests.

Of the 91 permutations actually covered by tests, 58 (63.70 %) were correctly detected as faulty by the test suite. In the remaining 33 permutations the change was not discovered by the tests, and we investigated the individual permutations and tests to classify the reason for the faulty code passing through the test.

For 27 of the tests it was a simple case of the test schedule never taking the particular path through the possible branches in which the permutation occurred. These cases could easily be fixed by simply describing more schedules in new testing automata. In four of the cases, it turned out that the permutation was preventing some internal components from starting, which, if started, would have operated independently of their parent and would not affect its external interface

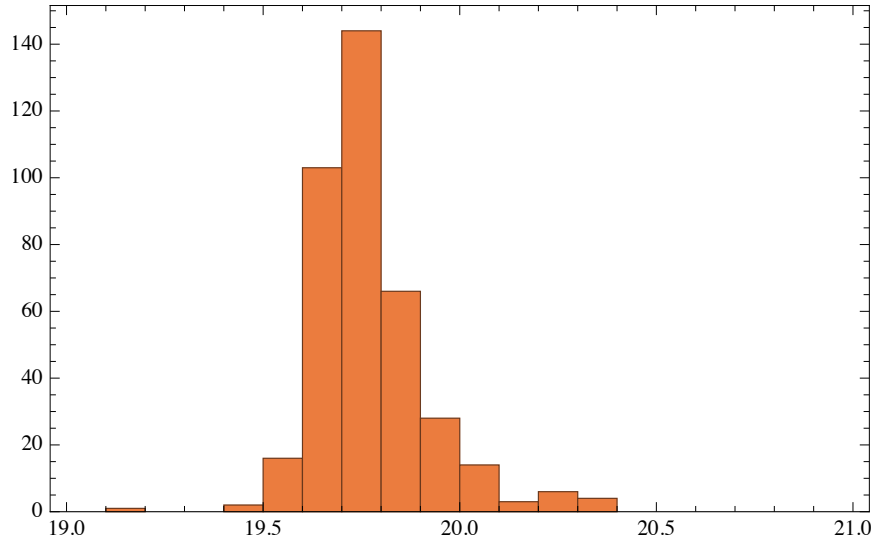


Figure 5.4: Distribution of test execution time for the permutations of the test suite in seconds.

and are thus not observable to the testing framework. The last class of undiscovered permutations turned out to be optimisations that did not affect the correctness of the component. An example for this is asynchronously cancelling a timeout, without knowing if the timeout message was already queued up anyway and thus could not be cancelled any longer.

None of the investigated permutations turned out to be not tested, because KompicsTesting could not express their properties. This supports our stance that the testing model presented is sufficiently expressive for the vast majority of real-world applications.

Performance In addition to investigating the test coverage, we also measured execution time and memory usage of the test suite, and counted the number of testing automata used and the number of states in each automaton.

The distribution of runtimes, as seen in figure 5.4, was strongly clustered around its median of 19.75 s. However, some exceptionally high values shifted the sample mean to 89.27 s with an expectably high sample standard deviation of 345.62. These outliers were caused by test cases failing due to a timeout, instead of a schedule violation and are cut off in figure 5.4 for readability.

The memory usage for the whole unit testing suite including *Maven Surefire* overhead was measured below 450 MB (resident set as given by `ps`) for every run.

There were a total of 35 testing automata described in the test suite, with an average of approximately 10 states per automaton. As can be seen in figure 5.5,

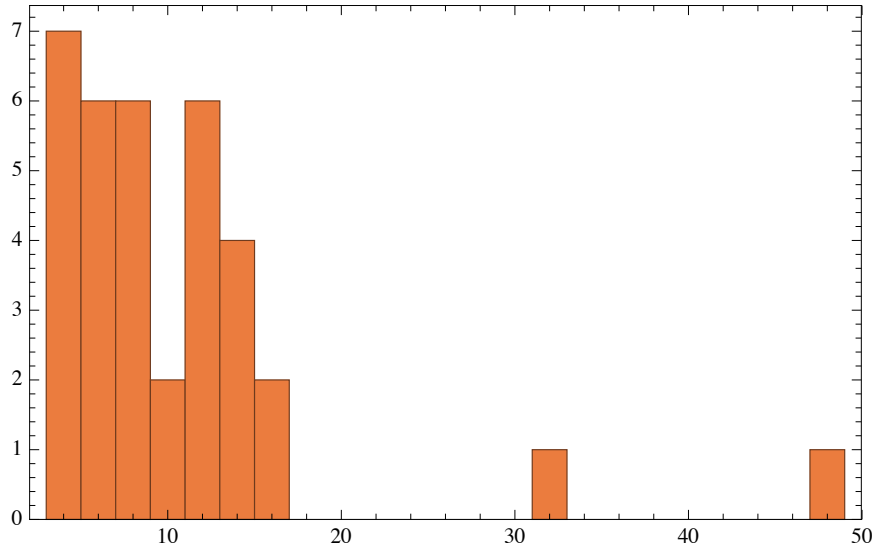


Figure 5.5: Distribution of the number of states of all the testing automata in the test suite.

there were some very small automata that were basically just testing correct setup and starts of the component hierarchy, with barely any schedule to speak of. On the other end of the spectrum were two rather large automata with 32 and 47 states respectively.

5.6 Related Work

5.6.1 Model-based Testing

Model checking or model-based testing (MBT) is a formal method used to verify the conformance of a given model of a system to its intended specification, in the form of properties that the system must satisfy. Many model checking methods use state-space exploration [23], and can instil more confidence in a system's implementation by automatically enumerating and exhaustively exploring the state-space of the system in order to find errors. If all paths in the state-space have successfully been executed, then the system is said to be correct.

Generating the state-space of a message-passing system can be an expensive process, and it is not uncommon for the result to consist of an exponentially larger number of states than the number of components and messages, or even an infinite number of states; a problem known as *state-space explosion*. Techniques such as partial-order reduction [77, 14] and dynamic partial-order reduction [67] exist to try to mitigate this problem by reducing the number of explored states.

There has a been large number of approaches to MBT [91, 27], of which only those that deal with concurrent and, particularly, message-passing systems are relevant in our context. Of the non-message-passing approaches, *Spec Explorer* [107] and CONCURRIT [30] stand out, because both of them also provide a DSL to describe model transition systems represented as FSMs.

Message-passing Model Checkers More closely related to our work are those model checking methods that have been applied to the testing of message-passing systems.

Basset [66] provides a model checker for actor systems that compile to Java bytecode, built on top the *Java PathFinder* [109], a more general model checker for Java programs.

Jacco [117] improves on the Basset system by implementing a new scheduling approach and technique for reducing the state space of the actor system in order to reduce the time taken for model checking.

McErlang [36] is a model checker for Erlang programs. It allows the user to encode correctness properties of the system as automata specified in Erlang, while the model checker checks the system against the specified properties. However, it uses state-space exploration for systematic checking of the system against the specification and suffers from the state-explosion problem especially in the face of nondeterminism.

Bita [95] is also targeted at actor systems, but focuses mostly on schedule generation.

5.6.2 Imperative Unit Testing

At the opposite end of the spectrum are more imperative unit testing approaches, such as the well-known *JUnit* framework [98] and *ScalaTest* [108], which focus on the interaction with objects via method invocation and field inspection.

These methods can also be applied to test message-passing systems, for example, by using custom testing harnesses providing concurrent queues and other coordination primitives such as latches, for the testing thread to track and verify progress in the message-passing system. The core runtimes of both Kompics Java and Kompics Scala, for example, are tested in this manner.

5.6.3 Schedule-based Unit Testing

The *Akka TestKit* tool [70], on the other hand, provides a platform for performing unit and integration testing on actor systems based on the Akka framework [69] at various level of granularity. It also allows the user to test that incoming sequences of events are processed correctly, even in the face of nondeterminism, leading to reordering of events as part of its DSL. However, such tests can only be performed using multiple actors to generate the stream of events, unlike our approach that allows an interactive mechanism for generating streams of events. Furthermore,

Akka TestKit does not use a language or automata based approach, nor does it require the tester to explicitly describe the expected behaviour of the actor. It does, however, shine with a more readable DSL than KompicsTesting, enabled by the use of Scala instead of Java for the embedding.

Another very closely related approach for *Scala Actors* [44], which is now deprecated in favour of Akka, is *Setac* [94]. Like KompicsTesting, Setac allows the programmer to specify allowed schedules with constraints and then execute those schedules and verify that the constraints are maintained. However, the Setac authors do not provide a formal description for how the schedule DSL relates to the test implementation that executed during the test run, nor do they clarify whether or not the implementation actually represents a state machine. Furthermore, Setac can only test actors that extend a `TestSubject` (Scala) trait. This limits testing to classes with control over the source code and adds potentially unnecessary testing code to the system, when it is deployed on a cluster. KompicsTesting has no such limitation, as the testing framework only interacts with Kompics' core runtime.

CHAPTER

6

Benchmarking Message-Passing Systems

In this chapter, we describe a framework for implementing and running performance benchmarks, both local and over a network, for message-passing systems. We then use this framework to investigate a number of common message-passing systems, and describe the state of-the-art in their performance on a wide range of different benchmarks.

Over the course of the last three chapters, we have mostly been dealing with the issues of *safety* and *correctness* in distributed programming abstractions. But the safest and most correct system in the world is not useful if it can not meet the demands placed on it in terms of performance. And while the performance of distributed applications is often mostly determined by the scalability of the algorithms implemented, having an efficient runtime for the implementation can save a lot of resources and thereby reduce the running costs of the system.

While *safety* can often be proven, and *correctness* can perhaps be tested for, *performance* aspects must be always *measured*. That is, while theoretical results about computation or message *complexity* provide useful information about the limitations of an algorithm, they have comparatively little influence on the real world performance of an actual implementation of that algorithm. Too many factors can influence the practical performance of an implementation, ranging from hardware scheduling, over the impact of caches, to the effects of many different — possibly independent — data flows on individual buffers in a network stack, for example. The only reliable mechanism for establishing the performance of an implementation is to measure it, preferably under circumstances similar to those encountered in the deployment environment it is targeted at.

Benchmarking The process of measuring the performance of a piece of code is often referred to as *benchmarking*, and for any benchmarked software a variety of metrics can be collected, as is appropriate for the performance question being evaluated. Common metrics include: *execution time*, *throughput*, and *latency*, as well as *resource usage* for different resources, for example, in terms of memory or bandwidth.

Benchmarking can also be performed at a variety of levels, ranging from *microbenchmarks* that measure a small area of code for a particular behaviour, such as the execution time of a single function, for example, to large scale system benchmarks, such as the database benchmarks from the *Transaction Processing Council* (TPC), which form the foundation of standardised database benchmarking.

Parallel and Distributed Programming Frameworks A particular challenge when designing and implementing frameworks for writing parallel and distributed systems in, is the ability to consistently compare their performance to other systems in the field. This is in part because these frameworks do not represent applications or services with a clear function and a fixed interface that can easily be measured against, as opposed to, for example, a database with a clear query interface such as SQL. Quite the opposite, these frameworks are more like toolboxes for building such applications and services. And thus the performance question shifts from *"How fast is my application using this service?"* to *"if I implemented my services in this framework, how would it perform?"*. Thus, instead of being able to run a standardised test against a particular interface, for each programming framework a particular

benchmark must be implemented, individually, and then the performance for the framework on that particular application can be measured.

One of the largest approaches to provide a standardised benchmarking suite for implementations of the Actor model, in particular, is *Savina* [54]. It provides a total of 30 benchmarks, covering a variety of common usage patterns, for JVM languages. The authors of Savina have taken great care to ensure the comparisons are as fair as possible, by reusing most of their benchmark implementation code for the nine different frameworks they tested. The approach taken by Savina, however, has two important shortcomings:

1) The Savina suite is targeted purely at parallel programming performance, and no attempt is made to compare frameworks on how well they perform in distributed settings, when their particular network stacks are involved. This information, however, is of crucial importance when selecting a framework to implement a distributed application in, as network performance will often be the major limiting factor in the final product.

2) Savina targets *only* JVM frameworks. While the authors are correct, that fair comparisons across languages are difficult if not impossible — depending on the definition of "fair" — to achieve, the attempt must still be made, if one is to truly build a standardised benchmarking suite for message-passing frameworks.

The first point is addressed in other benchmarking suites, such as `bencherl` for Erlang, while second one is targeted by Cardoso et al. in [19]. However, no solution has emerged that combines cross-language, distributed benchmarks into a standardised package.

In this chapter, we will present an approach and a framework to provide both parallel and distributed benchmarks for message-passing systems, that are independent of runtime environment[1] or language. We also describe a number of benchmarks, both microbenchmarks such as the ones from Savina, and larger system benchmarks testing simple systems, such as a key-value store. While we can not provide the same number of benchmarks as Savina, since the additional languages add significant development overhead, the system is explicitly designed to be extensible, such that more benchmarks can be added in the future. In particular, we would like to see all Savina benchmarks ported over to complement the available microbenchmarks for parallel systems.

We will refer to both the benchmarking framework and the set of implemented benchmarks as the message-passing performance (MPP) suite in this dissertation, and specify whether we mean the framework or the benchmarks where necessary.

[1] Assuming, of course, that the given environment is able to compile and run the provided code.

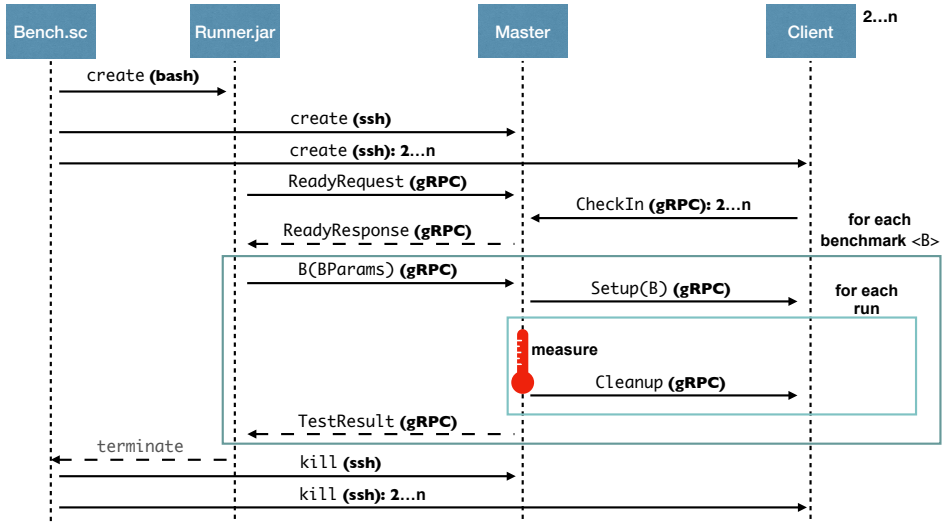


Figure 6.1: Overview of the distributed control flow for MPP benchmarks.

6.1 A Distributed Benchmarking Framework

The MPP framework consists of three major parts:

1) A set of scripts to build and run all the benchmarks, as well as to generate plots of the results.

2) A *Runner* binary, that orchestrates automated benchmark execution, and collects the results of the different experiment runs.

3) A set of benchmark descriptors in Google's *Protocol Buffers* format (*protobuf* for short), which compile to the different target languages and form the basis for language specific benchmark implementations. Additionally, there is a reusable library implemented for each language that is already supported, which wraps around the generated code and provides standardised ways to implement tests in any framework using that particular programming language.

These parts will be described in the rest of this section in detail, after we give a high level overview of how the different parts interact when benchmarks are executed.

6.1.1 Overall Execution Flow

As the MPP suite targets multiple different programming languages as "testees", it runs a number of different executables for every benchmark. The exact set of executables depends on the framework currently being tested, as well as the testing mode, but at least the `bench.sc` script and the `Runner.jar` are always in use. The MPP framework currently supports three different running modes:

local During purely local execution, only a single process in the testee language in started, and no distributed tests are run. This mode is essentially equivalent to Savina or the work in [19].

remote In this execution mode a configurable number n of testee language executables is started via `ssh`. One of them will be designated the *master*, while the others act as *clients*.

fakeRemote Like **remote**, except that all executables are started in their own temporary directory on the same host. This mode is mostly used for testing, but can also be used to measure loopback networking performance, which avoid real network latency artefacts in the results.

Figure 6.1 shows an overview of the execution flow between the different executables involved in a test run for a single testee framework, for the **remote** (or **fakeRemote**) scenario. **Local** mode is similar, except that master-client communication is not used. A full test run over all supported testee frameworks involves repetition of the flow described in figure 6.1 for each testee.

Execution Flow The `bench.sc` script begins by creating instances of the runner, as well as each of the n testees, which it designates to be either a master or a client. The runner will wait for the master to become ready, before initiating the first benchmark. The master is considered ready, when all $n-1$ clients have checked in, a process that informs the master which ports they ended up running on.[2] All communication between the runner, master, and client happens via gRPC, which is based on Google's Protocol Buffers.

For each available benchmark, the runner informs the master about the parameters to use when running it, and then waits for the run to complete. Typically each benchmark will be run with a range of different parameters, in order to explore different parts of its performance space.

Upon receiving the request for a new benchmark, the following sequence of events takes place:

1) The *master* configures itself using the parameters provided. This is called the *setup*-phase.

[2] This saves hardcoding too many port numbers, which can be problematic, especially in **fakeRemote** mode.

2) It then instructs each *client* to set itself up for for the benchmark. It can pass custom parameters to all clients during this part of the *setup*-phase.

3) Once all clients have responded that their setup succeeded, the *master* moves to the *prepare*-phase, where it sets up the first benchmark run. Clients also have a *prepare*-phase, but it does not involve a network round-trip. The first client *prepare*-phase is completed as part of their *setup*-phase, before responding to the master.

4) The *master* runs the benchmark and measures the total *execution time*. Benchmark start and end times are only recorded at the master, while clients only act as resources to distribute work over.

5) Once the benchmark run completes, the *master* starts a *cleanup*-phase both on itself and on all clients, where all benchmark related state is reset to the way it was before the last *prepare*-phase. This ensures that all runs start with similar conditions, but without losing warm-up benefits, such as JVM JIT compilation, for example. Unless the *cleanup*-phase is invoked with the *final* marker (s. below), it also implies a *prepare*-phase for the next run, both on master and all clients.

6) With the *cleanup*-phase complete, the *master* begins again at step 4, and continues to do so until a configurable statistical target is reached. The current default settings are to perform at least 30 runs, then measure the relative standard error (RSE), and complete the benchmark once it falls below 10 %. If this does not happen after 100 runs, then the benchmark is aborted and no values for that parameter set are recorded.

7) If the benchmark completed successfully, the *master* completes a *final cleanup*-phase, which prepares all involved testee instances to run a different benchmark. It then returns the list of *execution times* of all the individual benchmark runs to the runner for storage and analysis.

At the runner, the results will be written into comma-separated values (CSV) files together with the parameters that were used for the benchmark. Once the last benchmark instance completes, the runner terminates.

The `bench.sc` script takes the termination as a signal that the benchmarking for this particular testee framework is complete, and kills all instances of its executables via `ssh`. Once the last testee completes, the `bench.sc` script terminates as well.

Typically, the `bench.sc` script, the runner, and the master are all located on the same physical machine. While the master can also be located somewhere else, if desired, this leaves the machine running the script and the runner mostly idle, which only makes sense if it is too weak to participate in performance experiments, such as a laptop, for example.

```
val throughputPingPong = Benchmark(
    name = "Throughput Ping Pong",
    symbol = "TPPINGPONG",
    invoke = (stub, request: ThroughputPingPongRequest) => {
      stub.throughputPingPong(request)
    },
    space = ParameterSpacePB.cross(
      List(1l.mio, 10l.mio),
      List(10, 50, 500),
      List(1, 2, 4, 8, 16, 24, 32, 34, 36, 38, 40),
      List(true, false)).msg[ThroughputPingPongRequest] {
        case (n, p, par, s) =>
          ThroughputPingPongRequest(
            messagesPerPair = n,
            pipelineSize = p,
            parallelism = par,
            staticOnly = s)
      },
    testSpace = ParameterSpacePB.cross(
      10l.k to 100l.k by 30l.k,
      List(10, 500),
      List(1, 4, 8),
      List(true, false)).msg[ThroughputPingPongRequest] {
        case (n, p, par, s) =>
          ThroughputPingPongRequest(
            messagesPerPair = n,
            pipelineSize = p,
            parallelism = par,
            staticOnly = s)
      });
```

Listing 6.1: Description of the parameter space for the "Throughput Ping Pong"-benchmark.

6.1.2 The Runner

The runner is an executable written in Scala that orchestrates benchmark execution for a single testee framework at a time.

Apart from communicating with the master as described in figure 6.1, the runner also contains the parameter space descriptions for each benchmark (described in section 6.2), as well logic to handle test results.

Benchmark Parameters Each benchmark run is configured with a parameter message that is defined in protobuf. For example, the "Throughput Ping Pong"

descriptor is defined as follows, with the semantics of the fields[3] as described in section 6.2:

```
1 message ThroughputPingPongRequest {
2   uint64 messages_per_pair = 1;
3   uint64 pipeline_size = 2;
4   uint32 parallelism = 3;
5   bool static_only = 4;
6 }
```

These benchmark descriptors are instantiated by the runner with a concrete set of values, which are then sent to the master. For example, to see how the benchmark scales as the number of available CPU cores increases, we might increase the `parallelism` values, while keeping other parameters constant.

Clearly, it is impossible to explore the full parameter space defined by such a descriptor, as this would require over 2.9×10^{48} different benchmark runs for the descriptor shown above, for example.[4] In order to describe which parameters are of interest to explore, the runner's code contains a small eDSL to describe values or value ranges for each parameter, and then map an instance of the cross-product of these values to an instance of the parameter descriptor.

An example of such an assignment for the "Throughput Ping Pong"-benchmark can be seen in listing 6.1. In addition to marking the human readable `name` and the unique `symbol` identifying the benchmark, the description contains the necessary code to invoke the gRPC call for the benchmark, as well as two different parameter spaces for the benchmark. The values assigned to the `space` field are used during a full benchmarking deployment, while the values assigned to the `testSpace` are meant for relatively quickly testing the behaviour of a new implementation, in order to verify that it runs as expected. An advantage of using an eDSL — as opposed to specifying parameters in a simple configuration file, for example — is that we have access to Scala's `<start> to <end> by <inc>` syntax for creating ranges, as used in line 20, for example. This gives us a concise notation for parameter spaces, instead of having to write out every value individually.

In the example above, for the `testSpace`, we are sending between 10 000 and 100 000 messages between each pair of actors/components, queueing up either 10 or 500 messages at any given time, run either 1, 4, or 8 pairs of actors/components in parallel, and execute two different variants of the code (`staticOnly=true` or `staticOnly=false`).

Test Results The runner receives successful test results as a list of 64-bit floating point values representing the *execution time* (in milliseconds) of each individual run that was performed with the last parameters sent to the master. It stores these raw numbers in a `run-<id>/raw` folder, where the id is provided by the benchmarking

[3] Note that the values after the = in protobuf denote field ids, *not* default values.

[4] As a comparison, the current estimate for the age of the universe is less than 4.4×10^{17} s.

script; currently, a timestamp of when the run was started. The file containing the raw numbers is named for the benchmark and the parameters that were used.

In addition, the runner also produces a single file for each benchmark in a `run-<id>/summary` folder, where it stores the aggregates for each parameter set and each framework. In particular, it stores the unique symbol identifying the testee implementation, the set of parameters (in protobuf text format), and the sample mean of the execution times of the run, as well as its (corrected) sample standard deviation, RSE, and 95 % confidence interval upper and lower boundaries. These summary files are later used for plotting.

6.1.3 The Scripts

The MPP framework comes with three executable scripts for building, benchmarking, and plotting, as well as one configuration script that is used a as dependency by the others. All scripts are implemented in Scala using the *Ammonite*[5] shell for execution, since that is more readable for larger scripts than pure Bash.

Configuration The `benchmarks.sc` script contains descriptions for each benchmark implementation, describing how to run the code in the three different modes. It also contains other parameters for executing code, such as JVM configuration, for example.

Building The `build.sc` script contains information on how to build and clean the different sub-projects that contain either shared library code for a particular programming language (s. below) or one or more testee framework implementations. As different programming languages often use different build tools, this script mostly acts as wrapper to invoke each build tool in turn, before moving on to the next.

Benchmarking The `bench.sc` script is the entry point for actually running benchmarks. The parameters given to it decide which benchmarks and implementations are to be run, which mode to run in, and which parameter space to use (testing or full). It also manages the transition from benchmarking one testee implementation to the next, by starting and terminating master and client executables, as well as the runner. Furthermore, it creates the folders for benchmark results and log files for each run id, and passes information about them to the runner instances it creates.

Information on how to launch executables is loaded from the `benchmarks.sc` script, while information on which nodes to start executables on is loaded from a `nodes.conf` file, in **remote** mode.

In **fakeRemote** mode, it will also create copies of the benchmark folder in a temporary space, before launching executables there, to avoid them interfering with each other.

[5] `https://ammonite.io/`

Plotting The `plot.sc` script provides one way of quickly plotting the results produced from the benchmarks. It uses the *JavaPlot* library, which provides a wrapper around the well-known *Gnuplot* library for convenience. For each benchmark, the script contains descriptions of which plots to produce, instructions to select axes in the parameter space to plot along, and code to produce the necessary data layout from the summary format produced by the runner.

Additionally, it contains colour maps, so that testee implementations are assigned consistent colours across different benchmarks and plots.

6.1.4 Library Implementations

Although MPP is designed to work across different programming languages and frameworks, it is unavoidable that some code must be duplicated for each language and/or framework for the experiments to run correctly. Code that is only duplicated across programming languages can (and should) be abstracted into a shared library, which is then reused for different message-passing frameworks in that language.

While it is generally impossible reuse abstractions such as interfaces or traits across programming languages to enforce the implementation of certain behaviours for each language, protobuf *service* descriptions can be compiled into a large number of different programming languages. Three protobuf services form the basis of our shared code:

1) The `BenchmarkRunner` service describes the `Ready` exchange from figure 6.1, as well as the invocation for each individual benchmark.

2) The `BenchmarkMaster` service describes the `CheckIn` exchange from figure 6.1.

3) The `BenchmarkClient` service describes the exchanges for `Setup` and `Cleanup` from figure 6.1.

In addition to providing appropriate implementations of these services, each library and framework must also implement some code to execute the individual benchmark, as described below.

6.1.4.1 Per Programming Language Code

The most important task to perform for each programming language is the selection of an appropriate library providing support for protocol buffers, as well as gRPC. If no implementation exists, one must be written, which could significantly increase the amount of work required to add a new programming language to MPP. Luckily, most popular programming languages already have existing gRPC and protobuf libraries that can be reused.

```
trait Benchmark {
  type Conf;

  trait Instance {
    def setup(c: Conf): Unit;
    def prepareIteration(): Unit = {}
    def runIteration(): Unit;
    def cleanupIteration(lastIteration: Boolean, execTimeMillis: Double):
     ↪ Unit = {}
  }

  def msgToConf(msg: scalapb.GeneratedMessage): Try[Conf];
  def newInstance(): Instance;
}
```

Listing 6.2: Scala trait for local benchmarks.

Service Implementations Given a library to generate language specific service descriptors, the services must also be implemented. MPP requires two implementations of the `BenchmarkRunner` service, one for **local** mode and one for **remote** mode. The former is usually a very simple mapping of names to benchmark implementations for each framework, which can easily be abstracted to the programming language level using a factory pattern. The latter also benefits from the factory pattern, but must additionally interact with the `BenchmarkMaster` implementation, since their control flow overlaps, as was shown on figure 6.1. Finally, the `BenchmarkClient` must also be implemented appropriately. Depending on the programming language's mechanisms for loading implementations, it may also benefit from the benchmark factory. On the JVM, on the other hand, benchmark instances can also be loaded directly via reflection based on a class name specified by the `BenchmarkMaster` during the `Setup` call.

Implementations of these services must take care to handle concurrency correctly, as many gRPC libraries automatically run services on thread pools. Thus, benchmark state must be carefully synchronised, in order to avoid ending up in an invalid state. Furthermore, as networking is involved in many places in these implementations, transient failures must also be taken into account and handled appropriately.

Benchmark Abstractions It is recommended to add abstractions at the programming language level that encode benchmark configuration and setup, as well as execution and cleanup. This allows the service implementations to treat different implementations uniformly, reducing the amount of code that needs to be implemented for each message-passing framework in the language.

In the Scala shared library, for example, we define two traits shown in listings 6.2 and 6.3, one of which must be implemented by each benchmark. In those

```
trait DistributedBenchmark {
  type MasterConf;
  type ClientConf;
  type ClientData;

  trait Master {
    def setup(c: MasterConf): ClientConf;
    def prepareIteration(d: List[ClientData]): Unit = {}
    def runIteration(): Unit;
    def cleanupIteration(lastIteration: Boolean, execTimeMillis: Double):
     ↪ Unit = {}
  }

  trait Client {
    def setup(c: ClientConf): ClientData;
    def prepareIteration(): Unit = {}
    def cleanupIteration(lastIteration: Boolean): Unit = {}
  }

  def newMaster(): Master;
  def msgToMasterConf(msg: scalapb.GeneratedMessage): Try[MasterConf];

  def newClient(): Client;
  def strToClientConf(str: String): Try[ClientConf];
  def strToClientData(str: String): Try[ClientData];

  def clientConfToString(c: ClientConf): String;
  def clientDataToString(d: ClientData): String;
}
```

Listing 6.3: Scala trait for distributed benchmarks.

examples, `Benchmark.Conf` and `DistributedBenchmark.MasterConf` are instantiated to the protobuf parameter descriptor type, while `DistributedBenchmark.ClientConf` and `DistributedBenchmark.ClientData` may contain framework specific information, such as actor references, for example. The invocation of `runIteration()` is the measured part of the benchmark.

Other Shared Code Each language also needs some way to measure execution time of a run, and produce the millisecond value required by the test result message.

Furthermore, the conditions for completing a benchmark depend on some statistics of the sample data, in particular the RSE, which must be calculated.

All in all, creating a shared library for the MPP suite in a new programming language is manageable, but by no means a trivial task. The shared library for Scala,

for example, consists of around 2000 LOC, including tests.

6.1.4.2 Per Testee Framework Code

Apart from the code actually implementing the benchmarks themselves, there is luckily not too much code that needs to be written for each framework, assuming the programming language allowed for all of the abstractions described in section 6.1.4.1.

Firstly, the aforementioned benchmark factory needs to be implemented, so that benchmark implementations can be loaded by the shared library. Furthermore, the abstractions introduced for benchmarks must be implemented for each benchmark. It is also recommended to implement some kind of provider for instances of the message-passing system to be benchmarked, allowing its configuration to be located in a single place instead of being spread out across different individual benchmarks.

For networking benchmarks, serialisation and deserialisation code must be provided for all messages sent over the network. For fairness reasons, MPP insists on handwritten serialisation code, wherever possible.

Of course, some kind of main procedure for the executable must also be provided, typically just passing commandline arguments to the shared library.

6.1.4.3 Testing Code

It is strongly recommended to also implement tests for the individual benchmarks, as well as the overall communication flow between the client and the master. Whether this can be done at the programming language library level, or on a per framework basis depends on the languages used.

However, an easy mechanism for testing all implementations is to simply invoke the `BenchmarkRunner` service implementations for each benchmark with some very small parameters, and check that they return a sensible result. This can be done using a language specific unit testing framework, such as ScalaTest [108], for example.

6.2 Benchmarks

In this section we describe the benchmarks currently included in the MPP suite. The benchmarks are separated into two major groups: Those that run purely local, and those that run distributed, using either a real network or the loopback network as a zero latency network.

In order to avoid constantly writing “actor(s) or component(s)”, we will simply write “component(s)” to mean either for the rest of this section. Similarly we will write “messages” and intend it to include “events”, as well.

6.2.1 Local Benchmarks

```
1  package benches;
2
3  case object Start
4  case object Ping
5  case object Pong
6
7  class Pinger(
8    latch: CountDownLatch,
9    count: Long,
10   ponger: ActorRef) extends Actor {
11   private var countDown = count;
12
13   override def receive = {
14     case Start => {
15       ponger ! Ping;
16     }
17     case Pong => {
18       if (countDown > 0) {
19         countDown -= 1l;
20         ponger ! Ping;
21       } else {
22         latch.countDown();
23       }
24     }
25   }
26 }
27
28 class Ponger extends Actor {
29   def receive = {
30     case Ping => {
31       sender() ! Pong;
32     }
33   }
34 }
```

Listing 6.4: The Pong Pong benchmark implemented in Akka (Scala).

Table 6.1: Overview over all implemented benchmarks.

Name	Symbol	Measures	Section
Local Benchmarks			
Ping Pong	PINGPONG	message-passing overhead	6.2.1.1
Throughput Ping Pong	TPPINGPONG	message-passing throughput	6.2.1.2
Fibonacci	FIBONACCI	component creation/destruction	6.2.1.3
Chameneos	CHAMENEOS	mailbox contention	6.2.1.4
All-Pairs Shortest Path	APSP	bulk-synchronous computation	6.2.1.5
Distributed Benchmarks			
Net Ping Pong	NETPINGPONG	networking overhead	6.2.2.1
Net Throughput Ping Pong	NETTPPPINGPONG	networking throughput	6.2.2.2
Atomic Register	ATOMICREGISTER	broadcast and wait for majority	6.2.2.3
Streaming Windows	STREAMINGWINDOWS	computation spikes	6.2.2.4

6.2.1.1 Ping Pong

In this simple benchmark, two components bounce messages back and forth between each other. It is probably one of the most simple message-passing programs possible to write, and thus it is not quite clear where the origin of the benchmark lies. Savina [54] points out an old piece of Scala code at `http://www.scala-lang.org/node/54`, but in truth its probably as old as distributed programming itself.

Behaviour One component is designated as the *pinger* and the other one as the *ponger*. Both components are set up, before the run is started by a START message to the pinger. It then sends a PING message to the ponger, and waits for a response. Upon receiving the PING message, the ponger immediately responds with a PONG message. When the pinger receives the PONG, it decrements its message counter, which was initialised to the configuration parameter `uint64 number_of_messages`. If the counter is still a positive number, the process is repeated. When the counter reaches zero, the experiment is ended by the triggering of a latch, and the time is taken on the experiment thread.

An example implementation in Akka (Scala) can be seen in listing 6.4.

Measures This benchmark measures the message-passing overhead of the framework, since one component is always idle waiting for the other, and thus reducing the time taken to respond, reduces idle time, which in turn reduces the measured overall execution time. As the usage of the latch introduces some overhead, long enough sequences of message exchanges must be chosen to reduce its impact on the results.

It is important that the messages being sent back and forth are purely static and cause no allocations for every instance. Otherwise, the message-passing overhead is confounded with the allocation and deallocation (or garbage collection) overheads of the particular programming language in use.

6.2.1.2 Throughput Ping Pong

This benchmark is conceptually very similar to Ping Pong above, but this time the focus is on throughput and scaling behaviours. An example of this benchmark can be found in [82], which we used before in chapter 3.

Behaviour The behaviour is the same as in the normal Ping Pong, except as noted below.

Instead of idle waiting after the first message, the pinger now sends up to a number of messages determined by the `uint64 pipeline_size` parameter, before waiting for the first response. Afterwards each message is answered individually as before.

Additionally, instead of running a single pinger-ponger-pair, the `uint32 parallelism` parameter now controls the number of pairs to be started. Each pair interacts independently, causing no "mailbox" contention with other pairs. It may, of course, cause other contention within the implementation framework, such as scheduler contention, for example.

Finally, this benchmark comes in a second variant, where the current message index is passed both in the PING and PONG messages. The second variant is selected by setting the `bool static_only = false` parameter.

Measures This benchmark measures message-passing throughput at different levels of parallelism and pipelining depth. Many message-passing frameworks implement some kind of message batching behaviour, to optimise cache reuse and amortise component scheduling overhead. Pipelining depth can be used to find a sweet spot between latency and throughput for the particular framework.

Finally, the second variant of the benchmark with indices additionally stresses the programming languages allocator, allowing a comparison of overheads introduced by message-passing and allocation between the two variants.

6.2.1.3 Fibonacci

The Fibonacci benchmark stresses the systems ability to dynamically create and remove component instances in a recursive fashion. It was ported from Savina [54], whose authors in turn ported it from Cardoso et al. [19].

Behaviour A single top-level component is created initially by the benchmarking framework and acts as an entry point to the computation. During the *run*-phase, the "root" component is asked to compute *Fibonacci number* at index $n > 0$. If $n = 1$ or $n = 2$ it immediately responds with a result message containing 1. For $n > 2$, however, it will spawn two more instances of itself, and ask the first to calculate the Fibonacci number at $n - 1$ and the second at $n - 2$. Once both children have responded with their results, it will sum them, and respond to its parent. The benchmark run ends when the "root" component has responded to the benchmark

framework. Additionally, each component will kill itself as soon as it responds to its parent, in order to recover resources. This benchmark's only parameter is the index n, as given in protobuf by `uint32 fib_number`.

Measures This benchmark measures the testee's ability to manage large, dynamic component systems with certain interdependencies efficiently. In particular it measures component creation, and deallocation performance, as well scheduling efficiency, since the work performed by each individual component is negligible. As some components are very short lived and some are very long lived, this benchmark also somewhat challenges the host language's garbage collection mechanism, if any.

6.2.1.4 Chameneos

Kaiser and Pradat-Peyre introduced this game-like concurrency benchmark in 2003 [58]. It consists of a number of "chameneos", which are trying to meet each other, and a "mall" that tries to facilitate this by repeatedly pairing up two "chameneos". The purpose of the benchmark is to make progress despite a contended resource (the "mall").

Behaviour A single mall component and a configurable number of chameneos, given by `uint32 number_of_chameneos`, are set up as preparation. For the *run*-phase all the chameneos are started and immediately send a MEET message to the mall. The mall will store the first MEET message it receives, and wait for another. On the second one, it will send the stored message to the source of the second message and clear its storage. When a chameneo receives a MEET message it updates an internal field (its *colour*) based on the content of the message and then sends the updated value in a CHANGE message to the chameneo that originally sent that MEET message, which in turn will also update its internal field to the same value. Both partners will then immediately request another meeting from the mall.

The mall will continue its store and match alternating pattern until it has facilitated a configurable number of meetings, given by `uint64 number_of_meetings`. It will then respond with an EXIT message to every new MEET message. When a chameneo receives an EXIT message, it responds with the total number of meetings it has participated in to the mall and and then shuts itself down. The mall awaits all such MEETINGSCOUNT messages, sums the contents, and then ends the benchmark by triggering a latch.

Measures The *Chameneos* benchmark primarily measures the impact of resource contention on a single "mailbox" on overall execution time. While not completely sequential, the overall performance is still heavily dependent on the throughput of the single mall component. As the number of concurrently active chameneos increases, the behaviour of the writing side of the mall's "mailbox" becomes

Algorithm 3 Floyd-Warshall all-pairs shortest path (*adapted from [55, alg. 141]*)

```
1:  ▷ For V nodes, adj is the |V| × |V| adjacency matrix, with ∞ where no direct link
    exists, adj[i][i] = 0 for all i ∈ |V|, and the edge weight everywhere else.
2:  function ShortestPath(adj)
3:      dist := adj
4:      indices := (1, ..., |V|)
5:      for k ∈ indices do
6:          for i ∈ indices do
7:              for j ∈ indices do
8:                  curValue := dist[i][j]
9:                  newValue := dist[i][k] + dist[k][j]
10:                 if curValue > newValue then
11:                     dist[i][j] := newValue
        return dist
```

increasingly important for good overall throughput, as excessive blocking or back-off behaviours can leave the system's capacity temporarily underutilised.

If the `uint32 number_of_chameneos` parameter is increased significantly beyond the number of available (logical) cores on the host, the benchmark instead starts to test the performance of the testee's scheduler, as individual component work is negligible, but task dependencies are important and the mall needs prioritisation. "Mailbox" contention, however, can not increase beyond the number available (logical) cores.

6.2.1.5 All-Pairs Shortest Path

While the previous benchmarks have been micro-benchmarks, in the sense that they were testing the performance of a single or a small number of features, this block-distributed concurrent implementation of the Floyd-Warshall algorithm [33] (algorithm 3) is significantly more involved. It stresses the host language's allocator, memory-management (including garbage collection), and its ability to optimise random accesses in data-structures implementing adjacency matrices, in addition to challenging each individual framework's ability to schedule fairly and efficiently, and deal with potential "mailbox" contention. It was originally proposed as a benchmark by the authors of Savina [54].

Behaviour Initially the weighted adjacency matrix for a graph of size n, given by `uint32 number_of_nodes`, is randomly (but deterministically for the same parameters) generated, and a manager component is started. At initialisation time, the manager component is given a fixed size, which it may subdivide any adjacency matrix it is given into, defined by the `uint32 block_size` parameter. In the *run*-phase, the precomputed graph is sent to the manager, which must then compute the all-pairs shortest path matrix for it. Upon receiving this initial message, the manager splits

the matrix into a number of blocks as determined by the configured block size, and then assigns each block to a block worker component. As is required by the access pattern of the inner two loops of algorithm 3, each block worker is connected to the other block workers in its column and row, referred to as its *neighbours*. As soon as a block worker has all its neighbours, it will broadcast its current block state to them. When a block workers receives the state for the current phase, which is equivalent to the value k from the outer loop in algorithm 3, it will update its local block using the two inner loops of algorithm 3 using the data it has stored from its neighbours. It will then again broadcast its updated block to all neighbours. This proceeds until conceptually the outer loop completes, indicated by the completion of the phase with $k = |V|$. At this point each block worker sends their final block to the manager, which collects and reassembles all the blocks into a full distance matrix, before ending the benchmark by triggering a latch.

As an implementation note, our version uses floating point (at least 64-bit precision) weights, which support ∞ values in most languages[6], as opposed to Savina's implementation, which chose 64-bit integer (i.e., Java `long`) weights and always generated fully-connected graphs to avoid the need for ∞ values. Furthermore, to be more realistic, our approach also uses a graph implementation with generic weights in languages which can support type class abstractions for partial orders and addition (which is all that is needed for algorithm 3), such as Scala's `scala.math.Ordering` or Rust's `std::ops::Add`. This, of course, has some performance implications on the raw graph operations in languages that do not monomorphise generics efficiently during compilation, or generate efficient runtime instances via JIT compilation.

The impact of this, however, is attenuated as the number of blocks increases, as messaging overheads become the dominant performance characteristic. As this is designed as a message-passing benchmark, it is those scenarios we are mostly interested in, not the all-pair shortest path problem solving performance of the implementations.

Measures The bulk-synchronous nature of this particular algorithm puts particular challenges on the scheduler of the system to run components fairly and avoid stragglers. Frameworks that support priorities could benefit from prioritising earlier phases. Apart from that the topology is a fairly static pattern, with communication always going within a column or row. For small numbers of blocks, the host languages memory management and compute performance also has significant impact on the resulting performance, while for larger numbers of block raw messaging performance becomes increasingly dominant and "mailbox" contention may become a challenge.

[6] But not in Erlang, for example.

6.2.2 Distributed Benchmarks

Distributed benchmarks are only implemented if the testee framework provides a networking implementation, as they are designed to stress that part of the framework. For frameworks that come without a default networking implementation, these benchmarks are simply skipped.

6.2.2.1 Net Ping Pong

This benchmark behaves exactly as the Ping Pong benchmark described in section 6.2.1.1, except that now every message goes over a single Transmission Control Protocol (TCP) connection.

Measures This benchmark measures the networking and serialisation overhead of the framework.

The sequential nature of this benchmark makes it particularly sensitive to details of the underlying TCP implementation provided by the host operating system (OS) and the configuration parameters used by the testee framework for configuring its network channels.

6.2.2.2 Net Throughput Ping Pong

This benchmark behaves exactly as the Throughput Ping Pong benchmark described in section 6.2.1.2, except that now every message goes over a single TCP connection, as pingers and pongers are split between two hosts (with all the pingers on one host, and all the pongers on the other).

Measures This benchmark measures the networking and serialisation throughput of the framework. As every message goes through a single network connection, it also measures the coordination overhead of that connection, when processing both incoming and outgoing messages from multiple independent components.

Like Net Ping Pong (section 6.2.2.1) this benchmark is also very sensitive to details of the underlying TCP implementation and its configuration. Low level details, like the use of Nagel's algorithm [1], for example, can have very large impact on the performance with certain test parameters.

6.2.2.3 Atomic Register

This benchmark investigates the performance of a very simple in-memory key-value store, based on the well-known *Read-Impose Write-Consult-Majority* algorithm for an (N, N)-Atomic Register [16]. A single register version it is shown in algorithm 4, but since a key-value store usually holds a large number of keys, the benchmark implements a multi-register variant of the same algorithm instead, by keeping the state of each register separately in a map data structure.

Algorithm 4 Read-Impose Write-Consult-Majority (*adapted from [16, p. 165f]*)

Implements:

(N, N)-AtomicRegister, **instance** *nnar*.

Uses:

BestEffortBroadcast, **instance** *beb*.
PerfectPointToPointLink, **instance** *pl*.

1: **upon event** ⟨ INIT ⟩ **do**
2: $(ts, wr, val) := (0, 0, \bot)$
3: $acks := 0$
4: $writeval := \bot$
5: $rid := 0$
6: $\forall_{p \in \Pi}\ readlist[p] := \bot$
7: $readval := \bot$
8: $reading :=$ FALSE
9: **upon event** ⟨ *nnar*, READ ⟩ **do**
10: $rid := rid + 1$
11: $acks := 0$
12: $\forall_{p \in \Pi}\ readlist[p] := \bot$
13: $reading :=$ TRUE
14: **trigger** ⟨ *beb*, BROADCAST | [READ, *rid*] ⟩
15: **upon event** ⟨ *beb*, DELIVER | *p*, [READ, *r*] ⟩ **do**
16: **trigger** ⟨ *pl*, SEND | *p*, [VALUE, *r*, *ts*, *wr*, *val*] ⟩
17: **upon event** ⟨ *pl*, DELIVER | *p*, [VALUE, *r*, ts', wr', v'] ⟩ **do**
18: **if** $r = rid$ **then**
19: $readlist[p] := (ts', wr', v')$
20: **if** $|readlist| > \frac{|\Pi|}{2}$ **then**
21: $(maxts, rr, readval) :=$ HIGHEST($readlist$)
22: $\forall_{q \in \Pi}\ readlist[q] := \bot$
23: **if** *reading* **then**
24: $bcastval := readval$
25: **else**
26: $rr :=$ RANK(*self*)
27: $maxts := maxts + 1$
28: $bcastval := writeval$
29: **trigger** ⟨ *beb*, BROADCAST | [WRITE, *rid*, *maxts*, *rr*, *bcastval*] ⟩
30: **upon event** ⟨ *nnar*, WRITE | *v* ⟩ **do**
31: $rid := rid + 1$
32: $writeval := v$
33: $acks := 0$
34: $\forall_{p \in \Pi}\ readlist[p] := \bot$
35: **trigger** ⟨ *beb*, BROADCAST | [READ, *rid*] ⟩

```
36: upon event ⟨beb, DELIVER | p, [WRITE, r, ts′, wr′, v′]⟩ do
37:     if (ts′, wr′) > (ts, wr) then                    ▷ Tuple comparison.
38:         (ts, wr, val) := (ts′, wr′, v′)
39:     trigger ⟨pl, SEND | p, [ACK, r]⟩
40: upon event ⟨pl, DELIVER | p, [ACK, r]⟩ do
41:     if r = rid then
42:         acks := acks + 1
43:         if acks > N/2 then
44:             acks := 0
45:             if reading then
46:                 reading := FALSE
47:                 trigger ⟨nnar, READRETURN | readval⟩
48:             else
49:                 trigger ⟨nnar, WRITERETURN⟩
```

Behaviour The atomic register benchmark represents a replicated key-value store with a fixed replication degree, which is configured by the `uint32 partition_size` parameter. Replication groups should be of uneven number, typically, as the algorithm waits for a majority of responses before proceeding, and majorities for even numbers are larger than those for uneven numbers, causing the execution to be slower. The benchmark only runs a single replica per node, and consequently a total of 3 nodes (2 clients and 1 master) are required, in order to be able to run with replication degree of 3.

Each replica is loaded with default values for each key used in the experiment during setup. The number of keys is configured with the `uint64 number_of_keys` parameter, and each key corresponds to about 8 B of data plus internal data-structure overhead per data store map in the implementation.

During the *run*-phase each replica generates a combination of *write* and *read* requests for the subset of the keyspace it is responsible for. It will invoke operations for all keys in parallel, and wait for them to complete. At this point it informs the master, which completes the benchmark using a latch, once it has received such a notification from each replica. The ratio between *write* and *read* operations can be adjusted using the `float read_workload` and `float write_workload` parameters, the sum of which must be 1.0 (corresponding to 100 %).

As part of the underlying *Read-Impose Write-Consult-Majority* algorithm, each request causes at least one message round-trip to a majority of replicas. A *write* operation always causes a second round-trip, where the new value is written using the timestamp determined in the first round-trip, while a *read* operation only causes a second round-trip when there are concurrent writes. Thus a correct implementation should have higher performance as the ratio of operation shifts towards more reads.

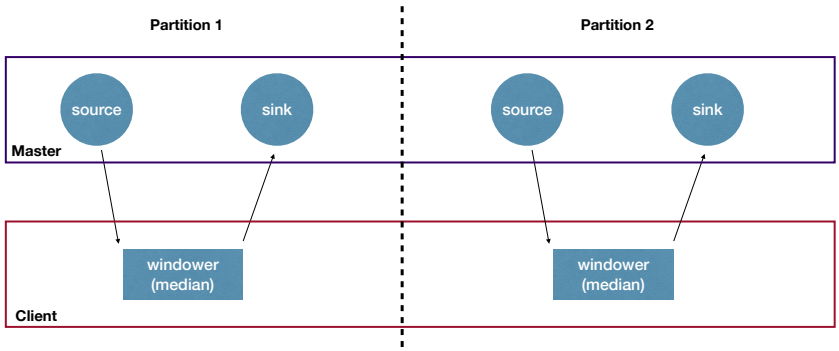


Figure 6.2: Overview of the data flow in the Streaming Windows benchmark.

Measures This benchmark measures algorithm performance on the extremely common communication pattern, where a node sends a request to a group of nodes and must wait for responses from a majority before being able to proceed. Testee frameworks with low-overhead network implementation tend to perform well in this benchmark, as (majority) response latency is of crucial importance for progress.

As the size of the key-value store increases, the performance of the map data structures provided by the underlying language gains in importance on the overall performance.

6.2.2.4 Streaming Windows

This benchmark is representative of a stream processing workload where a continuous stream of small records is grouped into fixed-length, non-overlapping (*tumbling*) windows, and then a computationally intensive reduction operation, in this case finding the *median* of all the record values, is performed, before emitting a result record for the window. In order to show core scaling, the benchmark can be run with multiple independent parallel partitions. An overview of the data flow with two partitions can be seen in figure 6.2.

In order not to overload the system during the expensive reduction operation, the benchmark also uses a very simple flow control mechanism, which informs source components how many records they are allowed to send downstream before waiting for an update from the *windower* component that grants them a new quota.

Behaviour During the *setup*-phase one instance of *source*, *windower*, and *sink* components is created and connected for each partition. The number of partitions can be controlled with the `uint32 number_of_partitions` parameter.

When the *run*-phase begins, the pipeline is started in reverse order; that is, from the sinks. A started sink informs its upstream windower that it is ready to begin, which in turn will send an initial quota message its upstream source. The size of

the quota is controlled by the `uint64 batch_size` parameter. While a source has available quota, it will send records with a (logical) timestamp, its partition id, and a randomly generated 64-bit integer value to the windower. The windower stores all received values in the buffer for the current window. It will also send a new quota message upstream, once it has processed half the records granted by the previous quota. This early reset avoids the pipeline running dry during normal operations, without the excessive overhead of acknowledging every single processed message.

Once all records within the current window boundaries have been collected, the windower will run the reduce function on the current buffer. Since the reduce function calculates the median of the window, that means the current window buffer will be sorted and then the middle value (or the average of the two middle values) will be taken and sent to the sink in a record containing the window's timestamp and partition id, in addition to the 64-bit floating point median. The window buffer and window boundaries are reset in preparation for the next window. Window size is specified in a logical time duration and given by the `string window_size` parameter. Since protobuf does not support a native duration type, a string is used instead, where a valid value is `"10ms"`, for example. The logical timestamp considers 1 step to correspond to 1 ms for this purpose. This in turn also means that window size in logical time corresponds directly to window storage size, as every stored value is 8 B, and one is generated every millisecond. Thus a window of 100 ms requires 800 B of storage.

In order to be able to test the behaviour of large windows without having to wait for the network link to be able to fill them up, an additional parameter `uint64 window_size_amplification` can be used to grow windows at a higher rate compared to the message flow. For an amplification factor of a, every message that arrives at the windower is added a-times, instead of just once, to the window buffer. Thus with $a = 10$ a window of 100 ms requires 8 kB of storage, instead of just 800 B.

Finally, the parameter `uint64 number_of_windows` specifies how many window result records each sink component must receive, before it triggers the latch that completes the run.

In order to ensure that no messages remain in transit before the next run starts, a flush protocol is executed during the *cleanup*-phase, and, of course, window boundaries and logical time are reset.

Measures With its characteristic spike in latency whenever a large window is being reduced, this benchmark stresses both the networking performance of the testee framework and the compute performance of the host language. Additionally, the growth of the window buffer, and the sudden deallocation as a window is reduced taxes the memory allocation system, as well as garbage collection mechanism, if any is present.

On the other hand, network latency is not a major concern, unless the batch size parameter is picked very small.

6.3 Frameworks and Languages

In this section we will give a quick overview of each message-passing framework and programming language we have already implemented the MPP suite for.

6.3.1 Scala & Java

We already used some examples from the Scala shared library in the previous sections, which provides a number of very nicely typed abstractions.

For convenience, we are also using the Scala library for Java frameworks, as it is easy to invoke Java code from Scala, and it saves the time of writing another implementation. Since the MPP framework code is not performance relevant, this has no impact on the benchmark results.

All Scala/Java implementations are run on the JVM, as opposed to being compiled to native code, for example. Thus all these implementations make use of garbage collection, which causes some overhead often only seen at higher load levels and in longer experiments.

Java Version All benchmarks are designed to run on a version `1.8` JVM, and must thus limit themselves to features already available there. The reason for this limitation is that, at the time of writing, Scala works most reliably on `1.8`.

Scala Version All Scala benchmarks are implemented for Scala `2.12`.

It should be noted that all Scala benchmarks use native Scala collections in their implementations, and since Scala `2.13` has had a major overhaul of the collection library, the more compute intensive benchmarks might see some notable improvements, if they were run on Scala `2.13` instead of Scala `2.12`. However, not all frameworks were available for Scala Scala `2.13` already, and thus for fairness' sake we decided to run everything on Scala `2.12`.

6.3.1.1 Kompics Java

Kompics Java is the reference implementation of the Kompics component model, and has been described in detail in section 2.2.2. Kompics Java also provides a networking abstraction, based on the well-known Netty[7] networking library for Java. It adds a custom serialisation library on top, as well as automated channel management.

We have been using version `1.1.0` for the benchmarks.

6.3.1.2 Kompics Scala

As described in chapter 3, Kompics Scala is an implementation of the Kompics component model, which supports pattern matching. It uses the same networking

[7] `https://netty.io/`

library as Kompics Java; however, different message types may incur different serialisation overhead.

Like in chapter 3, we have been using both different variants of Kompics Scala for the benchmarks: For the stable release version designed for large patterns — labelled as `KompicsScala1x` — we have been using version `1.1.0` and for the experimental version targeted at the more common smaller patterns — labelled as `KompicsScala2x` — we have been using `2.0.0-SNAPSHOT`.

6.3.1.3 Akka & Akka Typed

Akka is an implementation of the Actor model, and has been described in section 2.1.1.2. It also provides a networking extension based on Netty, like Kompics Java. However, unlike Kompics Java, the abstraction is integrated into the framework transparently, allowing the distribution of remote actor references, for example.

In addition to the well-known version of Akka, we have also implemented the benchmarks for the newer *Akka Typed* model, which uses typed actor references and does not use implicit sender references anymore.

We have been using version `2.5.25` for the benchmarks.

6.3.2 Rust

Rust is a natively compiled language designed by Mozilla [102]. It does not feature garbage collection, but instead comes with a powerful compiler that infers unused data from *lifetimes* and automatically inserts deallocation code.

All experiments have been run on and written for the `1.40.0-nightly` build, as many of the tested frameworks are still in early development and require experimental features only available in nightly builds.

6.3.2.1 Actix

Actix[8] is an implementation of the (typed) Actor model, that has been changing focus recently to be more of a web framework, than a full-blown actor framework. It is however very popular in that area, having topped out some of the TechEmpower *Web Framework Benchmarks*[9] since 2018.

Thus, while Actix comes with a convenient way to define networking protocols, it does not actually provide networking library that allows sending of messages between actors in a convenient way out of the box. It has thus been excluded from distributed tests, as any implementation of a networking library would reflect on the author of this dissertation more than on Actix' library capabilities.

Furthermore, Actix is very limited in its ability to scale actors over thread pools. It either allows multiple *different* actors to run on a *single thread*, or multiple *identical* actors to run on a *thread pool* in a load-balancing fashion. Since many benchmarks in

[8] `https://actix.rs/`

[9] https://www.techempower.com/benchmarks

the MPP suite rely on scaling different, or at least individually addressable, actors over many cores, Actix is somewhat limited in this regard.

We have been using version `0.8.3` for the benchmarks.

6.3.2.2 Riker

Riker[10] is another implementation of the (typed) Actor model, which very closely mirrors Akka in its goals. It is still in very early development, however, and like Actix it does not not provide a networking library yet, excluding it from distributed tests.

We have been using version `0.3.2` for the benchmarks.

6.3.3 Erlang

Erlang is a both a programming language and an implementation of the Actor model, and was described in section 2.1.1.1. Erlang's BEAM VM also comes with its own networking support, allowing actors to seamlessly communicate and even move between different host machines.

As networking is build into Erlang and available gRPC implementations are not well maintained, Erlang is the only framework that uses message-passing for communication between the master and clients, and only uses gRPC for communication with the runner.

Erlang is quite different from the other host languages, in that it is purely functional, dynamically typed, and uses variable size integers and floats, for example. This has significant impact on the benchmarks that are more compute heavy.

We have been using `Erlang/OTP 22` for the benchmarks.

6.4 Results

Using the MPP framework and the testee frameworks described in the previous sections, we have run a number of experiments to explore the state of the art performance space in message-passing systems.

6.4.1 Experiment Setup

Due to the large performance difference between some frameworks, we have performed two different runs of the benchmarks: A larger experiment with the full parameter space (see section 6.1.2) was run on Amazon AWS, but it excluded some of the slower frameworks from the more intensive benchmarks, which — based on extrapolation from previous test runs — would have taken multiple days each to complete. In order to evaluate all implemented frameworks, a smaller test run was

[10] `https://riker.rs/`

recorded using the testing parameter space (see section 6.1.2) on an office desktop machine.

In both setups, everything was run in `fakeRemote` mode (see section 6.1.1), to avoid any impact of network conditions, which the authors had no control over.

6.4.1.1 Amazon AWS Details

This setup used a 2019 `c5.metal` machine, which is a dedicated bare metal host without any virtualisation. This configuration comes with 192 GB of memory, and 2 Intel® Xeon® Platinum 8124M CPUs with 18 cores each, for a total 36 physical cores and 72 logical cores.

The host was running Ubuntu `18.04.3 LTS` with a `4.15.0` kernel for the `x86_64` architecture.

Java All JVMs in this setup were Oracle's 64-bit Java HotSpot™ Runtime Environment version `1.8.0_231` and were run with the following options: `-Xms1G -Xmx32G -XX:+UseG1GC -XX:+HeapDumpOnOutOfMemoryError`, allowing them to grow their heap up to 32 GB.

Erlang All experiments were run on Erlang/OTP version 22 with SMP enabled.

6.4.1.2 Desktop Details

In the office desktop setup, we used an on-premise standalone machine, with 16 GB of memory, and an Intel® Core™ i7-4770 CPU with 4 physical cores and 8 logical cores.

The host was running Arch Linux with a `5.3.7` kernel for `x86_64` architecture.

Java All JVMs in this setup were Oracle's 64-bit Java HotSpot™ Runtime Environment version `1.8.0_221` and were run with the following options: `-Xms1G -Xmx8G -XX:+UseG1GC -XX:+HeapDumpOnOutOfMemoryError`, allowing them to grow their heap up to 8 GB.

Erlang All experiments were run on Erlang/OTP version 22 with SMP enabled.

6.4.2 Raw Data

In the following sections, we will describe a selection of the results from the two benchmark runs described above. We must limit ourselves to a selection, as the total result set is too large to be described within these pages. However, we make all the raw data, as well as as an interactive, digital rendering of all the graphs available online as part of the benchmark suite repository at:
`https://kompics.github.io/kompicsbenches`

6.4.3 Local Benchmarks

6.4.3.1 Ping Pong

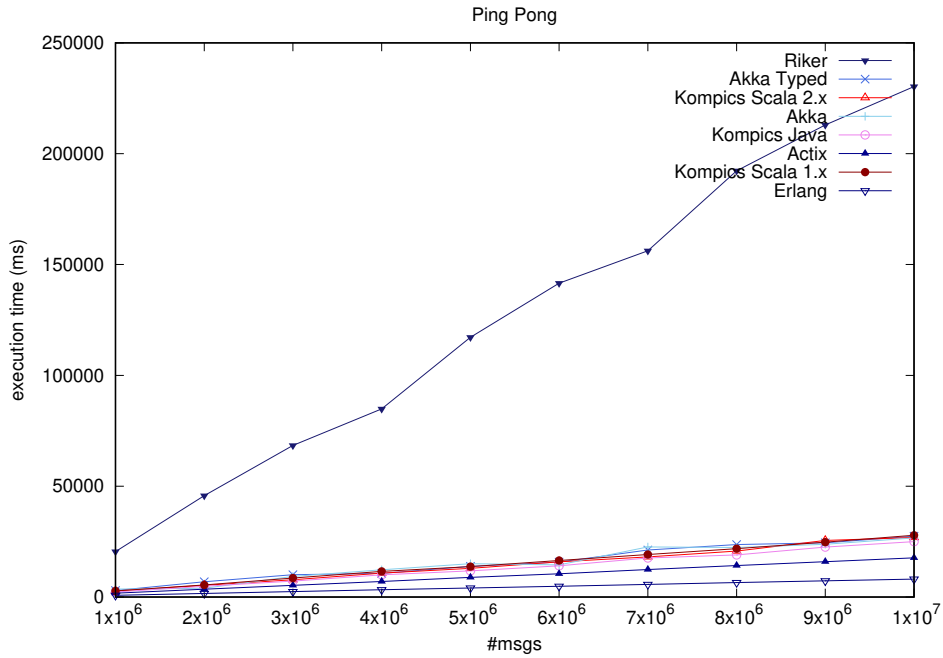
Figure 6.3 shows the results of the Ping Pong benchmark (section 6.2.1.1) for both setups. It can easily be seen that the results are fairly consistent between the two setups, and that Riker is the clear loser by a wide margin. The relative performance of the other implementations is a bit easier to see in figure 6.3b, but they mostly hold true in the AWS setup as well, with the exception of Akka (Typed). In particular it can be seen that the three Kompics implementations make up the next slowest group, with Kompics Java marginally faster than the two Scala implementations. The difference between them is only between 0.5 s and 2 s, corresponding to between 2 % and 8 %, at the largest parameter on AWS (10 million messages), for example. On AWS Akka and Akka Typed are also in that group, while in the Desktop setup they are within 20 ms of Erlang in the fastest group. It is most probable that the very short run times in the Desktop setup do not cause sufficient memory-pressure to trigger a garbage collection on the JVM, accounting for this difference in performance. Actix is in a category by itself, faster than Riker and the Kompics implementations, but slower than Erlang, the fastest implementation overall, by up to 10 s (corresponding to 54 % of its total execution time).

6.4.3.2 Throughput Ping Pong

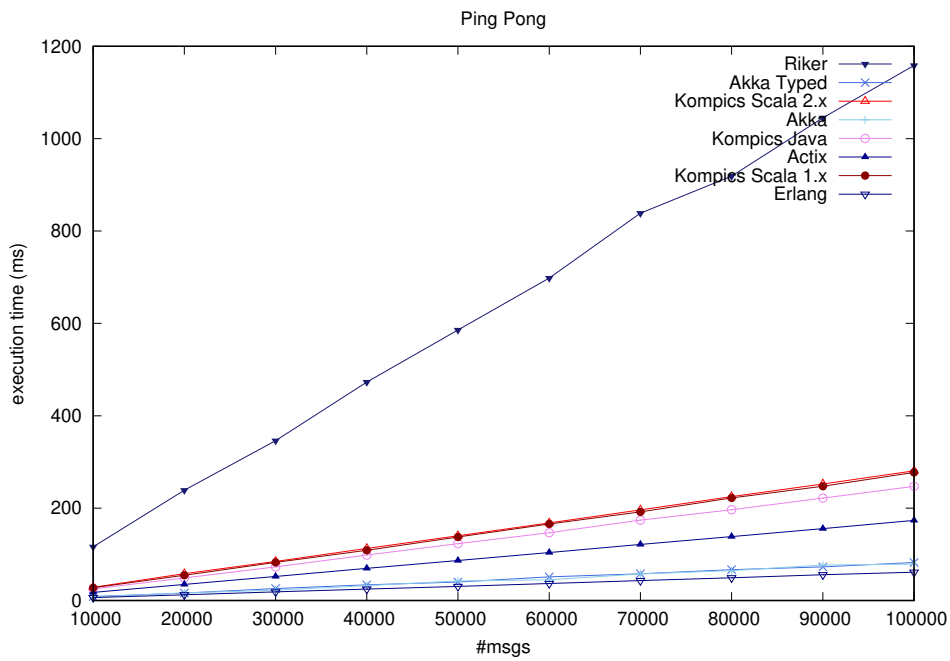
While Erlang was came out on top of the Ping Pong benchmarks, which tests primarily framework overhead, it struggles somewhat in the Throughput Ping Pong benchmark (section 6.2.1.2), as seen in figures 6.4 and 6.5. In all experiments Erlang shows a strong start where parallelism is low — the setup that is most similar to the Ping Pong benchmark — but quickly begins to fall behind as the parallelism increases. This is, of course, most pronounced on the large multi-core machine on AWS, as seen in figure 6.4, and the effect appears to be enhanced when the pipelining depth is relatively low, as is the case in figures 6.4a and 6.4c. The influence of pipelining depth does not appear to be relevant at low parallelism, and thus the latter effect is not observed at all in figure 6.5.

At the other extreme we find Riker — again — which also struggles in particular with low pipelining depth, as seen in figure 6.5. Additionally, it barely shows any performance gains from parallelism beyond 4 pairs. These issues, combined with its low overall performance already seen in the previous experiment, forced us to abort its run on AWS, as extrapolation suggested that the experiment would not finish within the month. For this reason the Riker data is incomplete in figures 6.4a and 6.4c, and completely absent in figures 6.4b and 6.4d.

Another issue was the Actix implementation, with its completely single-threaded runtime. While it is generally not a slow framework, as opposed to Riker, the lack of SMP meant that "scaling" it over 36 cores would be a rather meaningless exercise, and it was thus also excluded from AWS runs. However, figure 6.5 shows that for a

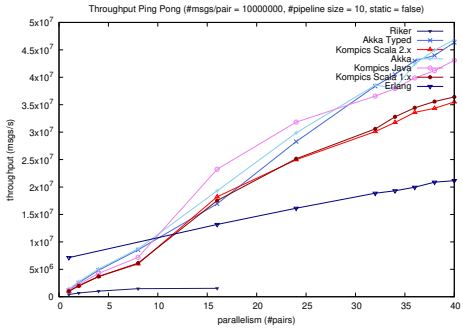


(a) The AWS setup.

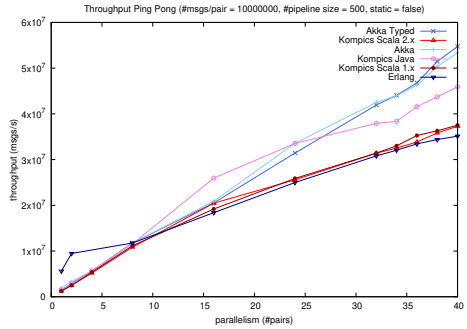


(b) The Desktop setup.

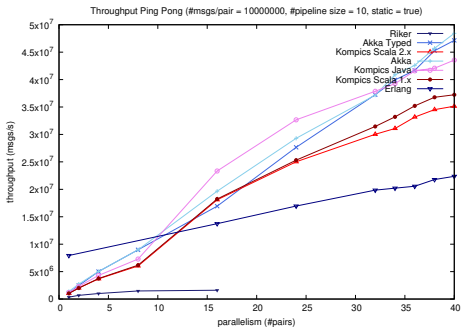
Figure 6.3: Results of the Ping Pong benchmark. Results show total execution time in milliseconds, thus lower is better.



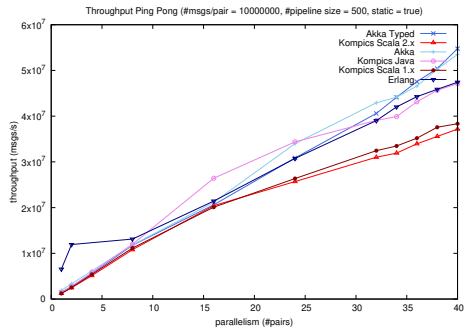
(a) Low pipelining with allocations.



(b) Deep pipelining with allocations.

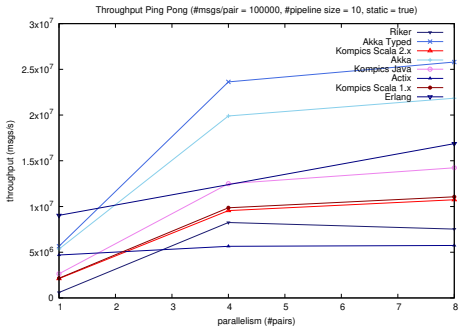


(c) Low pipelining without allocations.

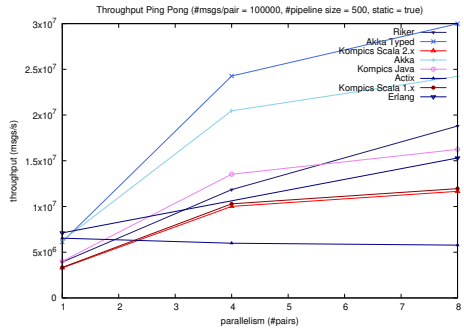


(d) Deep pipelining without allocations.

Figure 6.4: Results of the Throughput Ping Pong benchmark on AWS. Results show the average throughput of the run in messages per second, thus higher is better.



(a) Low pipelining without allocations.



(b) Deep pipelining without allocations.

Figure 6.5: Results of the Throughput Ping Pong benchmark on the Desktop. Results show the average throughput of the run in messages per second, thus higher is better.

single ping-pong pair it performs competitively with Akka and Akka Typed, and with a 500 messages deep pipelining even comparable to Erlang (see figure 6.5b).

Apart from these particularities, Akka and Akka Typed, and the three Kompics implementations perform relatively similar in all setups. There are a few noteworthy patterns to observe, however:

1) Kompics Java always outperforms the two Kompics Scala implementations. Its simpler event matching logic (see chapter 3) really shines in this particular benchmark.

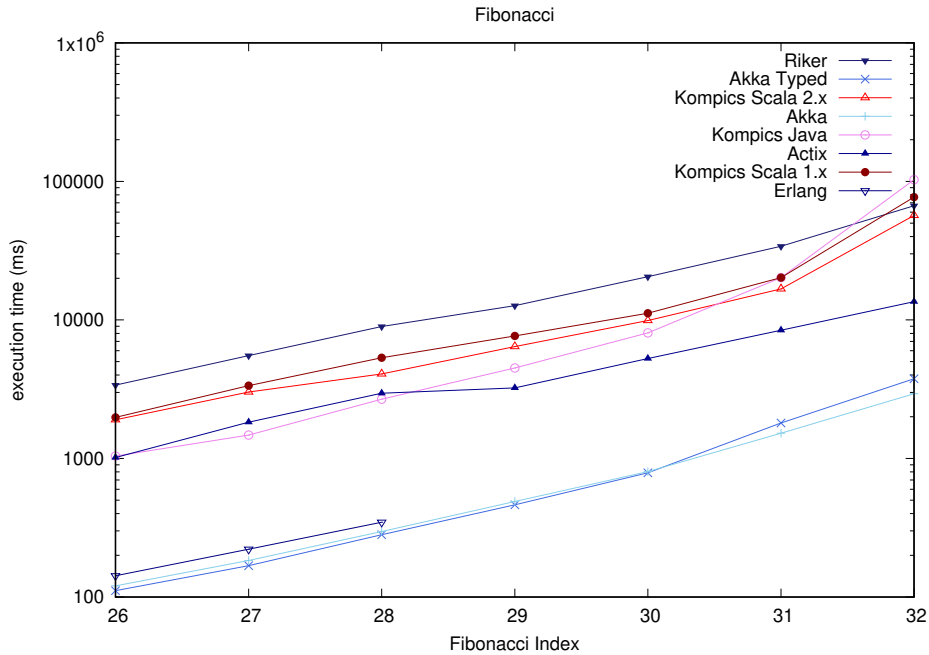
2) Akka Typed outperforms Akka in the Desktop setup, but on AWS their relationship is not as clear cut.

3) While Akka and Akka Typed tend to outperform Kompics, the AWS experiment shows that for parallelism values between 10 and 25, sometimes up to 30, Kompics Java in fact outperforms Akka. This is a very curious effect, considering that they use the same thread pool implementation, and this experiment has no particular contention effects on the mailboxes (or port queues in case of Kompics). The effect is both consistent across all parameter configurations, and statistically significant for most of them, depending on the desired significance level. It does appear to be somewhat less pronounced for shorter message sequences, however, suggesting different garbage collection timing as a possible cause.

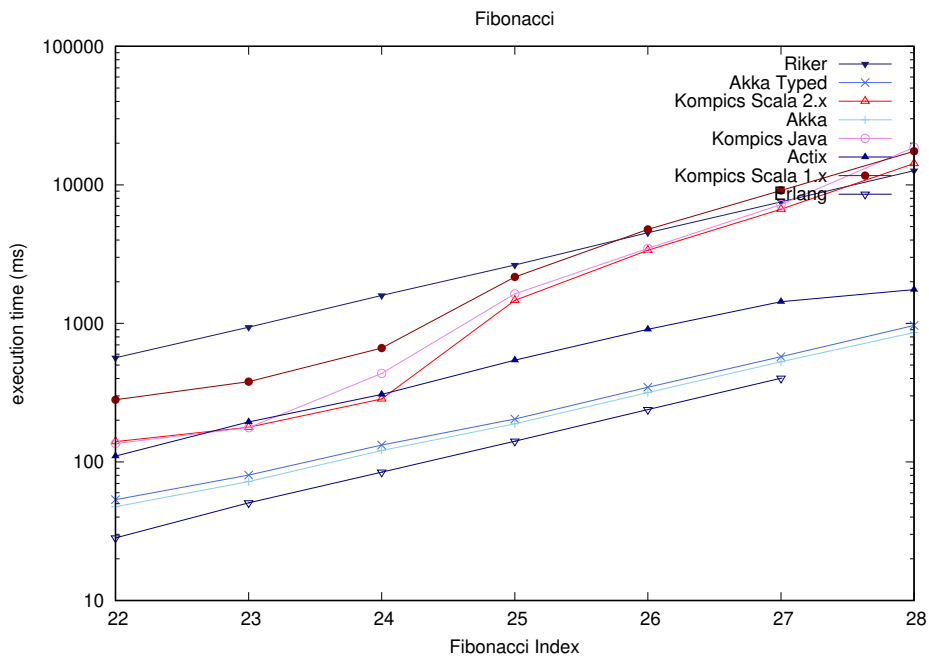
6.4.3.3 Fibonacci

The results for the Fibonacci benchmark for component creation overhead (section 6.2.1.3) are shown in figure 6.6. The results are relatively consistent between the AWS and the Desktop setup, and for the most part frameworks follow a clear exponential curve (note the logarithmic scale on the y-axis).

Notable exceptions from the latter observation include the Kompics implementations, which show two clear jumps in execution time: One between fibonacci index 24 and 25 on the Desktop setup, figure 6.6b, and another one between fibonacci index 31 and 32 on AWS in figure 6.6a. However, the fact both jumps coincide with the point where Riker becomes faster than Kompics in either setup suggests that they are probably the same effect simply shifted by the number of cores available in the particular setup. It is not completely clear what causes this effect, but it is likely related to the way Kompics handles stopping of components. In particular, Kompics is very strict in its lifecycle model and does not allow duplicate lifecycle events, such as receiving two STOP or KILL events in a row. This strictness, combined with the hierarchical nature of the benchmark and the fact that lifecycle events affect whole component subtrees in Kompics, requires an additional waiting period in the implementation of the benchmark, where a parent component can not kill itself before both of its children have finished shutting down, to avoid accidentally sending a second KILL event to them, which would cause a fault. It is likely that the



(a) The AWS setup.



(b) The Desktop setup.

Figure 6.6: Results of the Fibonacci benchmark. Results show total execution time in milliseconds, thus lower is better.

effect of this waiting becomes apparent at higher load levels and larger tree sizes, as the runtime spends more and more time with deallocation and rescheduling components just waiting to be shut down.

Another performance jump, but in the opposite direction, occurs with Actix between fibonacci index 27 and 29, depending on the setup. Again it is not quite clear what causes this effect, but we hypothesise that the system's memory allocator may have optimised for the particular size required by the Actix actors at this point. What is more important to note, however, is how competitively Actix performs in general in this benchmark, considering that it is running on a single thread, while all other frameworks use as many threads as there are cores available (up to 72 in the AWS setup). And yet for the vast majority of tested fibonacci indices, only Akka (Typed) and Erlang outperform Actix.

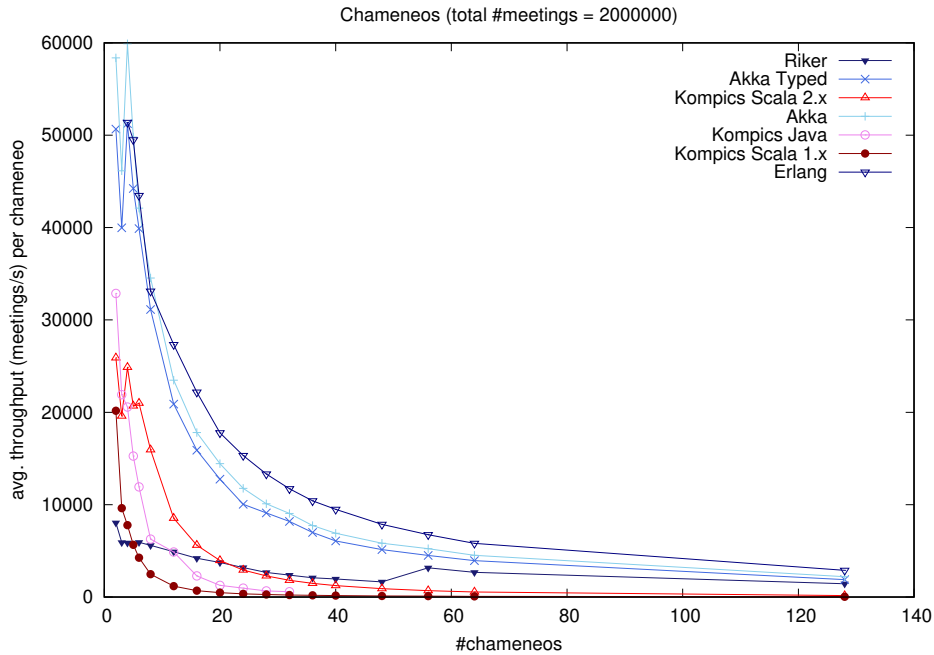
In this top performing group of the Akkas and Erlang, we can see a clear difference between the AWS setup, where the Akkas dominate, and the Desktop machine, where Erlang comes in ahead. However, it is also very clear from the sheer number of missing data points, that Erlang performs very inconsistently, especially at higher fibonacci indices. In fact, in our experiments it never converged to a RSE below 10 % within 100 runs for indices over 28, and in the Desktop setup not even for 28 itself.

6.4.3.4 Chameneos

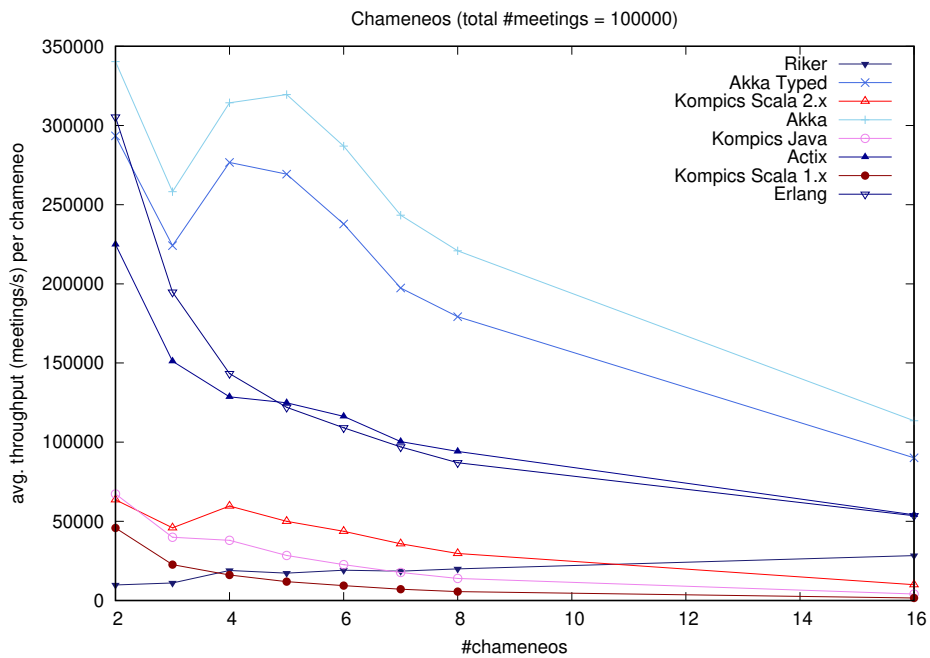
The Chameneos benchmark (section 6.2.1.4) is designed to test mailbox contention. However, it also requires any chameneo (worker component) to be able to communicate one-to-one with any other. That makes it a very tough benchmark for Kompics, where this particular communication is extremely expensive, and consequently all the Kompics implementations perform very poorly on this benchmark, especially as the number of chameneos increases. In fact, their performance is so bad that somewhere between 16 and 24 chameneos, depending on the setup, even Riker overtakes them. This can been seen very clearly in figure 6.7, which shows meeting throughput (meetings per second) normalised by the number of chameneos. In this way of plotting a hypothetical "ideal" framework would show no performance degradation whatsoever until there are more chameneos trying to meet than there are CPU cores available.

While the Akkas almost manage to fulfil this property in the Desktop setup, figure 6.7b, where the total number of physical cores is only 4, it is very clear in figure 6.7a that with the larger number of cores available on the AWS machine no framework comes even close to this ideal. Even for Erlang and the Akkas, which perform very well on this benchmark, performance drops sharply after about 8 chameneos. Erlang, however, does appear to be dealing better with the larger number of cores, compared to a somewhat weaker start below 6 chameneos.

A particular quirk of this benchmark is that at 3 chameneos many benchmarks show a bump in the performance. This happens because the third chameneo has no one to pair with while the first chameneo is still pairing with the second one, and



(a) The AWS setup.



(b) The Desktop setup.

Figure 6.7: Results of the Chameneos benchmark. Results show throughput in number of meetings per second per chameneo, thus higher is better.

thus a wait time occurs, which reduces overall performance. Curiously, not all testee frameworks display this behaviour in the results, indicating that their performance issues stem from something else than the throughput of the mall component. In particular, Erlang, Actix, Kompics Scala `1.x`, and Riker do not show the slightest hint of a bump in either of the two experiments.

For Actix, this benchmark in fact does not test mailbox contention at all, since its implementation is single-threaded, and thus Actix has been excluded from the AWS run. It is provided in the Desktop setup for reference only, where again it performs fairly well compared to multi-threaded implementations, which can not leverage their SMP advantage sufficiently, due to the limiting factor of the mall actor.

Riker once again proves to be an outlier in that it is not only very slow, but also the only framework that actually increases its throughput as more and more chameneos are added. This suggest that, like Actix, it does not suffer mailbox contention at all, but instead bottlenecks on something else.

6.4.3.5 All-Pairs Shortest Path

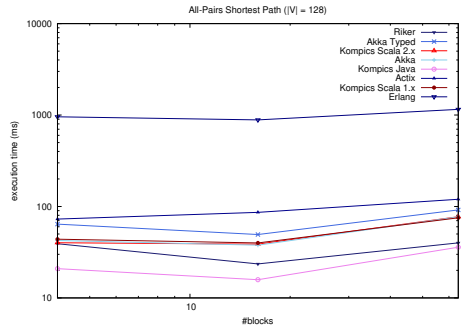
The results of the All-Pairs Shortest Path benchmark are shown in figure 6.8, and show almost a complete inversion of the previous results, with Erlang performing very poorly, due its challenges with updates in random-access data structures, and Kompics Java and Riker consistently performing best.

Looking at the results of figures 6.8a and 6.8b, which show in more detail the changes in performance with increasing number of blocks for a constant graph size, we make the following additional observation: The Java implementation, in fact, performs best with large block sizes (i.e. a small number of blocks), indicating that it outperforms the Rust-based Riker implementation on compute, not on messaging efficiency. As the total number of blocks grows, and the benchmark becomes more dependent on message-passing efficiency, Riker is more and more on par with Kompics Java.

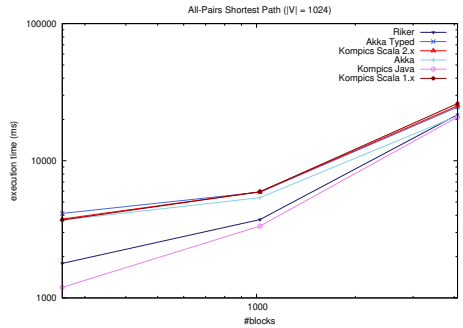
This is certainly surprising, considering the JVM's generally poor reputation for compute and memory intensive applications, and shows that the Rust compiler still has a long way to go in producing efficient code for nested loops with cross iteration access patterns, like this benchmark's `weight[i][j] = weight[i][k] + weight[k][j];` (where the loop nesting order is `k, i, j`, see algorithm 3).

Once again Actix, the other Rust implementation, had to be excluded from runs with larger numbers of nodes, due to its failure to scale over cores, which results in the second worst performance even on small graphs, as seen in figure 6.8a.

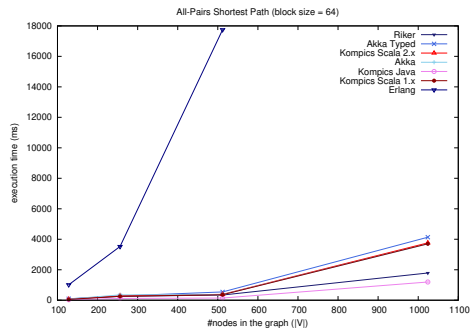
As predicted in section 6.2.1.5, all Scala implementation perform comparatively poorly on this benchmark, because their graph-related code is generic on the type of the weights and the JVM's JIT compiler does not appear to successfully monomorphise and inline the type classes used for addition and comparison of the weights in the innermost part of the Floyd-Warshall algorithm, which the Rust compiler clearly does.



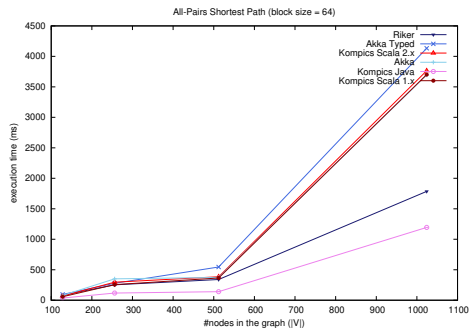
(a) On the Desktop with 128 nodes.



(b) On AWS with 1024 nodes.



(c) On AWS with 64 nodes per bock.



(d) Same as (c) without Erlang.

Figure 6.8: Results of the All-Pairs Shortest Path benchmark. Results show the total execution time of the run, thus lower is better.

In figure 6.8b we also observe that as the number of blocks increases, the *relative* differences between the frameworks shrink, as the task becomes more message-passing dominated. This shows clearly, that much of the performance of a good all-pairs shortest path implementation really depends on the underlying language and libraries used, and not so much on the message-passing performance, as increasing the number of blocks beyond the available cores is not advantageous anyway.

6.4.4 Distributed Benchmarks

Until this point, no networking code was included in any of the presented experiments. In this section, we look at the results of the distributed benchmarks in order to evaluate how well the frameworks' network libraries deal with different workloads.

As has been mentioned in section 6.4.1 already, all experiments are run over the local loopback network device, to avoid interference of network effects beyond the

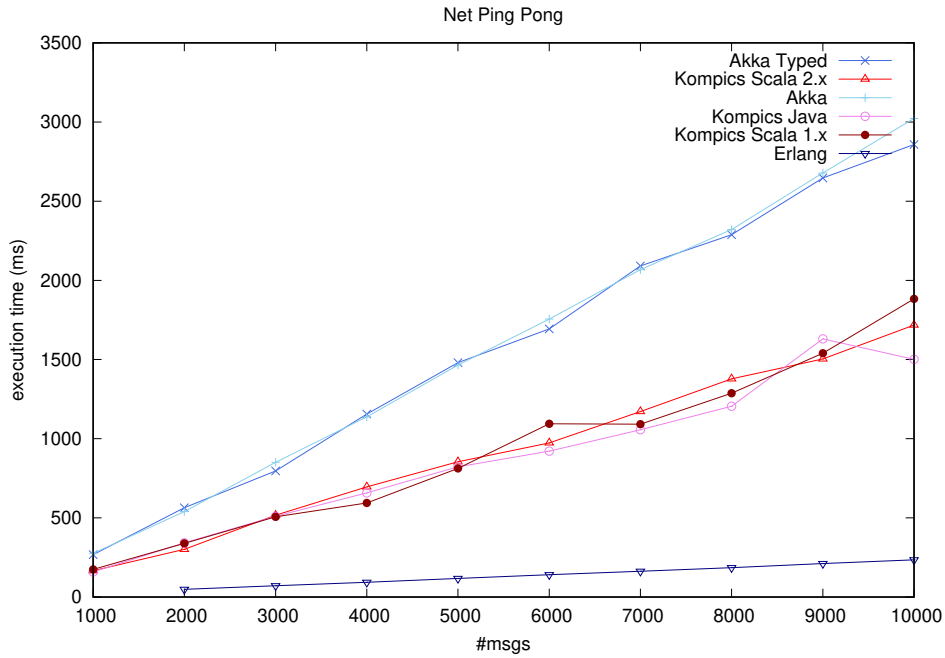


Figure 6.9: Results of the Net Ping Pong benchmark on AWS. Results show total execution time in milliseconds, thus lower is better.

control of the authors.

Since Actix and Riker do not have a network library yet, they do not appear in any of these experiments.

6.4.4.1 Net Ping Pong

Looking at figure 6.9, which shows the results for the Net Ping Pong benchmark (section 6.2.2.1) on AWS, it quickly becomes apparent that Erlang is in a class by itself when it comes to network overhead. At 10 000 messages, Erlang is, on average, 1266 ms faster than then next fastest framework, Kompics Java, which constitutes over 84 % of its total run time. Or in other words, Erlang is about 5 × faster than Kompics Java. For Akka the difference is even larger at 2787 ms on average, or almost 13 ×.

Figure 6.9 also clearly shows, that local performance of the frameworks has very little influence on their network overhead, as frameworks that use the same networking library also cluster together tightly. In fact the observed differences between Akka and Akka Typed, as well between Kompics Java, Scala `1.x`, and `2.x`, are not statistically significant.

It is curious that there is such a large difference between the Kompics implementations and the Akkas, considering that both of them rely on the Netty library as their underlying networking provider. This suggests that perhaps their different approaches to serialisation are the cause of the observed behaviours.

6.4.4.2 Net Throughput Ping Pong

The results of the Net Ping Pong benchmark from the previous section are also supported by this throughput variant (section 6.2.2.2), at least with low parallelism, as seen in figure 6.10. The only exception to this rule is Akka Typed, which performs more similar to Kompics Scala `1.x` with 1000 messages deep pipelining.

However, as the parallelism grows, the Kompics implementations slowly begin to fall behind the Akkas, due to the lack of direct addressing of components in Kompics. In fact, the best Kompics implementation quickly becomes Kompics Scala `2.x`, because its cheaper pattern matching (see chapter 3), combined with the ability to avoid scheduling components for messages they are not meant to receive, give it an edge at higher parallelism values.

The differences between the two Akka implementations are also significant in this benchmark, and figure 6.10 clearly shows that Akka Typed is superior to Akka when it comes to networking efficiency, in particular at higher levels of parallelism.

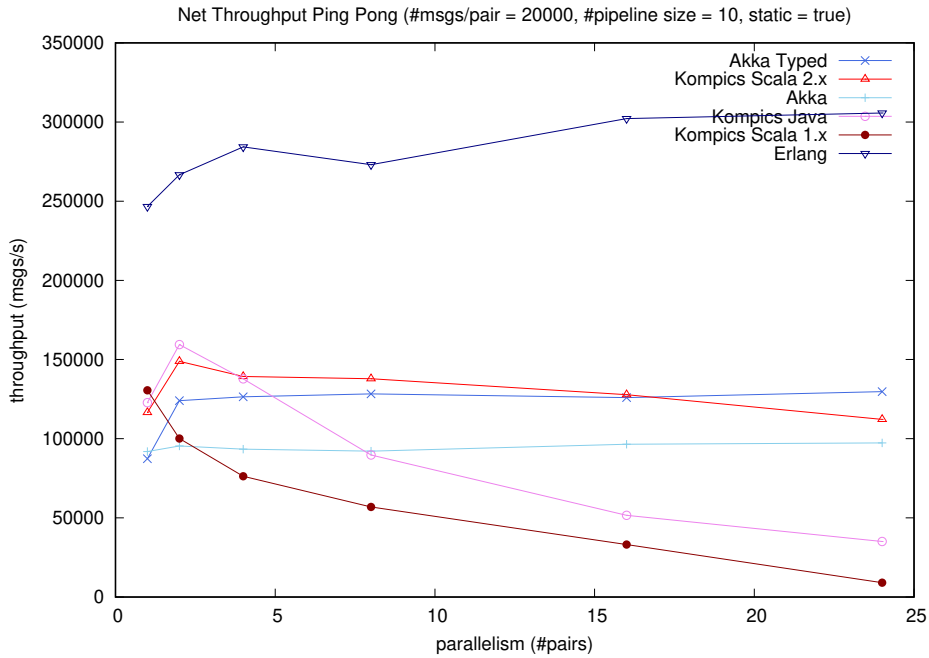
Nevertheless, Erlang still shows around double the throughput of Akka Typed at the maximum parallelism, whether with low or deep pipelining.

6.4.4.3 Atomic Register

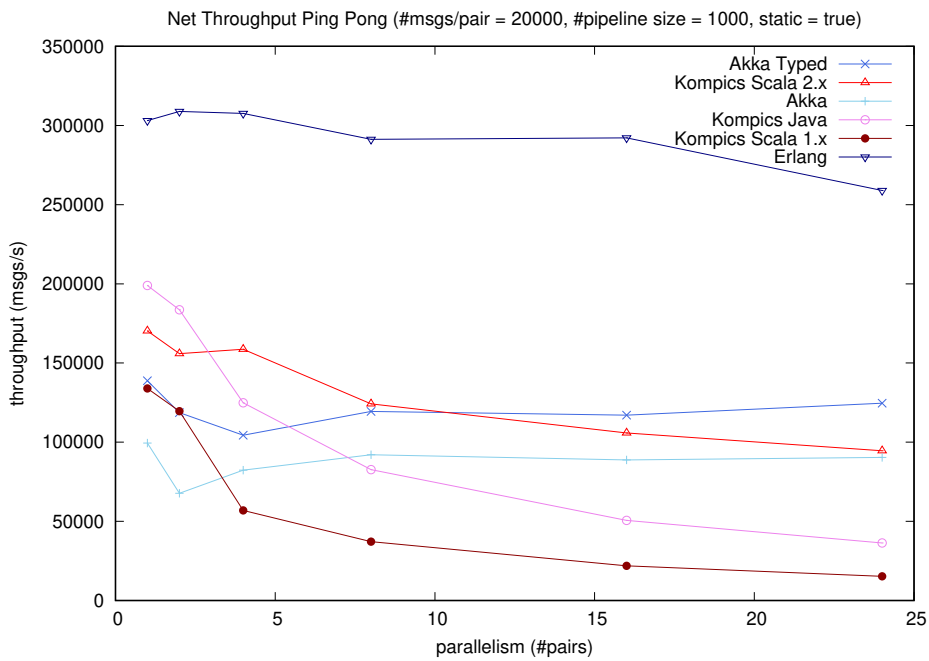
Figures 6.12 and 6.11 show the results of the Atomic Register benchmark on AWS and the Desktop setup respectively.

With the small numbers of keys used in the Desktop setup, the results are very clear cut and almost perfectly mirror the results of the previous section at low parallelism and deep pipelining. This is very reasonable, of course, as they do almost the same thing, except for an additional very quick lookup into a small hash map. This holds true both with the message-heavier parameters in figure 6.11a and in the read-heavy configuration, figure 6.11b, which can often avoid the second round-trip to a majority of replicas. Thus, on the Desktop, Erlang comes in with almost $3\times$ the performance of Kompics Java, which is the second fastest implementation here.

The situation, however, looks quite different in the AWS setup, where the parameters used result in much larger sizes of the local hash map stores, and also a larger number of operations overall, as seen in figure 6.12. These conditions allow the faster hash map implementations on the JVM to approach and sometimes even overtake Erlang, with its superior networking performance. In particular Kompics Java and Kompics Scala `2.x` perform very well in the scenarios with seven nodes, as seen in figures 6.12c and 6.12d. Kompics Scala `1.x` still struggles with the less efficient pattern matching implementation (see chapter 3) and even falls behind the Akkas eventually, despite their inferior networking library.

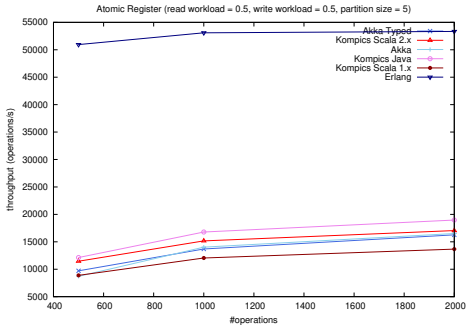


(a) Low pipelining.

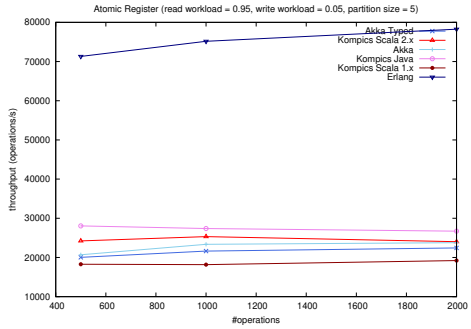


(b) Deep pipelining.

Figure 6.10: Results of the Net Throughput Ping Pong benchmark on AWS. Results show throughput in number of messages per second, thus higher is better.

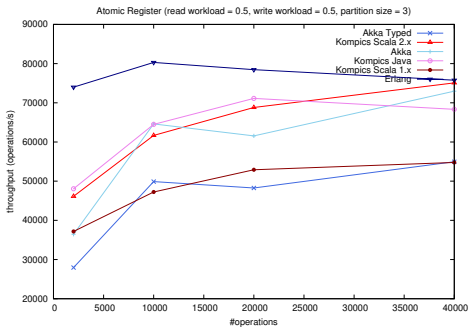


(a) 50/50 Reads/Writes.

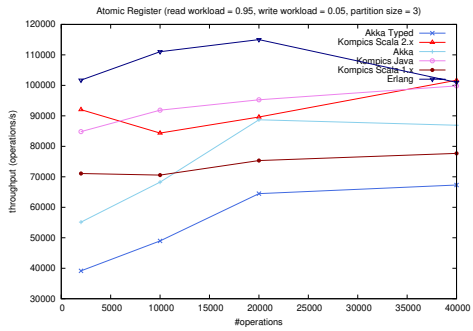


(b) 95/05 Reads/Writes.

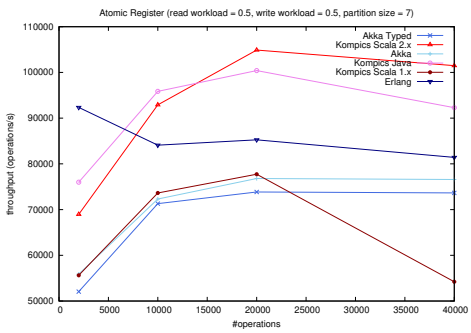
Figure 6.11: Results of the Atomic Register benchmark with 5 partitions on the Desktop. Results show the average throughput of the run in operations per second over all nodes, thus higher is better.



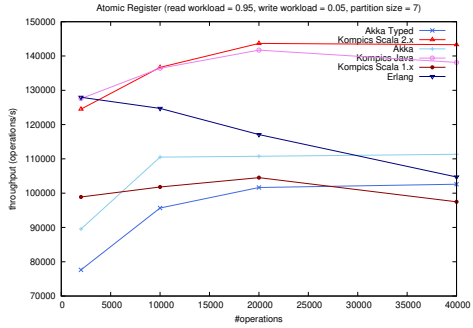
(a) 50/50 Reads/Writes with 3 nodes.



(b) 95/05 Reads/Writes with 3 nodes.



(c) 50/50 Reads/Writes with 7 nodes.



(d) 95/05 Reads/Writes with 7 nodes.

Figure 6.12: Results of the Atomic Register benchmark on AWS. Results show the average throughput of the run in operations per second over all nodes, thus higher is better.

Curiously, on this benchmark Akka outperforms Akka Typed consistently, despite having shows worse performance in the Throughput Ping Pong benchmark, and using exactly the same hash map implementation.

6.4.4.4 Streaming Windows

Figure 6.13 shows the results of the Streaming Windows benchmark. At a quick glance, it appears that no single implementation provides consistently good performance for this particular workload over a wide range of parameters.

While Erlang, as always, performs well when messaging-throughput is the dominating factor, as is the case in figure 6.13f as well as the higher x-axis values in figures 6.13a, 6.13c, and 6.13e, it does struggle significantly with large window sizes, such as for the lower x-axis values in figure 6.13e, as well as the larger x-axis values in figure 6.13d (it did not converge within 100 runs for figure 6.13b).

On the other hand, Kompics Java deals very well with large window sizes, but is again thwarted by its lack of addressing when the number of partitions increases. It is notable, though, that Kompics Scala `2.x` deals comparatively well with the parallelism here, again being able to avoid unnecessary scheduling of components from the wrong partitions.

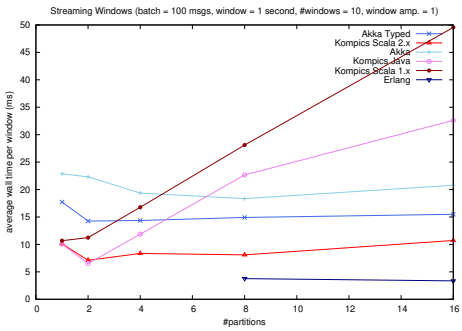
However, all the Scala implementations struggle with large windows, most likely because of the less efficient sorting options available in Scala `2.12`, as discussed in section 6.3.1.

6.4.5 Summary of Results

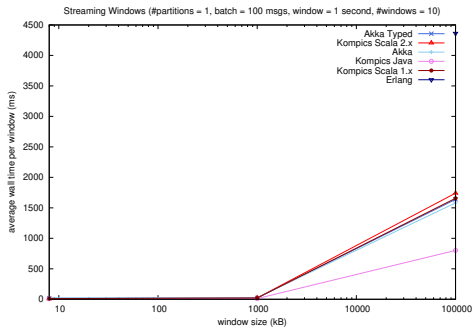
When it comes to pure message-passing tasks, Erlang performs strongly on the benchmarks presented in the previous sections, and outright dominates tasks where networking efficiency is important. However, its SMP implementation does still lag behind alternatives on the JVM, as seen in section 6.4.3.2, and its Beam VM is often challenged with performing memory and compute intensive tasks efficiently, as was seen in the All-Pairs Shortest Path and Streaming Windows benchmarks (sections 6.4.3.5 and 6.4.4.4), as well as the Atomic Register benchmark in section 6.4.4.3. It thus seems advisable for implementations targeting Erlang for its good communication performance, to "outsource" compute and memory bound subtasks to a native implementation, for example in C, which can be integrated in Erlang via *Native Implemented Functions*[11].

On the JVM, Akka and Akka Typed both offer very good performance for local tasks with classical actor behaviours, such as Throughput Ping Pong (section 6.4.3.2) and Fibonacci (section 6.4.3.3). This makes them a good compromise replacement for Erlang on compute and memory intensive tasks, as long as one keeps in mind certain inefficiencies in Scala libraries, that can usually be avoided by falling back to Java alternatives. However, their poor networking library often makes them

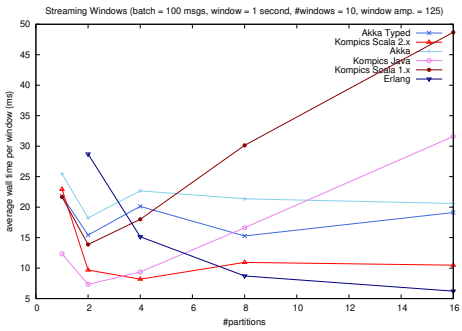
[11] See `http://erlang.org/doc/tutorial/nif.html`.



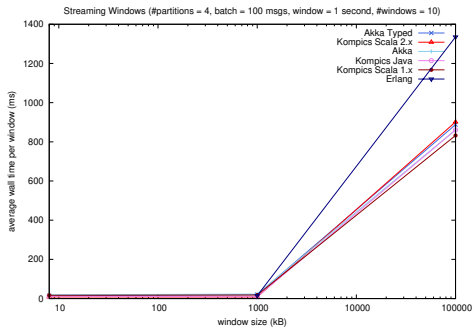
(a) 8 kB window size.



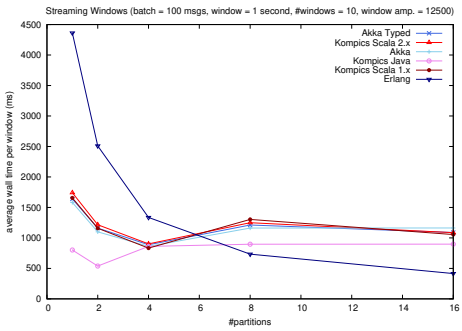
(b) Single partition.



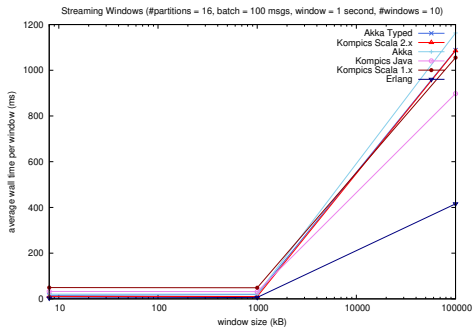
(c) 1 MB window size.



(d) 4 partitions.



(e) 100 MB window size.



(f) 16 partitions.

Figure 6.13: Results of the Streaming Windows benchmark in the AWS setup, with a batch size of 100 events and a total of 10 windows per partition. The left column shows scaling over the number of partitions for a fixed window size, while the right column shows performance for increasing window sizes for a fixed number of partitions. All results show the average time taken to complete a window, thus lower is better. More parallelism allows more windows be completed in the same time, thus reducing the per-window average time.

a less-than-optimal choice for a distributed actor system, where performance is relevant. On the hand, their convenience for the programmer in the distributed setup is only matched by Erlang.

Using Kompics implementations for performance relevant networking problems on the JVM can be very attractive, as long as it can be assured that their lack of addressing does not prove to be a bottleneck for the targeted application. But there clearly are networked application that match very well to Kompics' design and can benefit from its competitive networking library, as we will see in chapter 7. It is also worth noting that Kompics Scala `2.x` can often overcome the addressing challenges of its siblings, which in combination with its pleasant API (see chapter 3), make it the probably most attractive Kompics for developers.

When it comes to native code, the results have been mixed. Riker has clearly been a massive disappointment, only performing even marginally competitive on the All-Pairs Shortest Path benchmark (section 6.4.3.5), while displaying a particularly poor showing in every other local benchmark. Actix, on the other hand, often showed very competitive results at low parallelism, where its lack of SMP-support did not become a hindrance. This makes it a good choice for smaller deployment platforms, especially if memory is constrained and the remaining cores can be utilised effectively by other parts of the application, which do not run on Actix. Alternatively, Actix does support SMP for pure load-balancing applications, where identical actors perform the same task on round-robin balanced inputs, potentially allowing additional cores to be occupied with a number of fork-join tasks, while the main set of Actix actors runs single-threaded.

CHAPTER

7

The Power of Network Abstractions

In this chapter, we suggest a number of advantageous characteristics for APIs of message-passing network libraries. We then showcase the potential for performance gains of libraries employing these characteristics, using the example of a self-adjusting, application-level file transfer protocol.

While the results of chapter 6 have shown performance differences between the networking *implementations* of different message-passing frameworks, they cannot show the degree to which the choice of networking *abstractions* empowers programmers to design efficient, high-performance networked applications. In this chapter, we will describe the design of the Kompics networking library in detail, and investigate applications where it allows large performance gains under challenging conditions.

7.0.1 A Motivating Example

Consider, for example, the common task of moving a large file from one physical machine's local filesystem to that of another host over the network connecting them. If we want to do so as part of message-passing application, a very naïve implementation might simply read the whole file into memory — say into a Java `byte[]` array — in one component, pack it into a message together with some metadata about the file and send it to a component at the destination host. There it is received completely and put back into en equivalent message, and then written down to the local filesystem again.

While this may work well for small files, once the files sizes get larger our approach will eventually break down. This can happen when the file size exceeds the available memory, or much earlier when framework specific serialisation buffers fail to keep up with the data volume, for example.

We could deal with this problem, by splitting the file into multiple smaller messages, always noting the offset of the current "chunk" in the message headers, so the file can be correctly assembled again on the other side. If the chunk size is chosen appropriately, this approach prevents us from running out of serialisation buffer capacity, at the cost of some overhead for additional message headers.

However, if our limiting factor is physical memory availability, *chunking* alone does not help at all, as we will still be queueing up all the individual chunks on the network connection provider, and we also still keep the whole file in memory both at the receiver and sender side. Additionally, we need to ensure that we are only reading chunks from disk, when there is enough space available in memory to store them. In order to implement a resource management scheme for our transfer application's memory, we require information about when memory has been *freed* up, and can be reused. This acknowledgement could come from additional control messages running opposite the data transfer, similar to how TCP operates internally. But if we are anyway sending over TCP or other protocols with guaranteed delivery, sending this information again at the application layer is redundant. It would be sufficient for our networking abstraction to inform us once a message has been sent. Many message-passing frameworks, such as Akka [69] and Erlang [9], however, do not provide such a mechanism. In fact, they do not even allow us to differentiate between local messages and networked messages at all, a "feature" called *location transparency*. In such a framework, we are left stuck with either duplicating the

work TCP does, or using another protocol for all our traffic.

Assume now, we have chosen to continue with TCP in whatever way allowed by our message-passing framework. So far we have made no assumptions about network conditions between our source and target host. If the latency is low, our current approach is likely sufficient to make good use of the available bandwidth between the two machines. But as latency increases our throughput is likely to suffer, as it is a well-known result that common TCP implementations struggle with high bandwidth delay product (BDP) links. This occurs, because TCP implementations can not free up their sending buffers until they have confirmation that a package store in there was received, as otherwise they risk being unable to resend lost packages, violating their reliability guarantees.

In our application, however, we do not actually need to rely on TCP's no message loss guarantee, since we have all the data stored on local disk. We can simply read up the bytes again and resend them, as long as we can get information about which message was lost, and translate that to the corresponding offset in the file to read from. Thus again, TCP is doing redundant work for us, costing us in performance on inter-continental transfers, for example.

If we want our transfer performance to be useful in such circumstances, we have little choice but to either use a different protocol, or inverse multiplex our data over multiple TCP to increase the available sending buffer. But we only need these special guarantees for our file transfer; other parts of the application may not be able to tolerate message loss in this manner. Thus, we need a way for only our file transfer related messages to use a weaker protocol, such as User Datagram Protocol (UDP), while other messages continue to use TCP. Once again, in languages with location transparency, we can not make such a choice on a per-message basis. At best, we can completely change the implementation to the new protocol, and then implement our own reliability mechanisms wherever needed.

7.0.2 Network Abstraction Design

We can see from the examples in the previous section, that we require network abstractions with different properties than commonly available, if we want the ability to implement powerful application-level protocol embedded in our message-passing applications.

Explicit Networking It is clear that if we are to make detailed decisions about networking options as part of a message-passing application, we must be know which message may go over the network, and which may not. Apart from the fact that semantics of message delivery usually differ between local messages and networked messages, all the options related to protocol choices, for example, make no sense for local messages. Thus we propose that *location transparency* as a fundamental abstraction in message-passing systems be abandoned in favour of *explicit networking*. Of course, where appropriate location transparency can be

reintroduced as a higher-level abstraction, to allow convenient actor migration, for example.

Feedback Options Information on when and if a message was sent and received is needed for a wide variety of protocols. It should thus be provided by networking abstraction as a convenience option, that can be requested on demand.

Custom Headers Message headers are crucial part of most networking protocols. If we want to allow convenient creation of custom protocols, we should also allow programmers to describe custom header information for their protocol, instead of forcing them to include this information in message bodies, confounding data with meta information.

This abstraction can be particularly powerful in peer-to-peer systems, where due to network address translation (NAT) the direct destination of a message is often not the intended target recipient, and information about message forwarding points must be carried in the headers.

Customisation Options for Serialisation While automatic serialisation options, such as *Java Serializable* are often convenient for the programmer, it is often necessary to be able to to customise how messages are converted to and from bytes, in order to design efficient networked applications. However, as with network protocols, requirements for serialisation performance or size may vary across an application, and thus fine-grained control must be available to ensure a good balance between performance and developer productivity.

In this chapter we will describe the design of Kompics' (Java and Scala) networking abstraction, and explain how it fulfils the design goals outlined above. We will then show an implementation of the example use case from section 7.0.1 using Kompics Java and the described networking abstraction and its implementation. This file transfer application uses a reinforcement learning model to optimise the protocol selection for the encountered network conditions. We show how an implementation based on the design goals above can perform well in wide range of network conditions, and outperform other solutions for file transfer.

7.1 Kompics Messaging

Kompics' current default networking abstraction and implementation, which we shall simply call *Kompics Messaging* for convenience, is separated into two separate sub-projects:

Network Port This project contains the interfaces and port type declaration. It will be described in detail as the choices made there reflect the design decisions laid out in section 7.0.2.

```
1 public final class Network extends PortType {{
2     request(Msg.class);
3     request(MessageNotify.Req.class);
4     indication(Msg.class);
5     indication(MessageNotify.Resp.class);
6 }}
```

Listing 7.1: Kompics' Network Port.

Netty Network The default implementation for the **Network Port** is contained in this project. As the name suggests, the implementation uses the Netty networking library internally, which will describe briefly below for reference.

7.1.1 Netty

Netty [100] is a non-blocking I/O (NIO) network application framework for the JVM. It supports the rapid development of maintainable high-performance and high-scalability protocol servers and clients. Additionally, Netty provides a native byte buffer library that minimises unnecessary copying of data.

Netty is a perfect match for Kompics, because it is event-driven and asynchronous, making the interface between the two systems both seamless and efficient.

In order to provide non-blocking operations Netty has a highly customisable concurrency model employing constructs such as thread pools and futures, as well as a configurable channel handler pipeline.

Netty is well tested and already used in a number of large projects such as Apache Spark [116] and Apache Flink [18], as well as Akka [69] and at large technology companies like Facebook, Google and Spotify, according to `https://netty.io/wiki/adopters.html`.

7.1.2 Network Port

7.1.2.1 API

The core of the Kompics Messaging API is the `Network` Kompics port type itself, which is shown in listing 7.1. The network port allows messages that implement the `Msg` interface, which is reproduced in listing 7.2 along with its dependencies, to travel in both directions on network channels. Additionally, the network port allows the programmer to request notification of a message's status with respect to its destination, using the a `MessageNotify.Req` event. This must be answered with one or more corresponding `MessageNotify.Resp` events, indicating whether the message was sent, and later received successfully. If no such notification is requested, messages are simply "fire and forget".

```
1  public interface Address {
2      public InetAddress getIp();
3      public int getPort();
4      public InetSocketAddress asSocket();
5      public boolean sameHostAs(Address other);
6  }
7  public interface Header<Adr extends Address> {
8      public Adr getSource();
9      public Adr getDestination();
10     public Transport getProtocol();
11 }
12 public interface Msg<Adr extends Address, H extends Header<Adr>> extends
   ↪ KompicsEvent {
13     public H getHeader();
14 }
```

Listing 7.2: Kompics' address, header, and message interfaces.

Listing 7.2 also shows, that both `Msg` and `Header` allow subtypes of their generic parameters. Note that no implementations are provided for the the `Msg`, `Header`, and `Address` interfaces as part of the network port project. This was a deliberate choice, to allow and motivate application designers to pick implementations that suit their requirements, without having to extend existing classes and rely on runtime type-casts.

For example, if someone wanted to implement messages that can be forwarded through multiple intermediary hosts, but finally replied to directly, they might add an `Address origin` or `Address replyTo` field to the their `Header` implementation, and/or use a special routing header such as the one described in listing 7.3. Or a particular `Address` implementation might provide an additional `int id` field which disambiguates hosts with multiple network interfaces in use.

7.1.3 Semantics

It is important to note that the semantics of network messages differ from those of Kompics channels. While Kompics channels provide FIFO order with exactly-once delivery, network messages provide only at-most-once semantics. Even over protocols with no-message-loss reliability, like TCP, a sudden connection drop may lead to the loss of messages in practice. As replicating message acknowledgements on the middleware layer adds unnecessary complexity and overhead, it is, in general, not desirable. If message delivery is a concern for an application, it may use the `MessageNotify.Req` facilities to request delivery notifications. Alternatively, some algorithms can rely on session drop information alone, which is provided as part of an optional `NetworkControl` port.

```java
public class RoutingHeader<Adr extends Address> implements Header<Adr> {
  private final BasicHeader<Adr> base;
  //Forwardable Trait
  private Route<Adr> route = null;

  @Override
  public Adr getSource() {
    if (route != null){
      return route.getSource();
    } else {
      return base.getSource();
    }
  }
  @Override
  public Adr getDestination() {
    if (route != null && route.hasNext()) {
      return route.getDestination();
    } else {
      return base.getDestination();
    }
  }
  /* ... */
}
```

Listing 7.3: A multi-hop routing header.

While protocols like TCP and UDT maintain Kompics' FIFO semantics, they are not guaranteed when using protocols like UDP. Adding these semantics would defeat the point of having a lightweight protocol like UDP available in the first place.

The stark difference in semantics is balanced by the fact that Kompics Messaging does *not* provide location transparency and it is always clear to the developer whether or not messages might go over the network.

However, messages that *might* go over the network do not, in fact, always actually get serialised and sent over a link. In some circumstances, it is desirable to use "addressable" components in a way similar to the Actor model. Kompics, for example, provides support for this with a package for *virtual networks*, where, in addition to the network interface's IP address and port, an identifier is assigned to certain subtrees of the Kompics component hierarchy. Those subtrees are referred to as *virtual nodes* or *vnodes* for short. While communication across vnodes happens almost exclusively via the network port, the messages they send to each other within a single host are not guaranteed to ever be serialised and deserialised. Instead, it is recommended to detect such occurrences using the `boolean sameHostAs(Address`

`other`) method specified in the `Address` interface in listing 7.2 and "reflect" the message objects back up through the network port directly. In the case of the virtual network port, a special `VirtualNetworkChannel` implementation ensures that messages are only delivered to the destination vnode.

As a result of this behaviour, a programmer should never expect to receive actual copies of network messages, and always adhere to the default Kompics philosophy of *immutable messages/events*.

7.1.4 Transport Protocol Selection

While the `Header` interface does not require every implementation to have a setter for a `Transport` field, implementations are free to provide one. This leaves the system designer with the decision whether or not the transport protocol should be *hardcoded* for a specific message type, *injected* by a configuration at creation time, or even *replaced* on the fly by an interceptor component between the message sending components and the component implementing the network port. Such an adaptive transport system could measure network variables or have access to some deployment descriptor and would then decide the best protocol to use for a specific message type at runtime. See section 7.2 for details on our implementation of such a system.

Regardless of how the protocol selection for a message is performed, the networking component must ensure that the required channels, if any, are available. If necessary, new channels must be created and messages delayed until the requested channels are available.

7.1.5 Netty Network Implementation

As was already mentioned, Kompics' default network implementation, `NettyNetwork`, is based on the Netty networking library. `NettyNetwork` provides support for TCP, UDP, and UDT, based on Netty's built in support for those protocols.

Listening Ports When a `NettyNetwork` component is initiated, it must be provided with the protocols and ports to listen on. A single instance of the component only allows one port per protocol to listen on, but it is possible to start additional instances with different configurations, if more listening ports are required. Every instance manages its own Netty handlers and network channels, and must be appropriately connected with Kompics channels, to make sure messages reach the desired component instances.

Channel Management Channels for TCP and UDT are created automatically, when a message arrives with a destination that does not have an open channel already. UDP, of course, does not require channels. Channels are also eventually closed, when not in use for a long time, to reclaim resources. As channel establishment

can be expensive process, in circumstances where NATs must be dealt with, for example, the implementation is very conservative about closing channels.

As channels are created on demand, it can sometimes happen, when components from two different hosts send each other messages at roughly the same time, that two separate channels for the same protocol between the same pair of `NettyNetwork` instances are created. As this represents a waste of resources, the implementation runs a "disambiguation" algorithm over open network channels, that first identifies such instances, and then closes any superfluous channels. Great care has to be taken in this process, to avoid losing in-transit messages on the closes channel, or cause message reordering when switching from one channel to the other, either of which would violate message delivery guarantees expected by developers.

`NettyNetwork` also implements the optional `NetworkControl` port, which provides status information about opened and closed connections.

Channel Pipelines For the session protocols, channel pipelines include serialisation, framing, and compression, in that order on the outgoing side. Similarly, on the incoming side, frames are decompressed, then reassembled, and then deserialised. For UDP, serialisation happens directly on the `NettyNetwork` component thread, while channel pipelines for the session protocols run on Netty's thread pool, which allows for slightly better pipelining parallelism. UDP messages are also not compressed or framed, requiring developers to ensure that the messages they designate as to be send over UDP to be of appropriate size when serialised (typically less than 65 507 B minus the size of the `Header`, unless using *jumboframes*).

Serialisation Serialisation and deserialisation happens via Kompics' `Serializers` framework, which supports mappings from classes to specific custom serialisers. If no concrete mapping is supplied, supertype mappings are consulted. If no mapping is found for any supertype, then the framework refuses serialisation, and an exception is thrown. Individual serialiser implementations are expected to read from and write into Netty's `io.netty.buffer.ByteBuf` instances, in order to avoid the unnecessary copies incurred when writing to and reading from `byte[]` arrays.

7.2 Automatic Protocol Selection

In order to demonstrate the advantages of more powerful networking abstractions and implementations of such abstractions, such as the one described in section 7.1, we have implemented a file transfer application that automatically selects between sending individual chunks over TCP and UDT, in order to maximise overall throughput (and thus minimise transfer time).

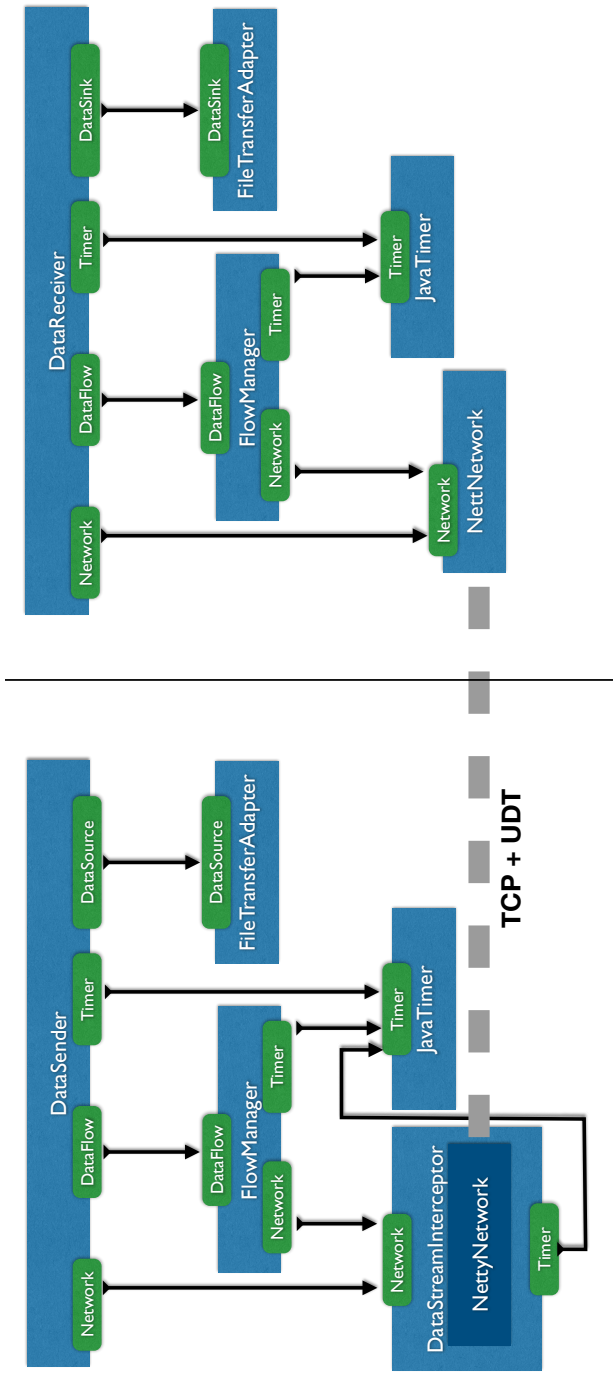


Figure 7.1: Component hierarchy for the file transfer experiment.

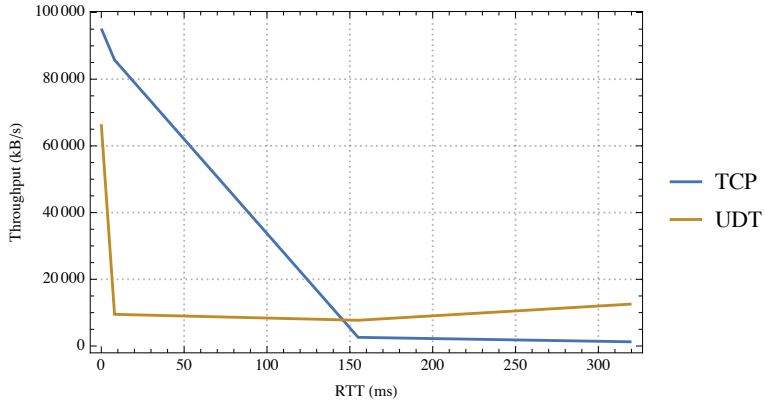


Figure 7.2: Comparison of transfer throughput at different RTTs between Amazon AWS data centres in Europe, the USA, and Australia.

7.2.1 Implementation

The file transfer application is written in Java using the Kompics Java framework, to make use of its KompicsMessaging networking implementation, as described in section 7.1. In addition to the Kompics default components `NettyNetwork` and `JavaTimer`, it consists of a number of custom components that can be seen in figure 7.1.

The `DataSender` and `DataReceiver` components act mostly as coupling components between the `FlowManager`, which negotiates available memory across sender and receiver system, and the `FileTransferAdaper`, which wraps a `RandomAccessFile` for writing or reading, respectively. The `DataSender` also acts as measuring point in our experiment, as it keeps track of the total time taken for a fixed sized transfer, to calculate its average throughput.

The most important component is the `DataStreamInterceptor`, which wraps the basic `NettyNetwork` on the sender side. In this component the selection between TCP and UDT occurs, if the selected transport protocol for a message was `Transport.DATA`. The mechanism for selection is described briefly below, with more details available in our paper on the subject [63].

7.2.1.1 Protocol Selection Mechanism

We know that, due to different implementations or environmental factors, TCP and UDT based transfer behave differently in real deployments, even though both protocols provide the same guarantees. In figure 7.2, for example, we see that in Amazon AWS the performance of UDT is very poor, unless using the loopback interface (marked at 0 ms latency). This occurs, because AWS rate limits UDP packages, which UDT is based on. However, as latency increases the performance

of TCP eventually falls below even the rate limited throughput of UDT, due to the limited buffering effect described earlier in this chapter. There is thus a cross-over point at which UDT based transfers are preferred over TCP based transfers.

We generalise from this example, and simply assume that in any environment there always exists an optimal protocol ratio r, of how much traffic should flow over TCP compared to how much traffic should flow over UDT.

Under this assumption, however, we face two challenges: First, we must be able to represent (a good approximation of) r on the wire, despite in the fact that we send discrete messages and not a continuous data flow. And second, we must discover the value of r for a particular environment. Since most network environments are not static, but change over time, we must, in fact, discover r for a particular environment at a particular time.

Pattern Selection Given a value $r \in [0,1] \subset \mathbb{R}$, such that r is the probability to send a message m over UDT instead of TCP, we must decide for each individual message arriving at our `DataStreamInterceptor` which actual protocol to replace `Transport.DATA` with. While we could simply pick UDT with probability r, this only produces the correct ratio on average over very long runs, but can show significant skew on short sub-sequences. As this skew hurts performance, we instead use fixed pattern based mechanism for selection. To do so, we approximate $r \in \mathbb{R}$ with a rational number $\tilde{r} = \frac{p}{p+q} \in [0,1] \subset \mathbb{Q}$ with $p, q \in \mathbb{N}_0$ and $p + q \neq 0$, such that $\tilde{r}$ is the nearest rational number to r. We then pick a pattern interleaving TCP and UDT messages, such that UDT occurs p times, and TCP occurs q times. For example, using regular expression notation and writing t for TCP and u for UDT, the pattern $(tu)^*$ represents $r = 0.5$, and the pattern $(tut)^*$ represents $r = 0.\bar{3}$.

As accurate patterns for very large values of p, q require equivalent storage space, we limit ourself to patterns that can be represented by relatively small DFAs, and accept the accompanying loss in accuracy and increase in skew as necessary.

The implementation of the protocol selection then runs through the DFA describing the current pattern, updating its state for each message and replacing `Transport.DATA` with the protocol indicated by the next transition in the DFA.

Ratio Selection In order to determine a currently advantageous value of r, we make use of a *online reinforcement learning* approach. For every time interval t (we use t =1 s), we calculate how much data has been transferred during t. We feed these values as *rewards* into an implementation of the *Sarsa(λ)* algorithm from Sutton and Barto [93], which is an online on-policy control algorithm for a *temporal difference learner* TD(λ). In our implementation of this algorithm, r represents the current state of the world, and possible *actions* are increases and decreases of r. We limit ourselves to a fixed step size $\kappa \in \mathbb{Q}$, as this allows us to represent the algorithms policy as a finite matrix and also interacts well with the pattern based learner described above.

However, even for moderate resolutions, such as $\kappa = \frac{1}{5}$, the resulting 11×5-matrix would be too large to explore efficiently. Luckily, it is unnecessary for us to keep the whole matrix, as we know that an action of incrementing by κ, changes the new state to be exactly $r + \kappa$ (or 1 if $r + \kappa > 1$). This allows us to reduce the policy size to only the 11 possible states.

As rewards are delayed with respect to our policy choices, however, this implementation still causes relatively slow convergence and excessive backtracking to clearly inferior states. To avoid this, we made an additional assumption that the function from ratio r to true throughput (ignoring delay) follows a simple quadratic function with a single maximum. Given this assumption, we can approximate the reward for an unexplored state using function approximation over the values we have already seen, with a minimum of two. This allows exploration to follow trends, reducing convergence time, and avoids excessive backtracking against the trend.

7.2.2 Experiment Setup

In order to measure the performance of our solution we performed two series of experiments. The first series took place completely in an Amazon AWS cloud environment, and measured throughput between TCP, UDT, and our "DATA" protocol at different latencies, using only our implementation described above. As AWS performs a lot of automatic management on their networks and their VMs, a second experiment was performed using "bare-metal" hardware and comparing against a number of other common file transfer solutions.

7.2.3 Amazon AWS

Environment We used pairs of Amazon EC2 `c3.2xlarge` (2016) instances, running Ubuntu `14.04 LTS` Amazon Machine Images (AMIs) with HVM virtualisation, in four different setups:

1) This setup (marked at 0 ms RTT) copied on the same node from one SSD to the other using the loopback interface. This setup simply measures disk throughput in the best case, or protocol or implementation buffer upper bounds in the worst case.

2) This setup (labelled "EU-VPC") had both instances within the same data centre, in this case the one in Ireland, and even within the same Virtual Private Cloud (VPC). We measured the RTT between the two hosts in this setup to be around 3 ms.

3) This setup (labelled "EU2US") used one instance in Ireland and the other one in North California in the USA. We measured the RTT between the two hosts in this setup to be around 155 ms.

4) This setup (labelled "EU2AU") still had one instance in Ireland, but the second instance was now in Sydney, Australia. We measured the RTT between the two hosts in this setup to be around 320 ms.

Dataset As a dataset to transfer, we used a NetCDF [31] climate data file, which is around 395 MB in size. A 395 MB dataset provided us with a good tradeoff between getting the protocols ramped up even on very fast links, while allowing us to run multiple iterations of the experiments even on the slowest links, for statistical significance. It should be noted that, since our KompicsMessaging implementation has *Snappy* compression in the channel pipelines by default, the exact results might differ if the experiments are repeated with data than can be more (or less) easily compressed. We could have used a completely random dataset to avoid this issue, but a realistic dataset seemed of greater interest. Climate data must commonly be transferred from the high-performance computing (HPC) datacentre running the simulation, or the archival datacentre storing simulation results long term, to researchers interested in analysing the data.

Methodology We measured the throughput of our implementation by repeatedly sending the transfer dataset from the first instance to the second and recorded the disk-to-disk transfer time. For each transport protocol we would do at least 10 runs, sometimes more, until the RSE dropped below 10 % of the sample mean.

The error bars, where visible[1], show the 95 % confidence interval for the sample mean.

Results The results for this experiment are shown in figure 7.3, which shows our DATA protocol in addition to the data for TCP and UDT already presented as motivation in figure 7.2. As expected, TCP shows very good performance at low RTT, but a sharp drop-off at higher values. In the local scenario, in fact, both TCP and DATA are limited by disk performance, as we reached even higher throughput of around 150 MB/s in memory to memory scenarios. In those scenarios UDT seems to be limited by internal queue and buffer sizes. As was already noted, UDT shows very consistent behaviour across all setups with real network, since Amazon artificially rate limits UDP traffic to around 10 MB/s (in 2016).

While TCP vastly outperforms UDT within a VPC, at longer RTTs UDT is up to an order of magnitude faster. Our learner implementation shows the desired behaviour of following TCP closely, where TCP is strong and matching UDT, where it outperforms TCP, giving the best of both worlds. The only drawbacks of the DATA implementation are the higher variance and a certain ramp up time, which cause the imperfect performance at the 3 ms level, where the transfer time is too short to amortise the learning time.

[1] For most data points, the errors are simply very small, as many of these tests have very consistent results.

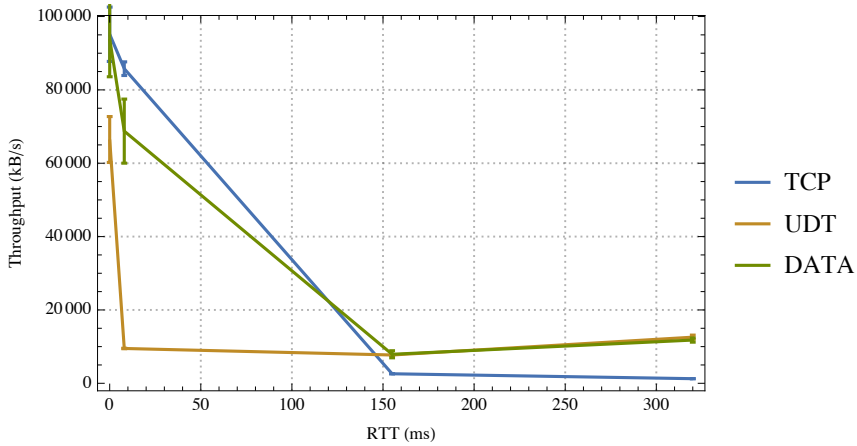


Figure 7.3: Data transfer throughput for different RTTs, over TCP, UDT, DATA (error bars show 95 % confidence intervals). It can be seen that DATA behaves close to optimal at all RTTs.

7.2.4 Bare-Metal

Environment This experiment was run between the SICS ICE cluster in Luleå, Sweden, and the Purdue Halstead cluster in West Lafayette, Indiana, USA. We measured the RTT between those two data centres to be around 145 ms. No virtualisation was used, neither for networking nor for operating systems.

The Halstead cluster consist of 508 machines, each with two 10-core Intel Xeon-E5 CPUs at 2.6 GHz and 128 GB of main memory. They are connected amongst themselves by 100 Gbit/s *Infiniband* links.

The SICS ICE cluster consists of 38 machines, each with two 6-core Intel® Xeon® E5-2620 CPUs at 2.4 GHz and 256 GB of main memory. They are connected among themselves with a mix of 1 Gbit/s and 10 Gbit/s network links.

Dataset For this experiment we used a completely random file of 3 GB, produced by the following shell command:

```
1 head -c 3G </dev/urandom >test.file
```

The reason that we moved away from real data is that in this experiment we wanted to compare with other file transfer solutions, which may use different compression than our own experiment. Thus a compressible file would risk distorting results, depending on which compression method is used by each implementation.

Methodology In addition to comparing our own solutions in the same manner described before, in this experiment we also wanted to compare against other

Provider	Throughput	Fraction of Maximum
DATA	65.0177 MB/s	100.00 %
UDT	60.5427 MB/s	93.12 %
Globus Relay ×2	33.90 MB/s	52.14 %
TCP	18.8806 MB/s	29.04 %
Globus Relay	16.95 MB/s	26.07 %
HTTP	14.8 MB/s	22.76 %
SCP	11.7 MB/s	18.00 %

Table 7.1: Average throughput in the ICE to Halstead bare metal experiment.

common file transfer solutions for reference. For this purpose we also measured performance of SCP (provided by OpenSSH) and HTTP (provided by the `wget` tool), as well as the Globus[2] platform for scientific data management. The latter is common for transfers of large scientific datasets, such as climate data, and was already pre-installed on the Halstead cluster.

While Globus supports inverse multiplexing over multiple TCP connections, we forced it to use only a single channel in this experiment, to allow for a fair comparison with TCP-only performance of our implementation, as well as SCP and HTTP. It should be noted, of course, that our DATA implementation is technically a two-channel implementation, and a fair comparison with it would require letting Globus run in two-channel mode, as well. However, we can approximate the best-case scenario of Globus' two-channel implementation, by simply doubling its single-channel performance, as clearly adding more channels can, at best, increase performance linearly.

Results The results of this experiment are summarised in table 7.1, and it becomes clear that our implementation is very competitive. Not only does our DATA implementation outperform every other implementation, with almost double the performance of a hypothetical two-channel Globus, but our own single-channel TCP-only implementation also outperforms every other single-channel TCP implementation.

7.3 Related Work

Protocols The performance of different transport protocols has been well studied, especially relating to TCP [32, 21], and many alternative congestion control algorithms and protocols have been proposed. Among them are solutions that address high BDP links [43, 111], specifically satellite links [83, 17], data centre networks [115, 3], lossy links such as wireless connections [78, 72] and adaptive approaches [113]. Changing TCP's congestion control algorithm, however, typically

[2] `https://www.globus.org/`

requires modifications to the operating system kernel and such access can not reasonably be expected in systems like internet-scale peer-to-peer deployments, for example.

There is also a number of alternative protocols with similar guarantees to TCP like *Aspera FASP* [11], UDP-based Data Transfer Protocol (UDT) [42], Performance-oriented Congestion Control (PCC) [28], and Low Extra Delay Background Transport (LEDBAT) [92].

Lastly it has also been attempted to inverse multiplex data over multiple parallel TCP connections [12] or even multiple networks paths [114, 87, 34]. This approach is used, for example, in Globus as well.

Middleware There is a number of message-oriented middleware systems including *Akka Remote* [69], *Distributed Erlang* [112], *Websphere MQ* [39], SIENA [20], and *ZeroMQ* [49]. As mentioned in chapter 2, Akka and Erlang are most similar to our work since they combine a message-passing programming framework with a network messaging layer. However, while Akka does provide support for plugging in custom transport implementations, actors within an `ActorSystem` are fairly inflexible when it comes to the choice of transport used for their messages. In Distributed Erlang, the choices of protocol is completely up to the runtime system, making it even less flexible than Akka. This is, of course, a consequence of both Akka and Erlang providing *location transparency* for their actors, never really exposing the circumstance that messages may in fact travel to another host. In our system, on the other hand, networking is always explicit, in order to avoid unexpected behaviour caused by different channel semantics.

To the best of our knowledge, even modern Actor-based programming models, designed with specific support for multi-cloud environments, such as Microsoft Orleans [101, 15], have not addressed the issue of mixed congestion control protocols.

However, the approach taken recently by *PARTISAN* [80], an alternative to the default networking layer for Distributed Erlang, is interesting in that it allows multiple TCP channels between the same pair of hosts. While this is not quite as flexible as allowing different protocol implementations to be selected at runtime, it does give the same advantages as the inverse multiplex TCP method used by Globus, as described above.

Automatic Protocol Selection Wachs et al. [110] have investigated solutions for selecting networking protocols automatically, based on heuristic, linear programming, and reinforcement learning, with a focus on decentralised, peer-to-peer networks.

CHAPTER

8

A Fusion of Components and Actors

In this chapter, we describe Kompact, a Rust DSL and library that implements a statically-typed hybrid of the Actor model and Kompics Component model. We show that this hybrid approach allows programmers to pick the best model for their current problem, without having to compromise on either performance or expressive power.

While chapter 7 has shown that we can build highly competitive networking applications in Kompics Java, it is also apparent that this performance comes from good abstraction choices, and not from a very fast implementation of the Kompics component model. In fact, we have already seen in section 3.3 that Akka easily outperforms all three Kompics implementations at very simple tasks, and this early impression has, more often than not, been supported in section 6.4.

The fundamental issue at the heart of Kompics' performance problem is, that any given component does not necessarily want to handle all messages delivered on the channels it is connected to, as we have described in detail in sections 2.2.2.2, as well as in the introduction to chapter 3. To avoid scheduling components unnecessarily on the thread pool, Kompics implementations rely on a *subscription-checking* mechanism that guarantees that only components that are actually going to handle an event e are scheduled into response to being delivered e, as described in section 3.2.2. This relatively expensive check is performed on the sending thread, reducing the potential for pipelining parallelism in the single connection case, but — even worse — slowing down the sending component linearly to the number of connected receiving components.

This issue becomes particularly exacerbated in components with very high connectivity, such as the network component. It can easily be seen in figure 8.1, which is part of the *Network Throughput Ping Pong* benchmark described in section 6.2.2.2, how the network throughput of all three Kompics implementations drops as more and more components are connected to the network component. As was already pointed out in section 6.4.4.2, Kompics Scala `2.x` suffers somewhat less drastically from this issue, but it still does eventually, as pattern matching overhead is still linear in the number of connected components on a port. Evidently, this particular behaviour is not mirrored by the Akkas, which essentially just perform a single actor reference lookup and mailbox append operation for each message. It does appear to be very slightly mirrored by Erlang, however, which may be due to poor actor reference lookup scaling in its runtime, or may be a completely unrelated problem, considering how much higher its base performance already is.

Thus, the fundamental difference between the Kompics component model, where receivers are abstracted away, and the Actor model, where receivers are explicitly addressed, has a significant impact on the assumptions we can make for our implementations, with respect to checks we must perform to deliver messages or events. As we have seen, these assumptions can have a powerful impact on the performance we can achieve on certain tasks. In other words, either model can perform well, when it is playing to its strengths, but very poorly when fighting against its limitations.

Channel Filters This particular limitation of the subscription-checking bottleneck on highly connected ports has already been recognised to a degree by the original authors of Kompics, who provide a *channel filter* mechanism to reduce its perfor-

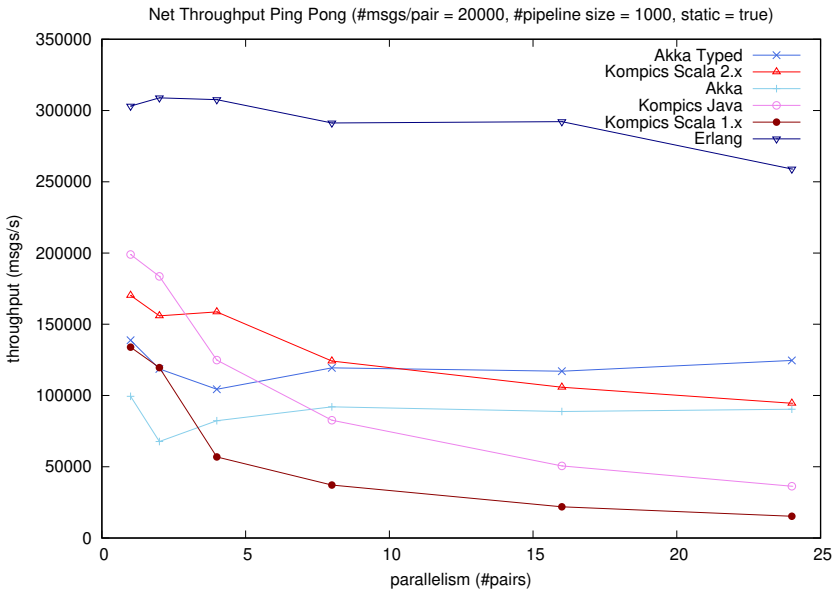


Figure 8.1: Falling network throughput as a result of the subscription-checking bottleneck. *(This is the same figure as 6.10b, reproduced here for convenience.)*

mance impact [8, section 2.4]. Channel filters, which are called *channel selectors* in the most recent Kompics implementation [99], allow for constant time lookups of relevant outgoing channels based on some configurable attribute of the event being triggered, thus often avoiding expensive and futile subscription-checking. The use of channel filters can improve performance of Kompics implementations significantly, when used judiciously.

However, apart from reducing code readability by increasing code size, channel filters do not alter the fundamental Kompics assumption; namely, that not every event arriving at a port is also actually handled by the port's component. This continues to necessitate the expensive subscription-checking mechanism.

Actors vs. Components Both models clearly have their strengths and weaknesses. The Actor model allows implementations to always schedule actors when a message is addressed to them, as most messages delivered are typically handled. The ability to send actor references as data also allows for a wide range of dynamic connection topologies. While these may be difficult to understand, and thus increase the mental burden of maintenance, they can be a powerful tool for building applications.

On the other hand, the Kompics model provides the ability to build very clear abstractions, using its port and component (implementation) separation. It is also a model that can very easily be typed statically, as seen in chapter 4, something that

actor implementations have struggled with for a long time. The Kompics model also provides excellent abstractions for one-to-one processing pipelines, as well as natural broadcast relationships, such as failure detectors.

Actors plus Components Given that both models have their strengths and weaknesses, the question arises, as to whether it is possible to combine them and get "the best of both worlds"; that is, always using each model in a manner that is playing to its strengths. In a certain sense, the networking implementation described in chapter 7 already provides some features of the actor model, namely direct addressing, but it is limited to whole component systems, instead of individual components.

Fowler et al. [35] have recently described a calculus that allows both typed asynchronous channels, similar to the ones found in Kompics, *and* typed actors, similar to those found in Akka Typed, and shown that the two models are, in fact, equivalent, in the sense that they can be translated into one another while preserving semantics. While the channel model they describe is not exactly the same as Kompics' model, it is similar enough to suggest that such a union between actors and components is not only possible, but even semantically sensible.

A Hybrid Model There is more than one way one could imagine to combine the Actor model with the Component model. A very simple approach could be to allow both traditional actors and traditional components to coexist within the same runtime, each with their own semantics. In such a framework we would have to introduce some kind of translation mechanism that we then would use whenever we need information to pass from actors to components or vice versa. This is, in fact, something the author of this dissertation has done before, mostly out of necessity in applications where some parts were implemented in Akka and some parts in Kompics, and they needed to interact somehow.

However, for this chapter we will adopt a more fundamental notion of the term *hybrid model*. And that is that *every component is also an actor*. Or in other words, our reactive state-machines, which is what both actors and components fundamentally are, are all able communicate via *ports and actor references*, and are all able to receive and handle both *direct messages and events*.

In some sense, then, we are simply treating actor mailboxes as a special kind of port. Or, if we are looking at things from Actor model's point of view, we are treating ports as additional mailboxes that have limitations on how they can be addressed. Either perspective is equally valid.

Whichever perspective we prefer, it is clear that we must rely on our scheduling and execution mechanisms to distribute computation time fairly between all the different ports or mailboxes on such an "actor component".

Design Goals for a Hybrid Framework In this chapter we will describe such a hybrid model between Kompics components and actors. Our primary motivation

for this implementation is the desire to arrive at an efficient framework for writing distributed systems in. In particular, our primary target applications are continuous online event processing applications, such as streaming analytics engines.

Those systems do not only require high throughput, but also low and, most importantly, *consistent* processing latencies. For this particular reason, we purposefully avoided garbage collected programming languages as implementation targets.

Additionally, static typing was of crucial concern to us, as catastrophic runtime failures of a streaming pipeline that is deployed on a cluster is not only difficult to debug, but can also have more dire consequences, in the case where it was part of a timing sensitive, safety relevant application, for example.

Of course, the same can be said for runtime memory violations in languages that are not garbage collected or those that are avoiding the garbage collector for performance reasons, as systems like Apache Flink [18] do, for example.

These factors led us to choose the Rust language [102] for our implementation, as it provides us with static memory safety guarantees in addition to avoiding garbage collection.

In the remainder of this chapter we will describe and evaluate the *Kompact*[1] framework, which combines ideas from the Actor model and the Kompics component model. In addition to presenting a new model, we also incorporate all the static typing rules from section 4.2.1, and even avoid warning 1 from section 4.1. This, of course, comes at the cost of reduced flexibility in certain areas. As we have done in previous chapters, we will begin by describing the DSL of our new implementation, before diving into some of the implementation details. Finally, we will compare Kompact with other message-passing frameworks, using the MPP suite described in chapter 6, and show its potential for significant performance improvements compared to the state-of-the-art.

For readers unfamiliar with the Rust language, we give a quick overview of its most relevant features in appendix B.

8.1 The Kompact DSL

The Kompact framework is a Rust library, which provides both abstractions and an eDSL to write message-passing applications in a combination of actor and component style. Additionally, the library also provides a default networking implementation, which is based around similar design goals as the ones described in the introduction of chapter 7.

Listings 8.1 and 8.2 show an overview of a Kompact component (and actor, but we will simply say component[2] in this chapter) using the by now hopefully

[1] Kompact is a portmanteau of **K**ompics c**omp**onents and **act**ors. A *compact* is also a word for an *agreement*, a reference to the statically typed nature of our approach.

[2] Some people like to refer to this as a *kompactor*, but the author of this dissertations opines that this does not evoke an appropriate mental model.

```
use kompact::prelude::*;
use std::collection::HashSet;

type V = /* anything that is serialisable and cloneable */;

#[derive(Clone, Debug)]
enum RegisterRequest {
  Write{ v: V },
  Read,
}
#[derive(Clone, Debug)]
enum RegisterResponse {
  WriteReturn,
  ReadReturn{ v: Option<V> },
}
struct OneNRegularRegister;
impl Port for OneNRegularRegister {
  type Request = RegisterRequest;
  type Indication = RegisterResponse;
}

#[derive(Clone, Debug)]
enum ROWAMsg {
  WriteMsg(V),
  Ack,
}
impl Serialiser<ROWAMsg> for ROWAMsg { /* ... */ }
impl Deserialiser<ROWAMsg> for ROWAMsg { /* ... */ }

#[derive(ComponentDefinition)]
struct ReadOneWriteAll {
  ctx: ComponentContext<Self>,
  onrr: ProvidedPort<OneNRegularRegister,Self>,
  pfd: RequiredPort<PerfectFailureDetector,Self>,
  peers: Vec<ActorPath>,
  correct: HashSet<ActorPath>,
  value: Option<V>,
  write_set: HashSet<ActorPath>,
}
impl ReadOneWriteAll {
  pub fn new(peers: Vec<ActorPath>) -> ReadOneWriteAll {
    let correct: HashSet<ActorPath> = peers.iter().copied().collect();
    ReadOneWriteAll {
      ctx: ComponentContext::new,
      onrr: ProvidedPort::new(),
      pfd: RequiredPort::new(),
      peers,
      correct,
      value: None,
      write_set: HashSet::new(),
    }
  }

  fn check_condition(&mut self) -> () {
    if (self.correct.is_subset(&self.write_set)) {
      self.write_set.clear();
      self.onrr.trigger(RegisterResponse::WriteReturn);
    }
  }
}
```

Listing 8.1: The Read-One Write-All algorithm in Kompact (*part 1*).

```
ignore_control!(Ponger);

impl Provide<OneNRegularRegister> for ReadOneWriteAll {
  fn handle(&mut self, event: RegisterRequest) -> () {
    match event {
      RegisterRequest::Read => {
        self.onrr.trigger(RegisterResponse::ReadReturn {
          v: self.value.clone()
        });
      },
      RegisterRequest::Write { v } => {
        for p in self.peers.iter() {
          p.tell(ROWAMsg::WriteMsg(v), self);
        }
      },
    }
  }
}

impl Require<PerfectFailureDetector> for ReadOneWriteAll {
  fn handle(&mut self, event: Crash) -> () {
    correct.remove(&event.p);
    self.check_condition();
  }
}

impl NetworkActor for ReadOneWriteAll {
  type Message = ROWAMsg;
  type Deserialiser = ROWAMsg;


  fn receive(&mut self, sender_opt: Option<ActorPath>, msg: Self::Message) {
    let sender = sender_opt.take().expect("Should have sender!");
    match msg {
      ROWAMsg::Write(v) => {
        self.value = Some(v);
        sender.tell(ROWAMsg::Ack, self);
      }
      ROWAMsg::Ack => {
        self.write_set.insert(sender);
        self.check_condition();
      }
    }
  }
}
```

Listing 8.2: The Read-One Write-All algorithm in Kompact (*part 2*).

familiar example of the Read-One Write-All regular register algorithm (algorithm 2). Clearly, Kompact compares poorly to Kompics Scala or Akka in conciseness, despite Rust's convenient local type inference and pattern matching capabilities. On the positive side, however, the example does not contain any dynamic checks (apart from deserialisation), and is thus fully statically typed. Note, however, that we introduced a type alias `V` for the value type of the register. This increases readability, as using Rust's top type `Any` (as we have done in listings 3.1, 3.2, 3.3, and 4.5) in an appropriate manner here would be rather verbose. We have also replaced the best-effort broadcast and perfect link abstractions with the use of simple remote messaging, as this is both more efficient and allows us to show off the hybrid nature of our approach in the example.

8.1.1 Events/Messages and Ports

In Kompact any type can be used as an event, provided that:

- Its size is known at compile time, as verified by the `Sized` trait.
- It can be passed safely between threads, as verified via the `Send` trait.
- It does not contain any reference with a lifetime other than `'static`.
- It implements the `Clone` trait, which is necessary to send different owned values to each receiver during channel broadcasting.
- It implements the `Debug` trait, which is used for internal logging.

Messages, which are addressed directly to an (actor) component, have similar but slightly different requirements. For one, messages do not need to be `Clone` as they are addressed to a single destination and not broadcast. Additionally, Kompact differentiates explicitly between *local* and *network* messages.

Local messages are statically typed in the same manner as in Akka Typed. That is, every Kompact component must choose a single type `M` of messages it handles. Every actor reference to that component is then parametrised by `M`, ensuring that only instances of `M` are ever send to the component.

Network messages must fulfil the same requirements as normal messages. Additionally, a network message of type M must be serialisable in some manner. This can be achieved by either implementing the `kompact::Serialisable` trait on them, or pairing them with an appropriate implementation of `kompact::Serialiser<M>`. Note that, in general, network messages are not required to respect the message type chosen for the components *local* messages. This accounts for the fact that it is generally impossible to prevent remote systems from sending arbitrary messages to a component over the network. However, if the `NetActor` trait is used to implement the actor part of a Kompact component, the implementation will enforce the local

```
#[derive(Clone, Debug)]
enum RegisterRequest {
  Write{ v: V },
  Read,
}
#[derive(Clone, Debug)]
enum RegisterResponse {
  WriteReturn,
  ReadReturn{ v: Option<V> },
}
struct OneNRegularRegister;
impl Port for OneNRegularRegister {
  type Request = RegisterRequest;
  type Indication = RegisterResponse;
}
```

Listing 8.3: Ports and events in Kompact.

message type on networked messages as well, discarding any messages that do not match silently.

A port in Kompact is a, typically zero-sized, `struct P`, that implements the `kompact::Port` trait, which has no methods, but only two associated types: `Request` and `Indication`. That is, Kompact ports only allow a single concrete type to pass in either direction. However, ports with multiple types per direction can easily be achieved, by wrapping all the types for one direction in a custom `enum E` and using `E` as the single direction type for the port `P`. An example of a port definition with its associated enum-wrapped events can be seen in listing 8.3.

8.1.2 Component Implementation

A component in Kompact is, at the most basic, simply a struct with a context field of type `ComponentContext<Self>`. Additionally, the type must implement the `Actor`, `ComponentDefinition`, and `Provide<ControlPort>` traits. A minimal example of such a component can be seen in listing 8.4.

Actor and Component Definition The `Actor` trait implements the handling of both local and networked messages, while the `ComponentDefinition` trait implements the execution of events on ports. In particular, the latter determines the order in which ports are checked for available messages and the fairness between ports. Additionally, it helps set up the component context correctly during component creation. The `Actor` trait is described in more detail in section 8.1.2.2.

Both traits can be automatically derived, as seen in the first line of figure 8.4, if no special handling is needed or desired by the programmer. In fact, this is

```rust
#[derive(ComponentDefinition, Actor)]
struct HelloWorldC { ctx: ComponentContext<Self> }
impl HelloWorld {
  pub fn new() -> HelloWorldC {
    HelloWorldC { ctx: ComponentContext::new() }
  }
}
impl Provide<ControlPort> for HelloWorldC {
  fn handle(&mut self, event: ControlEvent) -> () {
    if let ControlEvent::Start = event {
      println!("Hello World!");
    }
  }
}
```

Listing 8.4: Minimal "Hello World" component in Kompact.

the most common case for `ComponentDefinition`, which should rarely be written by hand. The derived version provides round-robin fairness among all ports and checks them in the order they are declared in on the component struct.

Control Port Implementing the `Provide<ContolPort>` trait is equivalent to a control port handler in other Kompics implementations, and gives access to the lifecycle events, which generally have the same semantics as in Kompics (see section 2.2.2.3):

```rust
pub enum ControlEvent { Start, Stop, Kill }
```

The `Provide<ContolPort>` trait *must* be implemented for all components, even if no special actions for lifecycle events are needed, as it serves as a type-level proof that a struct actually is a valid component. The Kompact crate[3] provides a convenience macro `ignore_control!(C)`, which generates a valid "empty" implementation for any type `C`.

Context The context field of type `ComponentContext<Self>` gives a component access to Kompact internal services, such as logging, timers, and message dispatching, as well as the component's unique id. It also provides both of the two Kompact variants for actor references: A local reference of type `ActorRef<Self::Message>` and a remote reference of type `ActorPath`, described in detail in section 8.1.2.2.

Constructors As opposed to other Kompics implementations, Kompact places no particular restrictions on component constructors, and they are simply created like any arbitrary struct would. Of course, hiding the fixed parts of the initialisation,

[3] A "crate" is a published library in Rust.

```
4  #[derive(ComponentDefinition)]
5  struct BasicBroadcast {
6    ctx: ComponentContext<Self>,
7    beb: ProvidedPort<BestEffortBroadcast, Self>,
8    pl: RequiredPort<PerfectPointToPointLink, Self>,
9    peers: Vec<Address>,
10 }
11 impl BasicBroadcast {
12   pub fn new(peers: Vec<Address>) -> BasicBroadcast {
13     BasicBroadcast {
14       ctx: ComponentContext::new(),
15       beb: ProvidedPort::new(),
16       pl: RequiredPort::new(),
17       peers,
18     }
19   }
20 }
```

Listing 8.5: A Pure component BasicBroadcast struct in Kompact.

such as component context and port instance creation, inside a constructor method, is generally recommended for readability. An example of this can be seen in listing 8.5 in the `pub fn new(peers: Vec<Address>)` method.

8.1.2.1 Ports and Event Handling

Port Instances are created as component struct fields of the `ProvidedPort<P, Self>` or `RequiredPort<P, Self>` types, for a port type `P: Port`. Declaring such a port field also causes the Rust compiler to enforce matching implementations of the `Provide<P>` or `Require<P>` trait respectively, ensuring that any message arriving on a declared port is actually handled. This is achieved by adding the following requirements to the second `Self` type parameter: It must implement the `Provide<P>` or `Require<P>` trait, in addition to being a `ComponentDefinition`, which may not contain any non-static references. These requirements are expressed as a set of *trait bounds* in Rust, which take the form `C: ComponentDefinition + Provide<P> + 'static` (or equivalent for `Require<P>`), where `C` corresponds to the `Self` type parameter, as seen in lines 7 and 8 of listing 8.5.

Like the `ComponentContext<Self>` field, port instances are simply initialised with their standard `new()` constructor, as seen in lines 14-16 of listing 8.5.

If multiple ports are declared, the default derive macro for `ComponentDefinition` automatically generates round-robin fair execution code, that ensures none of the ports suffer from *starvation*. However, custom implementations of that trait may enforce unfair *port priorities*, if desired.

```
26 impl Provide<BestEffortBroadcast> for BasicBroadcast {
27   fn handle(&mut self, broadcast: Broadcast) -> () {
28     for peer in peers.iter() {
29       self.pl.trigger(Send::new(peer.clone(), broadcast.msg));
30     }
31   }
32 }
33 impl Require<PerfectPointToPointLink> for BasicBroadcast {
34   fn handle(&mut self, deliver: Deliver) -> () {
35     self.beb.trigger(deliver);
36   }
37 }
```

Listing 8.6: Pure component BasicBroadcast event handlers in Kompact.

Event Handlers are created as implementations of the `Provide<P>` or `Require<P>` traits on the component. This means that each component can only have a single statically declared and typed handler for each port type they declare. As described above, the existence of these handlers is statically enforced by the compiler, which completely prevents the "forgotten handler subscription" warning 1 we discussed in section 4.1. The handler implementations for a hypothetical pure-component example of the Basic Broadcast algorithm (algorithm 1) can be seen in figure 8.6.

A mutable reference to the component instance (i.e. `&mut self`) is passed to any handler implemented for it, every time it is invoked with an event. In (safe) Rust, this type signature indicates clearly that the compiler will statically guarantee that the handler always has thread safe, exclusive access to the component's internal state.

Each of the handler traits must be implemented for a port type `P: Port`. If so, implementations of the `Provide<P>` trait will handle events of type `P::Request`, and implementations of the `Require<P>` trait will handle `P::Indication` events. This implements the strict T-Subscribe typing rule from section 4.2.1, restricted to a single type in each direction.

Event Triggers are achieved by invoking the `trigger(event)` method on the appropriate port instance field. Examples of triggers in each direction can be seen in listing 8.6 in lines 29 and 35.

Port instances of type `ProvidedPort<P, Self>` allow triggering of `P::Indication`, while instances of `RequiredPort<P, Self>` allow `P::Request` to be triggered. Again, this implements the strict T-Trigger typing rule from section 4.2.1, restricted to a single type in each direction.

8.1.2.2 Actors and Message Handling

The actor behaviours of Kompact are accessed by implementing the `Actor` trait reproduced below, instead of of deriving it:

```
1 pub trait Actor {
2     type Message: MessageBounds;
3
4     fn receive_local(&mut self, msg: Self::Message) -> ();
5
6     fn receive_network(&mut self, msg: NetMessage) -> ();
7 }
```

The trait separates local and network messages into two separate methods, both of which have exclusive access to components internal state, just like event handlers do, as indicated by the `&mut self` argument. Sender information is implicitly carried only over the network and is accessible via the `pub fn sender(&self) -> &ActorPath` function on the `NetMessage` instance. For local messages, sender information must be carried explicitly in the body of the message, as is the case for Erlang and Akka Typed, for example.

Local Messages are statically typed in Kompact, respecting the associated `Message` type defined on the `Actor` trait. This makes the handling of local messages feel very similar to events as described in the previous section. An example of local message handling can be seen in listing 8.7, lines 32-35, where `Self::Message` is of type `BroadcastMsg`, and we can simply extract the payload via pattern matching and deliver it to subscribers of the `self.beb` port.

Network Messages are in some sense more dynamically typed, as we can never truly know what is being sent to a component over the network. As opposed to Kompics and Akka, Kompact does not eagerly deserialise whole messages at the networking layer. This avoids both unnecessary work for messages that will not actually be handled, as well as the significant overhead of

1) inspecting a message against all globally subscribed deserialisers,
2) deserialising it, if a match was found,
3) moving it to the heap, so it can be hidden behind the dynamic `Any` trait,
4) sending a reference to heap allocated data (i.e. a `Box<Any>`) to the target component, and
5) finally downcasting it again via reflection and possibly moving it off heap again in the message handler at the target component.

Instead of following this inefficient chain of transformations, in Kompact deserialisation happens only at the target component, after it has verified it is actually interested in messages with the *serialisation id* that was provided in the message's

```
21 ignore_control!(BasicBroadcast);
22 impl Provide<BestEffortBroadcast> for BasicBroadcast {
23   fn handle(&mut self, broadcast: Broadcast) -> () {
24     for peer in peers.iter() {
25       peer.tell(BroadcastMsg::new(peer.clone()), self);
26     }
27   }
28 }
29 impl Actor for BasicBroadcast {
30   type Message = BroadcastMsg;
31
32   fn receive_local(&mut self, msg: Self::Message) -> () {
33     let BroadcastMsg(payload) = msg;
34     self.beb.trigger(Deliver(payload));
35   }
36   fn receive_message(&mut self, msg: NetMessage) -> () {
37     let deser_res = msg.try_deserialise::<BroadcastMsg, BroadcastMsg>();
38     if let Ok(BroadcastMsg(payload)) = deser_res {
39       self.beb.trigger(Deliver(payload));
40     }
41   }
42 }
```

Listing 8.7: Mixed BasicBroadcast event and message handlers in Kompact.

header. As the concrete deserialisation type is known at this point, no heap-moves are necessary and the deserialisation result, if successful, can be used immediately. An example of this can be seen in listing 8.7 on lines 36-41.

It should be noted that the serialisation buffer that is passed inside the `msg: NetMessage` instance is provided as mutable, indicating that reads from it may in general be destructive. This allows for efficient deserialiser implementations to potentially reuse the raw bytes directly if they are known to match the deserialised layout. However, this means that it must be assumed by developers, without any additional knowledge about the particular deserialisers involved, that a failed attempt at deserialising from a buffer has left the buffer in an inconsistent state from which a correct message can not be recovered by trying again with a different deserialiser. For this reason, it is recommended to write deserialisers that always return an enum of all the types they can handle, *or* guarantee that any failure for a particular type indicates that all other types would also fail. If such an implementation is not possible, deserialisers must instead ensure that the bytes in the buffer remain unaltered when failing.

```
29 impl NetworkActor for BasicBroadcast {
30   type Message = BroadcastMsg;
31   type Deserialiser = BroadcastMsg;
32
33   fn receive(
34     &mut self,
35     _sender: Option<ActorPath>,
36     msg: Self::Message,
37   ) -> () {
38     let BroadcastMsg(payload) = msg;
39     self.beb.trigger(Deliver(payload));
40   }
41 }
```

Listing 8.8: Mixed BasicBroadcast with NetworkActor in Kompact.

Network Actors Looking again to the actor implementation in listing 8.7, it seems that we are unnecessarily duplicating the deliver code in lines 34 and 39, as well as the code to access the payload, while also being required to state the blatantly obvious fact that we want an instance of `BroadcastMsg` to be deserialised, in the form of type annotations in line 37.

In order to avoid such code duplication in components that handle the same set of types both locally and over network, Kompact provides the convenience trait `NetworkActor`, which automatically provides implementations of `Actor` and `ActorRaw`. It has the following signature:

```
1 pub trait NetworkActor: ComponentLogging {
2   type Message: MessageBounds;
3   type Deserialiser: Deserialiser<Self::Message>;
4
5   fn receive(&mut self, sender: Option<ActorPath>, msg: Self::Message) ->
  ↪  ();
6
7   fn on_error(&mut self, error: UnpackError<NetMessage>) -> ();
8 }
```

In addition to providing an associated type for messages, we must now also specify an associated type for the deserialiser to be used when converting incoming instances of `NetMessage` into instances of `Self::Message`. If this conversion is impossible or simply fails the `on_error` method is invoked, otherwise the deserialised message is passed to the `receive` method. Only for the the latter is an implementation required, as Kompact provides a default implementation for `on_error`, which simply logs the error message as a warning.

An equivalent implementation of the basic broadcast message handling from listing 8.7, but using `NetworkActor` instead, can be seen in listing 8.8. Note that

```
3  pub fn main() {
4    let peers = /* load some Vec<Address> */;
5    let system = KompactConfig::default().build().expect("KompactSystem");
6    let basic_broadcast_c = system.create(|| BasicBroadcast::new(peers));
7    let perfect_link_c = system.create(PerfectLink::new);
8    biconnect_components::<PerfectPointToPointLink, _, _>(
9      &basic_broadcast_c,
10     &perfect_link_c,
11   ).expect("Components failed to connect.");
12   system.start(&perfect_link_c);
13   system.start(&basic_broadcast_c);
14 }
```

Listing 8.9: Setup code for components in Kompact.

we are using the `BroadcastMsg` type both as deserialiser and as deserialisation target, as we did in listing 8.7 on line 37, since the type itself implements the `Deserialiser<BroadcastMsg>` trait.

Raw Actors As was mentioned before, Kompact provides an alternative, lower level API for implementing actor behaviours, which can be accessed by implementing the following trait instead of `Actor` or `NetworkActor`:

```
1 pub trait ActorRaw {
2     fn receive(&mut self, env: ReceiveEnvelope) -> ();
3 }
```

The `enum ReceiveEnvelope` contains alternatives for all local and network messages, which are simply matched and deconstructed by the default `Actor` implementation. However, for certain low-level components, such as proxies or routers, for example, this behaviour can be unnecessarily inefficient. In order to avoid such overhead, these kinds of components can implement the `ActorRaw` trait directly.

8.1.3 Component Creation and Connections

Like in Akka and Kompics, it is required to create a "system" before any components can be created. Such a `KompactSystem` can either be created directly via `KompactConfig::new().build()` with minimal settings, or alternatively using `KompactConfig::default().build()` for default settings. Of course, a configuration instance can be modified before a system is created from it. Such modifications are done via its "builder API", which allows for an option chaining style. Since system creation can fail, due to faulty settings or blocked ports, for example, the system is returned wrapped in a Rust `Result` type, and must be unwrapped before use. This can, for example, be done via the `expect(...)` method, as seen in listing 8.9 on line 5.

```
16 pub fn main() {
17   let peers = /* load some Vec<ActorPath> */;
18   let system = KompactConfig::default().build().expect("KompactSystem");
19   let (basic_broadcast_c, registration_future) =
20     system.create_and_register(|| BasicBroadcast::new(peers));
21   let named_path_registration_future =
22     system.register_by_alias(&basic_broadcast_c, "basic-broadcast");
23   registration_future.wait_expect(WAIT_DUR, "Actor failed to register");
24   named_path_registration_future
25     .wait_expect(WAIT_DUR, "Actor failed to register");
26   system.start(&basic_broadcast_c);
27 }
```

Listing 8.10: Setup code for networked components in Kompact.

Component Creation At the most basic, a component is simply created via a `system.create(...)` invocation, as shown in listing 8.9 on lines 6-7. Note that this method expects a constructor function to be provided, not an already constructed instance. This, of course, also allows passing of a closure, with the desired parameters for the instance being created.

If a component is also meant to be used as an actor for networking, it must be registered with the system's *dispatcher* instance, which handles resolution of `ActorPath` instances to `ActorRef` instances that can actually be sent to. Registration with a *unique* `ActorPath` (one based on the component's unique id) can be done conveniently during creation using the `create_and_register(...)` method instead, which can be seen in listing 8.10 on line 20. In addition to returning the component instance, this method also returns a `Future`, which indicates whether registration succeeded. While registration for freshly created unique paths always succeeds, completion of the future also indicates when sending messages to the newly created `ActorPath` is possible. Messages sent before the future is complete may be rejected by the dispatcher.

In addition to unique paths, components can also be registered to one or more *alias paths*, which follow a Uniform Resource Identifier (URI) scheme, similar to Akka's *actor selection* mechanism [69, section *Remoting*]. This is done via the `system.register_by_alias(...)` method, which returns a `Future`, just like `system.create_and_register()`. This `Future`, however, can realistically fail under normal circumstance, if the alias is already in use, for example.

Generally, addressing messages via a unique path is somewhat faster, but the use of an alias allows abstraction over the concrete instance of the target actor, and thus interacts better with recovery after actor failures.

Port Connections Kompact does not provide runtime port instance lookup methods on components, as Kompics does via `getPostive(T.class)` and `getNegative(`

`T.class)`. Instead, port instances on component structs can be accessed indirectly via the `component.on_definition(...)` and `on_dual_definition(component1, component2, function)` methods. These methods acquire the component locks, before invoking the provided function, allowing guaranteed exclusive access to component state, which includes the port instance fields.

Port instances can be connected in a single direction, using the `connect(&mut self, c: RequiredRef<P>) -> ()` method on a `ProvidedPort<P, Self>` instance or its equivalent on the `RequiredPort<P, Self>` instance. Note that the type signature of the method enforces typing rule T-CONNECT from section 4.2.1.

Connections in both directions can be achieved more conveniently by using the method:

```
1 pub fn biconnect_ports<P, C1, C2>(
2   prov: &mut ProvidedPort<P, C1>,
3   req: &mut RequiredPort<P, C2>) -> ()
```

This is just a convenience utility for creating individual connections in each direction.

However, there is another group of convenience utilities, which are available whenever a component uses the derive macro to generate the required implementation of the `ComponentDefinition` trait. This macro will additionally generate implementations of extractor traits for every port instance field defined on the target struct. It is these extractor traits, which allow us to use the `biconnect_components<Port,_,_>(component1, component2)` method, as seen in listing 8.9 in lines 8-11. This call is really just syntactic sugar for calling the `biconnect_ports(...)` function within the `on_dual_definition(...)` function.

Lifecycle Kompact components follow a similar lifecycle as Kompics components, do. However, as Kompact components are not hierarchical, the intermediate stages of STARTING and STOPPING are not applicable. Just like Kompics, though, components need to be started before they will handle any events or messages. As seen at the end of listings 8.9 and 8.10, this can be achieved with the `system.start(&component)` method.

Sometimes it can be important to know when a component is actually running. This can be achieved conveniently using the `system.start_notify(&component)` method instead, which returns a `Future` that is completed when the component is running, similar to how dispatcher registration works. Exactly the same logic and DSL is also followed for the `stop` and `stop_notify` methods, as well as the `kill` and `kill_notify` methods, which pause or end, respectively, the life of a component.

8.2 Kompact Implementation

The Kompact framework is implemented as a Rust library and is available at `https://crates.io/crates/kompact`. It does currently require a *nightly* build of Rust, due to the use of some unstable features in its implementation.

It is also an open source project under MIT license, which can be found on Github at `https://github.com/kompics/kompact`.

8.2.1 Message and Event Execution

With an abstraction for direct addresses messaging now available in the `ActorRef` type, the default assumption of the Kompact runtime is that all delivered messages and events are going to be handled, as is the case in Actor systems. Thus, Kompact components are scheduled on the configured scheduler implementation for every delivered event or message.

When a component executes, the `Mutex` holding its internal definition struct is locked. Each component has multiple message/event queues it needs to fairly schedule its execution time over. As in Kompics, lifecycle messages are handled before any other messages, and are not affected by the processing count limit imposed by Kompact's equivalent to the $\eta_{\max}$ parameter (cf. section 2.2.2). If there is processing quota remaining after lifecycle events, then events from Kompact's built-in timer facilities are processed with highest priority, as they become stale (outdated) over time.

After the important internal messages have been processed, the remaining quota is split over messages and events from all ports. A new configuration parameter η_{msgs} controls the ratio of messages to events being processed, allowing the programmer to prioritise one over the other, if so desired. However, both $\eta_{\max}$ and η_{msgs} are global to a full Kompact system, and not configured per component, at this point.

Fairness between different declared ports is achieved by storing the index of the last port that was processed before the event limit was reached, and continuing processing from that index during the next invocation, starting over when the last port is handled. However, as was mentioned before, this behaviour can be customised by implementing the `ComponentDefinition` trait, instead of deriving it, allowing "unfair" components to be developed.

Note that it is crucial for type safety and performance that the port execution code is unique for every component definition implementation, as opposed to the approach taken in Kompics, where all ports are treated uniformly in a collection. Adding ports to a collection loses the important information about their individual types — as the collection would clearly have to conform to a common super-trait — and this then necessitates indirect handler invocations, for example via a *trait object*. By generating (or writing) unique port execution code for every component, we can achieve high performance by allowing the compiler to generate efficient code for handler invocation.

8.2.2 System Components

Kompact uses a number of *system components*, which provide fundamental services to a Kompact system:

Deadletter Box is a component that is the target for messages sent to and `ActorPath`, which could not be resolved to an `ActorRef`. It can also be used to provide sender information where message types require it, but it is not available (such as outside a component context).

Dispatcher A minimal *local dispatcher*, simply resolves local `ActorPath` instances to `ActorRef` instances, and forwards messages appropriately. However, Kompact can be started with a default (or custom) *network dispatcher*, which then acts as the interface for networking in Kompact. In our default implementation, the dispatcher manages network channels, as well as sending and delivery of messages, using the *Spaniel*[4] networking library, which was written for Kompact by Johan Mickos as part his Master's thesis [81].

Timer A timer with millisecond accuracy, based on a *Hash Wheel* [106], is provided as part of Kompact by default. It can be used to schedule the execution of custom handling code on a component, in both a "one-off"-style and in a repeating manner. However, sometimes developers may wish to replace the default timer with a custom variant, if different resolutions or implementations are desired, and this is indeed supported, as well.

Supervisor Instead of having a component hierarchy like Kompics, or an Actor hierarchy like Akka and Erlang, Kompact simply has a flat component space, managed by a single *Supervisor* component. This is a result of different assumptions about failure behaviour in our design space. While Kompics, Erlang, and Akka are designed for dynamic systems, where behaviours change at runtime and failures may be common in certain parts of the hierarchy, Kompact is designed for relatively static deployments, where failures are either handled locally within a handler, or propagate to terminate the Kompact system, if not the whole process it is running in. This outlook also to some degree reflects Rust's approach to failure handling, where exceptions (or "panics") are usually fatal.

Custom implementation for the *Deadletter Box*, *Dispatcher*, or *Timer* can be provided to the `KompactConfig` before creation of a `KompactSystem`. The *Supervisor* can not be customised, however, as it is integral to correct lifecycle behaviour.

8.2.3 Component Execution

Kompact components are executed on a scheduler that is associated with the Kompact system instance they are created in. Kompact supports different mechanisms for scheduling individual components, as well as different scheduler implementations to be "plugged in".

[4] `https://crates.io/crates/spaniel`

8.2.3.1 Scheduling

Like in Kompics (see section 2.2.2.1), Kompact components are also only scheduled when they have outstanding work. In particular, there are only two concrete conditions that cause a component to be added to the scheduler's work-queue:

1) The component's work-count variable is incremented from 0 to 1 while adding an event or message to one of its queues. In this case the thread adding the event or message will invoke the scheduling logic.

2) The component's work-count variable is decremented after completing its quota $\eta_{\max}$, but outstanding work remains, thus the value after the decrement is greater than 0. In this case the scheduling logic will be invoked from the thread the component was just running on.

This strict mechanism guarantees that a component is never scheduled concurrently on two different threads.

In Kompics this guarantee is correctness relevant, as a component's internal state is not protected from concurrent accesses in any other way, while in Kompact it is only a performance optimisation. As was described in the paragraph about connecting components, section 8.1.3, the internal state of a Kompact component is indeed protected by a `std::sync::Mutex`, guaranteeing exclusive access. However, a component that is scheduled on two different threads would cause the thread arriving at the `Mutex` second to block until the first thread relinquishes its hold on the lock. This would prevent the second thread from making progress with another component, thus reducing overall throughput of the Kompact system. The work-count tracking mechanism described above prevents this problem from occurring.

8.2.3.2 Schedulers

Like Kompics and Akka, Kompact components are typically run on a configurable thread-pool scheduler, which can be selected during creation of the Kompact system. However, when we began developing Kompact there was no Rust library that provided a scheduler (also sometimes referred to as an "executor"), that was designed for the particular load characteristics of an actor or component system. Message-passing system, and in particular Kompact, with the scheduling logic as described in the previous paragraph, exhibits the following load characteristics:

1) Most scheduling events are triggered from within the thread-pool, that is by one of its threads.

2) External scheduling, from threads outside the thread pool, is mostly limited to a small number specific external threads, such as a networking thread, for example.

3) As rescheduling components is common under high load, thread-locality can avoid cache-misses, especially when the number of components is similar or smaller than the number of available threads in the pool.

4) Work-loads can shift over time, requiring rebalancing of work between threads.

At the time we started writing Kompact, existing schedulers in Rust mostly employed a single global queue, often even with a `std::sync::Mutex` on the receiving side, resulting in extremely poor multi-threaded performance, often even worse than running the same code on a single thread. At the same time on the JVM the `java.util.concurrent.ForkJoinPool` scheduler had been available for many years and its design lent itself extraordinarily well to our requirements. For this reason both Akka and all Kompics implementations make use of the `ForkJoinPool` scheduler.

In order to achieve our design requirements of scaling performance on large CPUs, such as those commonly found in commodity clusters used for data analytics, we had implement our own work-stealing thread-pool scheduler. The current implementation is based around thread-local, work-stealing queues combined with a global queue for externally scheduled components. The queue design itself is provided by Stjepan Glavina as part of the excellent *crossbeam* library[5], and amortises queue memory management and concurrency overhead by stealing in whole segments instead of individual entries, among other perks. This design addresses our work-load very well, and in particular the thread-local approach addresses characteristics 1 and 3, while work-stealing addresses number 4. The segment stealing approach is also applied when taking items from the global queue, which to some degree addresses characteristic 2. Our implementation is published as a separate open-source library[6], as it is in no way particular to Kompact and, in fact, provides multiple different scheduler (or "executor") implementations, suited for different use cases.

8.2.3.3 Dedicated Components

In a component system scheduled on a fixed-size pool, fairness relies on individual components relinquishing control of their thread reliably and regularly, as the system can not actively interrupt them, in the way the OS scheduler can. Both Kompics and Kompact employ a maximum event count ($\eta_{\max}$) mechanism to achieve this fairness effect in the absence of preemption. This approach, however, relies on the assumption that the work performed per event is approximately equal and relatively short.

A common violation of this assumption occurs when a component is used to perform blocking input/output (I/O), such as reading a file from disk, for example. Not only is this process potentially lengthy, if the file is large, but it can also stall the thread performing it frequently, if the file is stored on a spinning magnetic disk, or

[5] `https://github.com/crossbeam-rs/crossbeam`

[6] https://crates.io/crates/executors

Table 8.1: Benchmark implementation variants for Kompact.

Name	Symbol	Actor	Component	Mixed
Local Benchmarks				
Ping Pong	`PINGPONG`	✓	✓	X
Throughput Ping Pong	`TPPINGPONG`	✓	✓	X
Fibonacci	`FIBONACCI`	✓	X	X
Chameneos	`CHAMENEOS`	✓	X	✓
All-Pairs Shortest Path	`APSP`	✓	✓	X
Distributed Benchmarks				
Net Ping Pong	`NETPINGPONG`	✓	X	X
Net Throughput Ping Pong	`NETTPPPINGPONG`	✓	X	X
Atomic Register	`ATOMICREGISTER`	✓	X	✓
Streaming Windows	`STREAMINGWINDOWS`	✓	X	X

perhaps even on a mounted network device. It can easily be seen that this violates fairness. Consider an extreme example, where the component system only runs with a single thread in its pool. In this case, no component can make any progress until the I/O component has completed its read, even if the reading is actually stalled.

Since performing blocking I/O is occasionally necessary when interacting with other libraries or the filesystem in general, Kompact allows components to optionally be spawned on a dedicated thread by themselves, instead of the shared thread-pool. This is achieved via the `system.create_dedicated(...)` function, and an equivalent is available for system components, such as the network dispatcher, as well. This feature, however, should be used with care, as dedicated components do not benefit from any of the scheduling optimisations described in the previous section and can cause frequent OS preemption to occur on a loaded system with too many threads.

8.3 Evaluation

We have also performed the experiments described in section 6.4 for Kompact version `0.8.1`, the most recent at the time of writing. Due to its hybrid model, Kompact sometimes allows multiple fundamentally different implementations of the same benchmark (see section 6.2 for details on the benchmarks themselves). In order to evaluate advantages of the hybrid model, we have thus provided up to three different Kompact implementations of benchmarks, where this was both possible and sensible:

Kompact Actor provides implementations that only use the direct message communication via instances of `ActorRef` or `ActorPath`, but does not use ports, except for lifecycle events.

Kompact Component is the opposite of Kompact Actor, and only uses events and ports for communication. Since Kompact's networking relies on its actor

API, there are no implementations of distributed benchmarks for Kompact Component.

Kompact Mixed uses the full API of Kompact, but is only explicitly marked as such where the implementation actually differs from Kompact Component and Kompact Actor. Where this is not the case, Kompact Mixed performance can be assumed to be equivalent to the better of the Kompact Actor or Kompact Component implementations.

For convenience, table 8.1 provides an overview of which benchmarks variants are implemented for Kompact.

While pure component implementations for the Chameneos and Fibonacci benchmarks are technically possible, as shown by the fact that all Kompics versions have implementations for them, they are so inefficient compared to direct addressing, that they have been elided.

As many of the benchmarks are very simple and rely on a single communication pattern, the room for exploiting Kompact Mixed implementations was somewhat limited, resulting in only two provided implementations: In the Chameneos benchmark, the mall communication has been implemented as a port pattern, while the inter-chameneo communication uses messages. In the Atomic Register benchmark, the broadcast abstraction has been separated out and communication between the register and the broadcast component uses ports.

As was explained in section 6.4.2, we must limit ourself to describing a mere selection of the full benchmark results in this dissertation. As before, the full raw data and all generated graphs can be found online at:
`https://kompics.github.io/kompicsbenches`

8.3.1 Local Benchmarks

In this section we describe the benchmark results for Kompact, which do not include its networking library. The results are compared to those of the other frameworks already described in section 6.4.3.

8.3.1.1 Ping Pong

Figure 8.2 shows the results of the Ping Pong benchmark, including two Kompact implementations: Kompact Actor and Kompact Mixed. While previously Erlang dominated this benchmark, it has now been replaced by both Kompact implementations. At the highest parameter setting, Kompact Component is on average 22 % faster than Erlang, and about 7 % faster than Kompact Actor. As before the next best performer is Actix, another Rust framework, which is 2.81 × slower than Kompact Component at the same parameter setting.

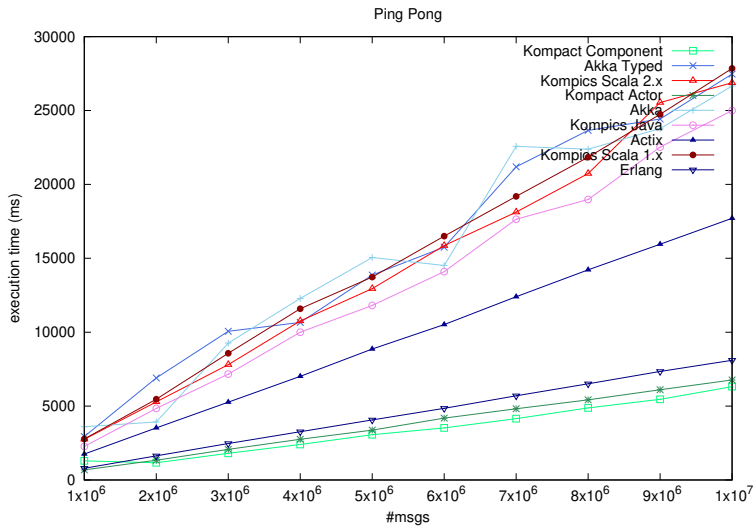
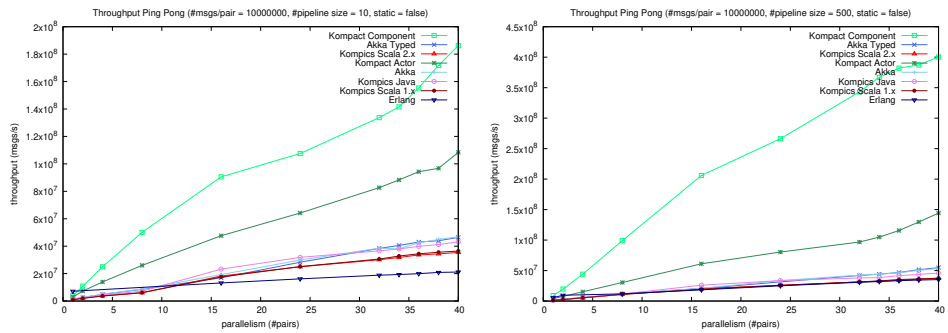


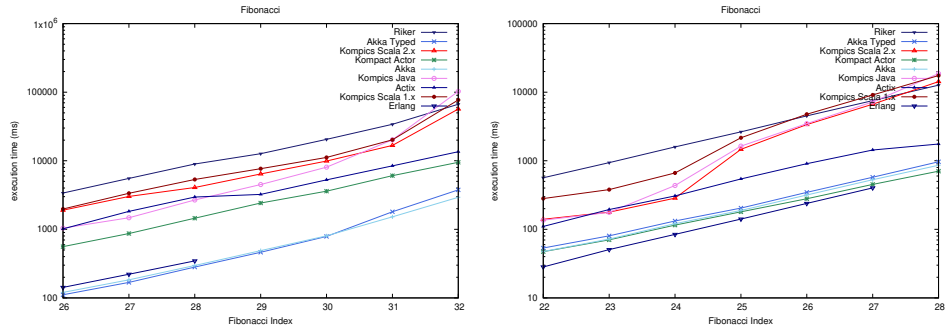
Figure 8.2: Results of the Ping Pong benchmark including Kompact in the AWS setup. For readability, Riker has been excluded from the plot. Results show total execution time in milliseconds, thus lower is better.



(a) Low pipelining.

(b) Deep pipelining.

Figure 8.3: Results of the Throughput Ping Pong benchmark including Kompact on AWS. Riker has again been excluded from the plot, due to poor performance and incomplete results. Both plots show results with allocations per message, the results without allocations were very similar. Results show the average throughput of the run in messages per second, thus higher is better.



(a) The AWS setup.

(b) The Desktop setup.

Figure 8.4: Results of the Fibonacci benchmark including Kompact. Results show total execution time in milliseconds, thus lower is better.

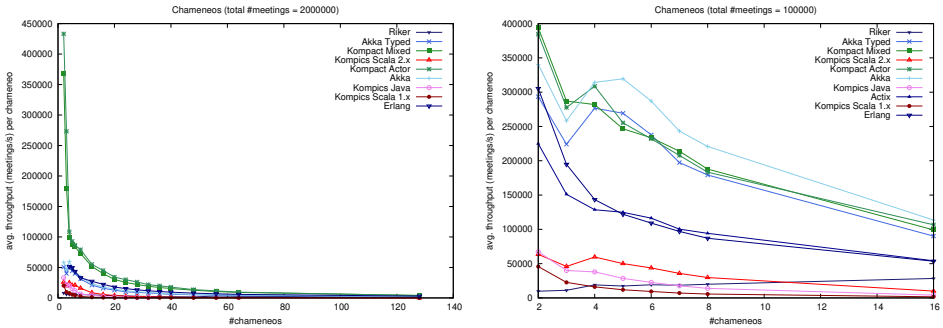
8.3.1.2 Throughput Ping Pong

The results of the Throughput Ping Pong benchmark for Kompact Actor and Kompact Component are shown in figure 8.3. At first glance it is immediately clear how Kompact, and in particular Kompact Component, dominates this particular benchmark. Especially in the scenario with deep pipelining, figure 8.3b, the scaling of Kompact Component performance is well beyond any other tested implementation, including Kompact Actor, and peaks at over 400 million messages per second. Not only is that 2.77 × the performance of the next best implementation, Kompact Actor, but also 7.31 × the performance of the fastest non-Kompact implementation, which is Akka Typed at just under 55 million messages per second.

In the low pipelining scenario, figure 8.3a, the differences are not quite as large, but the ranking of the frameworks mostly holds, except at very low parallelism, where Erlang actually outperforms both Kompact Component and Kompact Actor by around 2 ×. This confirms our previous observation, that Erlang's runtime is in general very good and well-optimised, but suffers from comparatively poor SMP support.

8.3.1.3 Fibonacci

Figure 8.4 shows the results of the Fibonacci benchmark, both on AWS and in the Desktop setup. For this benchmark we only implemented Kompact Actor, as the component communication would certainly have shown more overhead. In both test setups, Kompact performs consistently faster than the slower group, but significantly slower than the fastest implementation. In fact, in the Desktop setup Kompact is actually slightly faster than the Akkas, which continue to dominate in the AWS setup.



(a) The AWS setup.

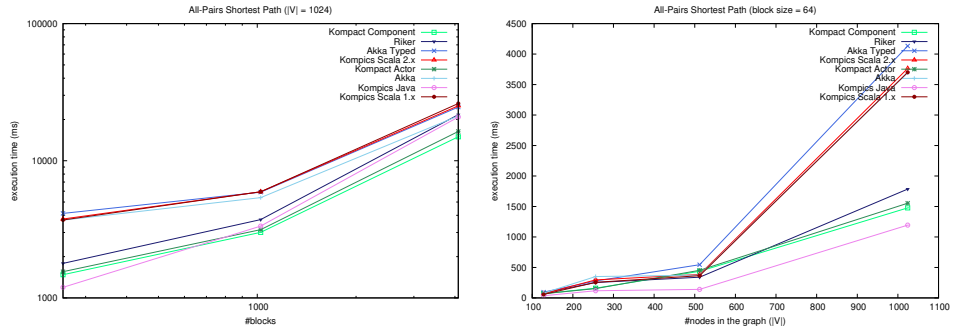
(b) The Desktop setup.

Figure 8.5: Results of the Chameneos benchmark including Kompact. Results show throughput in number of meetings per second per chameneo, thus higher is better.

These results are consistent with our expectations, as we trade steady-state performance for a somewhat increased component creation overhead. In other words, while Kompact components are not necessarily very expensive to create, they are by no means so cheap as to be considered disposable, which is common for pure actor systems, like Akka and Erlang. As our implementation targets relatively static component systems, such as fixed-deployment continuous streaming applications, this trade-off appears to be well chosen.

8.3.1.4 Chameneos

As can be seen in figure 8.5, Kompact performs very competitively on the Chameneos benchmark as well. While the Akkas show similar, if not better, performance in the Desktop setup, they fall behind significantly on AWS, which consists of longer runs, and thus likely stressed the JVM's garbage collector more. While Kompact by no means even approaches the idealised implementation of showing no performance degradation until there are as many chameneos as CPU cores, it does show higher performance than any other testee framework at all parameter settings on AWS. At the minimum number of 2 chameneos, this translates to 433 128 meetings per second per chameneo for Kompact Actor, and closely behind 367 505 meetings per second for Kompact Mixed. The next fastest is Akka with 58 376 meetings per second per chameneo, leaving Kompact Actor about 7.41 × faster at this setting. At 36 chameneos, exactly the number of real cores on the AWS machine, the performance of Kompact Actor has shrunk to 19 857 meetings per second per chameneo, and to 17 043 for Kompact Mixed. At this point the next fastest framework is actually Erlang, which did not converge in the 2 chameneo scenario, but manages to perform around half as many meetings Kompact Actor, at 10 413 meetings per second per chameneo.



(a) 1024 nodes.

(b) 64 nodes per block without Erlang.

Figure 8.6: Results of the All-Pairs Shortest Path benchmark on AWS including Kompact. Results show the total execution time of the run, thus lower is better.

The results show that the hybrid Actor/Component communication did not provide any advantage in this scenario, but, in fact, was usually slower than the pure Actor implementation. The reason for this is most likely that there is some overhead associated with mixing multiple mailboxes on the same component, as is the case when using both actor and component communication, and in this benchmark the generally better performance of the component communication was not sufficient to balance out this overhead.

8.3.1.5 All-Pairs Shortest Path

In figure 8.6 we can the results of the All-Pairs Shortest Path benchmark (see section 6.2.1.5). As we already discussed in section 6.4.3.5, Java does particularly well on this benchmark, while Rust appears to struggle with the computational part of the task. This clearly also affects our two Kompact implementations, which perform worse than Java, and at some parameters even worse than Scala, as seen in figure 8.6b for graphs with 512 nodes. However, we can see in figure 8.6a that the relative performance of Kompact improves as the total number of blocks grows, and by 4096 blocks Kompact Component is the fastest implementation, followed by a 9 % slower Kompact Actor, and only then Kompics Java, which was on average 28 % slower.

This again reinforces our notion that is benchmark is heavily influenced by the performance of the underlying language and runtime, while the message-passing performance only becomes relevant at very large numbers of blocks, which no real-world application would ever want to use, due to the resulting overall performance being very poor.

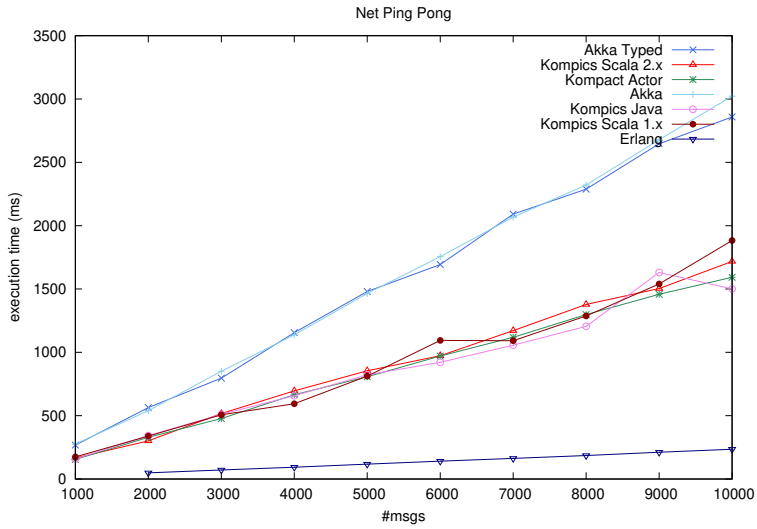


Figure 8.7: Results of the Net Ping Pong benchmark on AWS including Kompact. Results show total execution time in milliseconds, thus lower is better.

8.3.2 Distributed Benchmarks

In this section, we investigate how Kompact's networking library performs compared to the other testee frameworks'.

It should be reiterated that all experiments are run over the local loopback network device, as was mentioned in section 6.4.1 already.

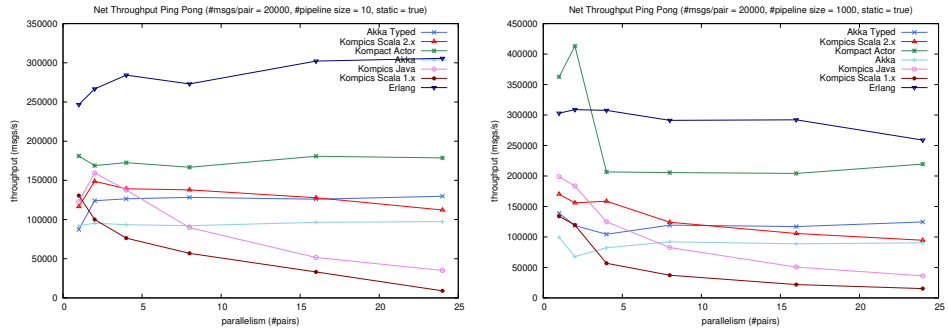
Furthermore, pure Kompact Component implementations do not appear in this section, because Kompact's networking library uses messages, not events.

8.3.2.1 Net Ping Pong

As described in section 6.2.2.1, the Net Ping Pong benchmark measures pure networking overhead, with no pipelining optimisations being applied. Kompact's performance on this benchmark is mediocre, falling into the same group as all the Kompics implementations, albeit with somewhat more consistent results, most likely due to the lack of garbage collection overhead. It is very clear from the large gap to Erlang here that much work to optimise the network library remains to be done.

8.3.2.2 Net Throughput Ping Pong

In figure 8.8 we can see the results of the Net Throughput Ping Pong benchmark (see section 6.2.2.1). While Kompact again performs very competitively, it still can not



(a) Low pipelining. (b) Deep pipelining.

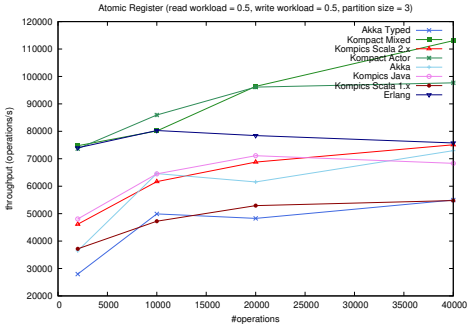
Figure 8.8: Results of the Net Throughput Ping Pong benchmark on AWS including Kompact. Results show throughput in number of messages per second, thus higher is better.

compete with Erlang for the most part. An exception to this are the low parallelism scenarios with deep pipelining, figure 8.8b, where Kompact outperforms Erlang by up to 35 %. This clearly points to the potential of the implementation, if it were as optimised as Erlang has become over its many years of development. However, for the the vast majority of settings, including all settings with shorter pipelining as shown in figure 8.8a, Erlang outperforms Kompact — and thus everyone else — by up to 71 %.

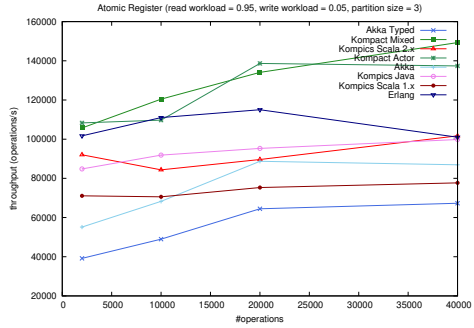
8.3.2.3 Atomic Register

Moving on from micro-benchmarks, figure 8.9 shows the results of the Atomic Register benchmark (see section 6.2.2.3), where the picture certainly is not as clear as in the previous two sections. This benchmark challenges the network implementations in very different ways than Net Throughput Benchmark does, as now communication must be efficient across multiple parallel channels, and as the algorithm relies on rounds of communication, latency is not completely irrelevant either. Additionally, as the number of operations — and with them the size of the key-value-store — grows, local access efficiency becomes increasingly performance relevant.

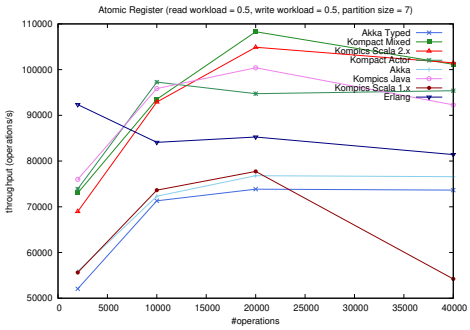
In small setups with only 3 nodes, we can see in figures 8.9a and 8.9b that Kompact eventually dominates, as with the growth of the store Erlang gets left behind. In fact, as the store grows Kompact Mixed also begins to leave Kompact Actor behind. This is likely because in Kompact Mixed the message broadcasting is handled by a separate component from the one performing key-value-store accesses, which allows the implementation to parallelise those two tasks. While this comes with some overhead, it becomes valuable as the store becomes a performance



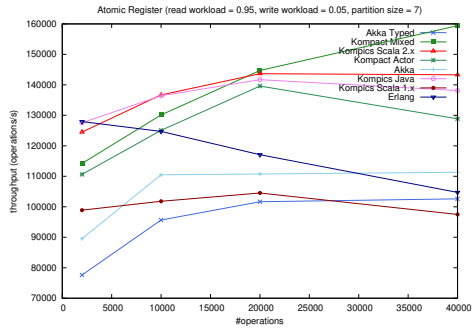
(a) 50/50 Reads/Writes with 3 nodes.



(b) 95/05 Reads/Writes with 3 nodes.



(c) 50/50 Reads/Writes with 7 nodes.



(d) 95/05 Reads/Writes with 7 nodes.

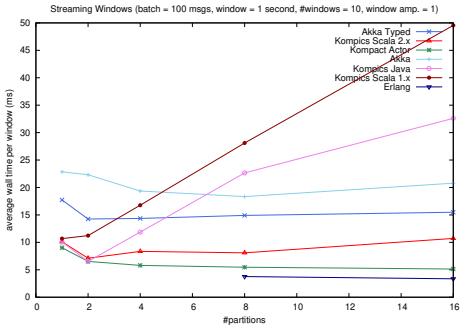
Figure 8.9: Results of the Atomic Register benchmark on AWS including Kompact. Results show the average throughput of the run in operations per second over all nodes, thus higher is better.

bottleneck. This particular trend is shared at all settings and can be seen in all graphs in figure 8.9.

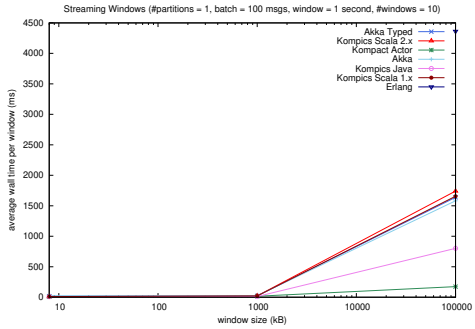
In larger setups, such as the 7 node cluster shown in figures 8.9c and 8.9d, we can see that Kompics Scala `2.x` and Kompact Java challenge, and sometimes outperform, Kompact. It is highly likely that this extra performance is provided by their multi-threaded networking library, which can parallelise work over multiple target connections in ways that Kompact can not. We do, however, plan to add multi-threaded networking in a future Kompact version, to support exactly such many-to-many communication workloads.

8.3.2.4 Streaming Windows

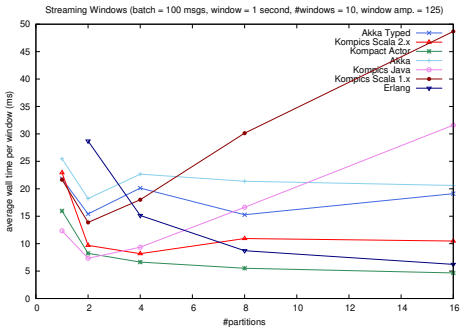
The last benchmark we ran was Streaming Windows (see section 6.2.2.4). As we mentioned in the introduction to this chapter, continuous streaming operations



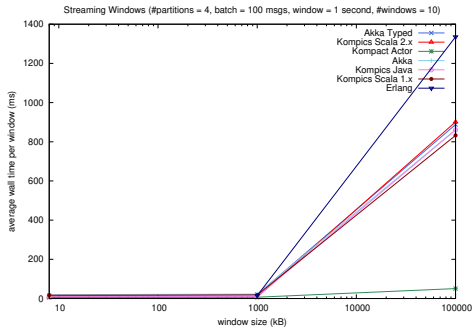
(a) 8 kB window size.



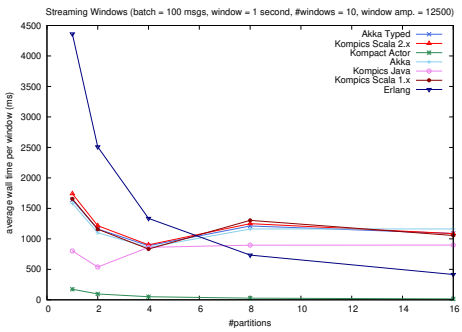
(b) Single partition.



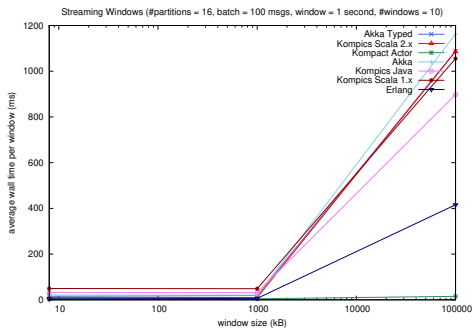
(c) 1 MB window size.



(d) 4 partitions.



(e) 100 MB window size.



(f) 16 partitions.

Figure 8.10: Results of the Streaming Windows benchmark in the AWS setup including Kompact, with a batch size of 100 events and a total of 10 windows per partition. The left column shows scaling over the number of partitions for a fixed window size, while the right column shows performance for increasing window sizes for a fixed number of partitions. All results show the average time taken to complete a window, thus lower is better.

such as the one used in this benchmark are what Kompact was specifically designed for.

In figure 8.10 we can see that for the majority of settings Kompact performs very well on these kinds of problems. While it does lag about 35 % behind Erlang for very small windows, which are mostly dominated by networking throughput, as seen in figure 8.10a, Kompact outperforms Erlang, and every other framework, at almost all other settings. Kompics Java is still competitive at low parallelism in figure 8.10c, but beyond that for higher parallelism and larger window sizes Kompact outperforms its peers by a large margin. Looking at the extremes, for windows of 100 MB in size, figure 8.10e, and at a parallelism of 16, Kompact is more than 27 × faster than the next fastest framework, which is Erlang.

Looking at figures 8.10b,8.10d, and 8.10f, it is also worth noting that Kompact shows comparatively little impact from the increase in window size. For example, at 16 partitions, the *total* execution time for 100 MB windows is only 3.24 × longer than that for 1 MB windows. In contrast, for Kompics Java, for example, the execution time increases by 28.41 × between the same settings.

Clearly, Kompact benefits heavily from efficient support for the window management and aggregation operations used in this benchmark, which are provided by the Rust standard library and its compiler. This, combined with a good, if not stellar, networking library, gives these impressive benchmark results.

8.3.3 Summary of Results

To summarise what we have seen in the past section, it is clear that Kompact's local runtime shows very good performance on the majority of the local benchmarks. In particular, it scales very well across larger thread-pools, resulting in the impressive outcome of over 400 million messages per second in the Throughput Ping Pong benchmark. However, there have also been two weaker local benchmarks: On the one side, component creation clearly comes with some significant overhead, and does not optimise well over multiple cores, as it does in Erlang or Akka, for example. Furthermore, while Kompact often benefits from the Rust compiler's optimisations, `rustc` can also sometimes hold Kompact back, as we saw in the All-Pair Shortest Path benchmark.

When we looked to distributed benchmarks, it became clear that Kompact's network library, while competitive, still shows unnecessarily high overheads, which cause it to be surpassed by Erlang and sometimes even by Kompics implementations, depending on the scenario. However, some good runs in the Net Throughput Ping Pong, as well as generally strong results in the Atomic Register and especially in the Streaming Windows benchmark show that the concept of Kompact's networking is sound. It simply requires better fine tuning, in order to perform more consistently across a wider range of scenarios.

8.4 Related Work

Clearly, Kompact has drawn inspiration from a number of message-passing systems. The most obvious are, of course, the relations to Actor model implementations, as well as Kompics implementations, which Kompact purposefully combines into a powerful hybrid model.

In particular, Kompact's actor API heavily resembles that of Akka Typed [71], as well as those of Actix [96] and Riker [88], although the latter is technically a younger project than Kompact and had little influence on its design. We also find earlier approaches to typed actors, such as TAkka [46], which explored the feasibility of statically typed actors, in particular in Scala.

The inspiration for the statically typed component API mostly came from Kola [62], however with the additional new limitation of having only a single type in each direction. This limitation was introduced to allow the expression of the type constraints from section 4.2.1 within the Rust type system.

The current design actually makes both event-based and message-based communication quite similar to the typed channels found in the Go language [40].

The integration of Kompact's network library into the runtime was partially inspired by the design of Akka and Kompics, however Rust's inefficient reflection capabilities necessitated some novel choices, such as deserialising messages lazily on the receiving component.

The design of the network protocols themselves, on the other hand, was heavily inspired by Google's QUIC protocol [56], as well as the HTTP/2 protocl [13]. More details on this library can be found Johan Mickos' Master's Thesis [81].

As was already noted in the introduction, the theoretical approach of combining channel and actor models is reminiscent of the work by Fowler et al. showing that typed channels and typed actors are semantically interchangeable [35].

CHAPTER

9

Conclusions

Why do you go away? So that you can come back. So that you can see the place you came from with new eyes and extra colours. And the people there see you differently, too. Coming back to where you started is not the same as never leaving.

– Terry Pratchett, A Hat Full of Sky

Over the course of the preceding chapters, we have studied a variety of concerns relating to the design and implementation of programming tools for writing parallel and distributed systems. While writing such systems will most likely never be a trivial concern for a programmer, we dedicated the body of our work to making such endeavours less error prone, more expressive, and often simply faster or more resource efficient. We also hope that we have perhaps made it just a little more enjoyable for programmers, be they students new to distributed systems, or veterans of the field.

We began by taking a close look at programming ergonomics, particularly with a focus on education, in chapter 3. This resulted in a concise Scala DSL, closely modelled to follow the descriptions of distributed algorithms, such as those used by common textbooks. As we showed, not only does the new DSL for our venerable Kompics system improve readability and reduce accidental errors, but it also turned out to often show competitive performance with the more verbose Java variant.

We then continued investigating ways to reduce and avoid sources of programming mistakes, both those that can be identified at compile time, and those that are limited to being discovered during program execution. In chapter 4, we introduced typing rules for the Kompics model, which we then implemented in two different systems: Kola, a compiler frontend for Kompics Java, and Kompact, a Rust implementation, which also includes, but is not limited to, the Kompics component model. We followed up on that in chapter 5, with a DSL designed to ease unit testing for general message-passing systems. Our approach to exterminating execution time bugs has been used to implement a unit testing framework specific to the Kompics framework in Java.

Having dealt with programming ergonomics and programming error reduction, we subsequently turned our attention towards runtime performance of message-passing systems. Chapter 6 introduced the, to the best of our knowledge, first cross-language message-passing benchmarking suite that supports both local and distributed scenarios. By using this framework to evaluate a number of popular actor systems, as well as the systems we ourselves had worked on previously, we identified a number of shortcomings. In particular, these included relatively poor throughput on large thread pools, as well as inconsistent results of many frameworks between local performance, networked performance, and performance of the host language and standard library. That is to say, no framework performed very well across all different problem domains, and often a framework that stood out far ahead of the "pack" in one test, was just as far behind everyone else in another.

Before we decided to address these issues, we first expanded on them in chapter 7, by noting that the availability of network protocol choice, for example, can have a major impact on the performance of distributed abstraction that can later be built on top of a framework. As a concrete example, we showed how a simple file transfer problem can be accelerated significantly compared to the state-of-the-art, when transport protocol selection can be performed dynamically over the course of the

transfer operation, adapting to changing network conditions.

Finally, we addressed the issues described in chapters 6 and 7, as well as incorporated the static typing from chapter 4, into a new and performance-oriented message-passing framework, which we named Kompact. Not only is Kompact, to the best of our knowledge, the first actor-component hybrid model, but our evaluation also showed that it performed extremely well in a wide variety of scenarios. It sometimes outperformed the next best peer by almost 30 ×, and still always performed competitively even when not playing to its strengths.

9.1 Discussion & Future Work

No thesis is ever truly complete, and there always remains work where we feel we could have done more, or simply better, if time had allowed. In this section we will discuss a few limitations of the work presented in this dissertation, and some ideas of how it could be improved.

9.1.1 Kola Tooling

A new programming language, even if closely related to an existing one, is practically unusable without the availability of proper tooling, such as syntax highlighting for popular editors, incremental compilation and type checking, and consistent formatting tools, and other pieces for integrated development environment (IDE) support. Kola basically has none of these things, except a somewhat rushed syntax highlighter for Pygments [1], which was written specifically for this dissertation and is not even publicly available this point.

Additionally, the newly increased pace of development of the Java standard by Oracle has made it particularly difficult for us to keep `kolac` up to date and able to support the latest Java features, to the point that we have essentially given up on maintaining it, due to lack of resources. If there is a lesson to be learned here, it is that writing a S2S compiler for a moving target is certainly a rather risky and time intensive endeavour and should be weighed very carefully against alternative options.

9.1.2 Multi-Unit Testing

While the KompicsTesting framework works well for testing a single unit, be that a single component or a single hierarchy of nested components communicating through a single interface, this is not always sufficient to test arbitrary properties of interest. Especially when it comes to distributed algorithms, many properties of import are defined more globally on whole system executions, which our framework is not designed to capture. An example of such a property would be linearisability,

[1] `https://pygments.org/`

which requires execution sequences from all clients to be compared in order to find violations.

A possible approach for testing such properties would be to allow the description of testing automata which run through multiple execution sequences in parallel, but may share certain state, such as a global time stamp, for example, similar to how our existing simulation framework approaches this problem. However, it is unclear at this point what good semantics for such an approach would be, and the investigation of this issue would certainly be an interesting vector for further research.

9.1.3 More Benchmarks and Frameworks

As the saying goes — "more is always better" — especially when it comes to benchmarking. We would have liked to have the full Savina suite available locally, and a number of different distributed algorithms to test a larger variety of workloads. But as it stands, we already have around 80 different implementations, and especially the distributed ones are often challenging to write in a comparable manner across different languages, requiring multiple design iterations between benchmarking runs.

That being said, we would like to see the MPP suite becomes a standard and gain support from more different frameworks by supplying their own pull-requests with implementations that use their own frameworks to the full extent of their capabilities.

9.1.4 Networking Customisation

While explicit networking and being able to choose protocols on a per message basis already are a huge boon for performance-critical applications, even more could be gained by offering additional customisation options.

In particular on high BDP links, the ability to ignore transport level buffers, and instead re-request messages via callbacks, for example, could present a huge boon to performance and memory management in applications that can tolerate out-of-order delivery. Much of the performance of a modern BitTorrent [24] client, for example, comes from the ability to request packages again, if they did not get delivered, instead of waiting for in-order delivery.

Another crucial feature necessary for custom protocols would be access to high-accuracy timers, which Kompics currently does not provide. The delay between the triggering of a timer and the subscribed component being scheduled and executed often ranges in the multi-millisecond range, which is too large to use for accurate flow-control, for example.

9.1.5 Kompact Async/Await Integration

While our evaluation in section 8.3.2 already showed that Kompact's networking library needs some work, there is additional work on the API to be done as well. In particular, with Rust recently stabilising the *Async/Await* feature, which this humble author would prefer to call the *Futures* system, it will both be interesting and necessary to support interaction with libraries written in this paradigm. However, languages without built-in support for generating continuations generally present a challenge to incorporating the awaiting of non-message-based asynchronous results within a message or event handler.

The challenge at hand, of course, is the question about appropriate semantics when the code that uses the result of the Future accesses, or even updates the internal state of the component that created the Future. In Akka's "ask"-paradigm, for example, a one-off actor is started to process the result of every Future created by calling `ask()` on an `ActorRef`. This means, however, that any closed over state from the original actor may be accessed in a thread-unsafe concurrent manner! These semantics are highly unintuitive to the programmer, because the one-off actor creation basically is an implementation detail.

For Kompact, we envision a Futures system that has intuitive semantics, where state accesses are properly synchronised with the parent component and the runtime can properly coordinate driving Futures to completion, with executing related component handlers. How exactly the resulting semantics and implementation will look, is the subject of future research.

APPENDIX

A

Syntax of Kola Language Extensions

"Begin at the beginning," the King said, very gravely, "and go on till you come to the end: then stop."

– Lewis Carroll, Alice in Wonderland

In this appendix we describe the exact syntax of the language extensions made to Java by the Kola DSL. The notation used is Backus–Naur form (BNF), where nonterminals are given as ⟨*NonTerminal*⟩ and terminals as `'terminal'`. Expressions in brackets are optional. All nonterminal symbols, that are not explicitly described here, refer to constructs from the Java Language Specification [41].

⟨*EventDeclaration*⟩ ::= [⟨*Modifiers*⟩] `'event'` ⟨*Identifier*⟩ [⟨*TypeParameters*⟩] [⟨*HeaderFields*⟩] [`'extends'` ⟨*ClassType*⟩] [⟨*Interfaces*⟩] [⟨*ClassBody*⟩]

⟨*HeaderFields*⟩ ::= `'('` [⟨*FormalParameterList*⟩] `')'`

⟨*PortDeclaration*⟩ ::= [⟨*Modifiers*⟩] `'port'` ⟨*Identifier*⟩ ⟨*PortBody*⟩

⟨*PortBody*⟩ ::= `'{'` ⟨*PortBodyDeclaration*⟩* `'}'`

⟨*PortBodyDeclaration*⟩ ::= `'indication'` `'{'` ⟨*EventType*⟩* `'}'`
| `'request'` `'{'` ⟨*EventType*⟩* `'}'`

⟨*ComponentDeclaration*⟩ ::= [⟨*Modifiers*⟩] `'componentdef'` ⟨*Identifier*⟩ [⟨*TypeParameters*⟩] ⟨*ComponentBody*⟩

⟨*ComponentBody*⟩ ::= `'{'` ⟨*ComponentBodyDeclaration*⟩* `'}'`

⟨*ComponentBodyDeclaration*⟩ ::= ⟨*ClassMemberDeclaration*⟩
| ⟨*InstanceInitializer*⟩
| ⟨*StaticInitializer*⟩
| ⟨*ConstructorDeclaration*⟩
| ⟨*InitDeclaration*⟩
| ⟨*PortFieldDeclaration*⟩
| ⟨*ChildDeclaration*⟩
| ⟨*HandlingDeclaration*⟩

⟨*PortFieldDeclaration*⟩ ::= `'requires'` ⟨*PortType*⟩ ⟨*Identifier*⟩ `';'`
| `'provides'` ⟨*PortType*⟩ ⟨*Identifier*⟩ `';'`

⟨*ChildDeclaration*⟩ ::= `'component'` ⟨*ComponentType*⟩ [⟨*ComponentInitialization*⟩] ⟨*Identifier*⟩ `';'`

⟨*ComponentInitialization*⟩ ::= `'('` [⟨*ArgumentList*⟩] `')'`

⟨*InitDeclaration*⟩ ::= [⟨*Modifiers*⟩] 'init' [⟨*HeaderFields*⟩] ⟨*ConstructorBody*⟩

⟨*HandlingDeclaration*⟩ ::= ⟨*HandleDeclaration*⟩
| ⟨*HandlerDeclaration*⟩
| ⟨*ConnectStatement*⟩
| ⟨*SubscribeStatement*⟩
| ⟨*DisconnectStatement*⟩
| ⟨*UnsubscribeStatement*⟩

⟨*HandleDeclaration*⟩ ::= 'handle' ⟨*Identifier*⟩ '=>' ⟨*PortIdentifier*⟩ ':' ⟨*EventType*⟩ ⟨*EventIdentifier*⟩ ⟨*Block*⟩

⟨*HandlerDeclaration*⟩ ::= 'handler' ⟨*Identifier*⟩ ':' ⟨*EventType*⟩ ⟨*EventIdentifier*⟩ ⟨*Block*⟩

⟨*SubscribeStatement*⟩ ::= '!subscribe' ⟨*HandlerRef*⟩ '=>' ⟨*PortRef*⟩ ';'

⟨*UnsubscribeStatement*⟩ ::= '!unsubscribe' ⟨*HandlerRef*⟩ '=>' ⟨*PortRef*⟩ ';'

⟨*ConnectStatement*⟩ ::= '!connect' ⟨*Expression*⟩ '=>' ⟨*Expression*⟩ ':' ⟨*PortType*⟩ ';'

⟨*DisconnectStatement*⟩ ::= '!disconnect' ⟨*Expression*⟩ '=>' ⟨*Expression*⟩ ':' ⟨*PortType*⟩ ';'

⟨*TriggerStatement*⟩ ::= '!trigger' ⟨*Expression*⟩ '=>' ⟨*Expression*⟩ ';'

APPENDIX

B

Primer on the Rust Language

There is no real ending. It's just the place where you stop the story.

– Frank Herbert

As background to Kompact's DSL, presented in section 8.1, and internal implementation, as described in section 8.2, we quickly summarise the most relevant features of the Rust language in this appendix. More details on everything described herein can be found in Rust's official documentation [102].

In general, Rust is a strongly, statically typed, imperative language with first-class function objects, allowing many functional patterns to be expressed in it. It is compiled to native code by the `rustc` compiler, which uses LLVM [65] as a backend. What is particularly interesting about Rust, is that there exists a subset of the language for which the compiler can statically determine memory-safety, allowing it to generate all deallocation instructions automatically. This subset is called *safe Rust*, and is the only part of Rust we are going to consider in this chapter, as Kompact is written almost exclusively in safe Rust.

B.1 Data

Apart from primitives, such as `bool`, `u8`, `i32`, `f64`, the Rust language provides two important ways to describe data: *Structs* and *enums*.

Structs are simple records with zero or more fields, which are either named or numbered, each of a particular type. A struct with zero fields is essentially a singeleton, and is simply declared as `struct SingletonMsg;`. For a simple counting struct, i.e. a named wrapper around a number, we may write `struct NamedCount{ count: u64 }` or `struct NumberedCount(u64);`. We could then create instances of these structs by simply assigning a value of the appropriate type to each field, for example:

```
let named = NamedCount{ count: 5u64 };
let numbered = NumberedCount(5u64);
```

If we need to access fields, we can use the common dot-notation, referring to either the fields name or its index, for example:

```
println!("{}", named.count); // prints 5
println!("{}", numbered.0); // also prints 5
```

Enums in Rust are tagged unions over structs. If, for example, we wanted to abstract away over whether our counting data is named or numbered, we could declare an enum like:

```
enum Count {
  Named {count: u64},
  Numbered(u64),
}
```

Using this enum, instead of the two different structs before, we can rewrite our previous example as:

```
let count1 = Count::Named{ count: 5u64 };
let count2 = Count::Numbered(5u64);
```

Both variables `count1` and `count2` have the same type, `Count`, now. This allows us, for example, to declare a method that takes this type, instead of two different methods for the two different variants. If we want to read the content now, we must differentiate which instance of the enum we got. This is done via *pattern matching*, such as in the following example:

```
let counted: Count = /* some instance */;
match counted {
  Count::Named{count} => println!("{}", count), // prints 5
  Count::Numbered(count) => println!("{}", count), // also prints 5
}
```

B.2 Behaviour

If we want to do anything with our data, we need to be able to define behaviours on it, which is typically done in functions. In Rust, functions are declared as:

```
fn fname(arg1: Arg1Type, arg2: Arg2Type, ...) -> ReturnType {
/*  body */
}
```

Functions can be declared in three different contexts: At the *module-level*, associated with a *type*, or associated with a *trait*.

Module-level functions can be thought of as global functions within a particular namespace. We are going to ignore the concrete namespace in the rest of this chapter, and simply assume that there is one we are currently in. If, for example, we wanted a quick way to produce a fresh `Count` instances, we could declare a module-level function:

```
fn zero_count() -> Count {
  Count::Numbered(0u64)
}
```

Type associated functions are declared within an `impl`-block. This basically adds them under the type's namespace, instead of the module's. As our `zero_count` function is basically a special constructor for `Count` instances, it may be more sensible to associate it with that particular type instead, and write:

```
impl Count {
  fn zero() -> Count {
    Count::Numbered(0u64)
  }
}
```

If we wanted to invoke it now, we would write `Count::zero()`. Functions declared in this manner, however, do not have access to any of the fields of a `Count`, as they are only associated with the *type* `Count`. If we wanted to produce an incremented new count, we would have to pass a `self`-argument to the function, and write:

```
impl NumberedCount {
  fn inc(self) -> NumberedCount {
    NumberedCount(self.0 + 1u64)
  }
}
```

Not only does this code construct a new, incremented count instance, but it also automatically deallocates the old instance, before it returns. This happens because we passed an *owned* `self`-argument to the function, which the compiler can prove is not used any more after the function body. This may, of course, not be the most efficient way implement our counter increment, depending on how we use it. Perhaps we want to continue using the old instance, in addition to the new one. Or perhaps we wish to simply increment the counter in-place and avoid additional memory allocations for every increment operation.
Rust allows us two additional ways to pass arguments, apart from passing them *owned*: We can pass them as a *shared reference*, for example `&self`, or as a *mutable reference*, such as `&mut self`. The Rust compiler will guarantee us that only a single mutable reference to a particular objects exists at any given time in the code, and while it exists no shared references exist either. The opposite is true for shared references, where an arbitrary number can exist at any given time, but they do not allow modifications to be made to the data. Using these references, our two increment semantics could be implemented by the following two functions:

```
impl NumberedCount {
  fn inc_no_deallocate(&self) -> NumberedCount {
    NumberedCount(self.0 + 1u64)
  }
  fn inc_in_place(&mut self) -> () {
    self.0 += 1u64;
  }
}
```

Of course, the notation for references works for any type, not just for the special `self`-argument. We could, for example, also write module-level function with the same semantics as `inc_in_place` above:

```
fn inc_count_in_place(count: &mut NumberedCount) -> () {
  count.0 += 1u64;
}
```

Trait associated functions allow us to separate the signature of a function from its implementation. Rust's traits are thus somewhat similar to interfaces in languages

like Java. If we wanted to stipulate that anything countable must allow for in-place increments, we could define a trait for this, like:

```
trait Incrementable {
  fn inc_in_place(&mut self) -> ();
}
```

Now we could associate an implementation of this trait with a particular counter using an `impl`-block, for example:

```
impl Incrementable for NamedCount {
  fn inc_in_place(&mut self) -> () {
    self.count += 1u64;
  }
}
```

Whichever of these implementations we have chosen, we can always invoke them on a variable `count` via `count.inc_in_place()`. Note that we are not writing the `self`-argument into the argument list here. This is a convenient semantic sugar the Rust compiler grants us, as long as the invocation is not ambiguous. If, for example, two different traits were to provide an `inc_in_place` function, we would have to specifically write the desugared variant `Incrementable::inc_in_place(&mut count)`.

B.3 Type Parameters, Trait Objects, and Associated Types

In order to write more generic code, Rust allow us to parametrise many kinds of declarations with type arguments. For example, we may want to have a module-level `increment` function, which only requires that whatever argument we pass it provides the `inc_in_place` function from our `Incrementable` trait. Thus the type of its single argument would be a mutable reference to some arbitrary type `I`, with the additional requirement that any such `I` must implement the `Incrementable` trait. The following code would achieve this:

```
fn increment<I: Incrementable>(counter: &mut I) -> () {
  counter.inc_in_place();
}
```

Note that the Rust compiler will monomorphise such polymorphic functions during compilation. That is, it will produce specialised code for every concrete `I` we used to invoke the `increment` with anywhere in our code. If we wish to avoid this code duplication, we can instead pass a *trait object* to the function, which will cause the compiler to generate a dynamically dispatched call instead. This is achieved by writing the following code:

```
fn increment(counter: &mut Incrementable) -> () {
  counter.inc_in_place();
}
```

There are a number of limitations in Rust for when such trait objects can be used. While the details are out of the scope of our discussion here, simply said, it boils down to whether or not the compiler can (and needs to) statically determine the memory size of whatever argument is being passed.

Associated Types Similar to Scala, Rust's traits can also have associated types, in addition to type parameters. An associated type behaves very similar to type parameters, except that it only allows a single trait implementation for a particular type. Consider, for example the `Add` trait from Rust's standard library:

```
pub trait Add<Rhs> {
    type Output;
    fn add(self, rhs: Rhs) -> Self::Output;
}
```

This trait uses both a type parameter `Rhs` and an associated type `Output`. This allows us to implement addition for our type with multiple different right-hand-sides, but never producing different outputs for a given `Rhs` type. For example, consider that we have defined a type `struct Angle(f64)` and we want to allow it to add both other angles and scalars, which we will interpret to represent angles as well. Thus we may write:

```
impl Add<Angle> for Angle {
  type Output = Angle;
  fn add(self, rhs: Angle) -> Self::Output {
    Angle(self.0 + rhs.0)
  }
}
impl Add<f64> for Angle {
  type Output = Angle;
  fn add(self, rhs: f64) -> Self::Output {
    Angle(self.0 + rhs)
  }
}
```

This allows us to write code like `Angle(5.0) + Angle(1.0)` and `Angle(5) + 1.0` with the same result.

What we most certainly can *not* write is:

```
impl Add<Angle> for Angle {
  type Output = Angle;
  fn add(self, rhs: Rhs) -> Self::Output {
    Angle(self.0 + rhs.0)
  }
}
impl Add<Angle> for Angle {
  type Output = f64;
  fn add(self, rhs: Angle) -> Self::Output {
    self.0 + rhs
  }
}
```

In this case we would have two implementations for the same trait for the same object, which would be terribly ambiguous.

Note that, if `Output` had been a second type parameter, instead of an associated type, it would have been perfectly valid to write:

```
impl Add<Angle, Angle> for Angle {
  fn add(self, rhs: Angle) -> Angle {
    Angle(self.0 + rhs.0)
  }
}
impl Add<Angle, f64> for Angle {
  fn add(self, rhs: Angle) -> f64 {
    self.0 + rhs
  }
}
```

The decision on whether to use an associated type or a type parameter, thus, boils down to the semantics one wishes to achieve or enforce with the trait.

B.4 Macros

In addition to writing code that will be evaluated at execution time, Rust also provides two ways to specify code which should be evaluated to other code at compile time: *Procedural macros* and *derive macros*.

Procedural macros are similar to a function invocation, but they are evaluated at compile time and replaced with the result of that evaluation in the code. For example, we have already seen the `println!("{}",count)` macro invocation, which is replaced by appropriate code to splice the `count` field's current value into the given string and then print the result to standard output.

Derive macros, on the other hand, are not invoked, but rather annotated. They allow direct manipulation of the token stream or syntax tree within the Rust compiler. Typically, derive macros are used to automatically implement some behaviour,

often a trait, for a particular struct or enum. This frees programmers from writing often tedious code manually and keeping it updated whenever the underlying data changes. The Rust standard library provides a number of derive macros for common traits, such as `Clone`, which provides a field by field copy of a struct, and `Debug`, which generates pretty-printing code. If we wish our `Angle` struct to be copyable and pretty-printable, we could annotate it in the following manner, for example:

```
#[derive(Clone, Debug)]
struct Angle(f64);
```

This would then later allow us to write things like:

```
let a = Angle(5.0);
// clone() uses the generated cloning code
let b = a.clone() + 1.0;
// {:?} uses the generated Debug pretty-printer
println!("a={:?} vs. b={:?}", a, b);
```

Note that derive macros are not technically limited to generating only a single a trait implementation of the same name as the macro. They could generate any number of trait implementations, or some kind of companion struct or function, for example. However, as their predominant usage in libraries is to generate a single trait implementation of the same name, this is certainly the general expectation a programmer seeing a derive macro is going to have. Thus, deviating from this standard may lead to some confusion and may not be the best library design, generally speaking.

Bibliography

[1] "Congestion Control in IP/TCP Internetworks," RFC 896, Jan. 1984. [Online]. Available: https://rfc-editor.org/rfc/rfc896.txt

[2] "ISO/IEC/IEEE 24765: 2010 (E) IEEE, Systems and Software Engineering - Vocabulary," Tech. Rep., 2010.

[3] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," *ACM SIGCOMM computer communication review*, vol. 41, no. 4, pp. 63–74, 2011.

[4] N. Amin, S. Grütter, M. Odersky, T. Rompf, and S. Stucki, "The essence of dependent object types," in *A List of Successes That Can Change the World*. Springer, 2016, pp. 249–272.

[5] C. Arad, J. Dowling, and S. Haridi, "Building and evaluating P2P systems using the kompics component framework," *IEEE P2P'09 - 9th International Conference on Peer-to-Peer Computing*, pp. 93–94, 2009.

[6] ——, "Developing, simulating, and deploying peer-to-peer systems using the kompics component model," in *Proceedings of the Fourth International ICST Conference on Communication system software and middleware*. ACM, 2009, p. 16.

[7] ——, "Message-passing Concurrency for Scalable, Stateful, Reconfigurable Middleware," in *Proceedings of the 13th International Middleware Conference*, ser. Middleware '12. New York, NY, USA: Springer-Verlag New York, Inc., 2012, pp. 208–228. [Online]. Available: http://dl.acm.org/citation.cfm?id=2442626.2442640

[8] C. I. Arad, "Programming Model and Protocols for Reconfigurable Distributed Systems," Ph.D. dissertation, KTH - Royal Institute of Technology, Stockholm, 2013. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:ri:diva-24202

[9] J. Armstrong, "Making reliable distributed systems in the presence of software errors," no. December, p. 295, 2003. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-3658

[10] ——, "A history of Erlang," in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. ACM, 2007, pp. 1–6.

[11] Aspera FASP, "http://asperasoft.com/technology/transport/fasp/."

[12] A. Baldini, L. De Carli, and F. Risso, "Increasing performances of TCP data transfers through multiple parallel connections," in *Computers and Communications, 2009. ISCC 2009. IEEE Symposium on*. IEEE, 2009, pp. 630–636.

[13] M. Belshe, R. Peon, and M. Thomson, "Hypertext Transfer Protocol Version 2 (HTTP/2)," RFC 7540, May 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc7540.txt

[14] P. Bokor, J. Kinder, M. Serafini, and N. Suri, "Supporting Domain-specific State Space Reductions Through Local Partial-order Reduction," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 113–122. [Online]. Available: http://dx.doi.org/10.1109/ASE.2011.6100044

[15] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin, "Orleans: cloud computing for everyone," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 2011, p. 16.

[16] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[17] C. Caini and R. Firrincieli, "TCP Hybla: a TCP enhancement for heterogeneous networks," *International journal of satellite communications and networking*, vol. 22, no. 5, pp. 547–566, 2004.

[18] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas, "Apache Flink: Unified Stream and Batch Processing in a Single Engine," *Data Engineering*, 2015.

[19] R. C. Cardoso, M. R. Zatelli, J. F. Hübner, and R. H. Bordini, "Towards Benchmarking Actor- and Agent-based Programming Languages," in *Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control*, ser. AGERE! 2013. New York, NY, USA: ACM, 2013, pp. 115–126. [Online]. Available: http://doi.acm.org/10.1145/2541329.2541339

[20] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Achieving scalability and expressiveness in an internet-scale event notification service," in *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*. ACM, 2000, pp. 219–227.

[21] A. Chien, T. Faber, A. Falk, J. Bannister, R. Grossman, and J. Leigh, "Transport protocols for high performance: Whither TCP," *Communications of the ACM*, vol. 46, no. 11, pp. 42–49, 2003.

[22] N. Chomsky, "Three models for the description of language," *IRE Transactions on information theory*, vol. 2, no. 3, pp. 113–124, 1956.

[23] E. M. Clarke and B.-H. Schlingloff, "Handbook of Automated Reasoning," A. Robinson and A. Voronkov, Eds. Amsterdam, The Netherlands, The Netherlands: Elsevier Science Publishers B. V., 2001, ch. Model Chec, pp. 1635–1790. [Online]. Available: http://dl.acm.org/citation.cfm?id=778522.778533

[24] B. Cohen, "Incentives Build Robustness in BitTorrent," 2003.

[25] J. Conway, "The game of life," *Scientific American*, vol. 223, no. 4, p. 4, 1970.

[26] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, jan 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[27] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A survey on model-based testing approaches," *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007 - WEASELTech '07*, pp. 31–36, 2007. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1353673.1353681

[28] M. Dong, Q. Li, D. Zarchy, B. Godfrey, and M. Schapira, "PCC: Re-architecting Congestion Control for Consistent High Performance," sep 2014. [Online]. Available: http://arxiv.org/abs/1409.7092

[29] G. Dubochet, "Computer Code as a Medium for Human Communication: Are Programming Languages Improving?" *Proceedings of the 21st Working Conference on the Psychology of Programmers Interest Group*, pp. 174–187, 2009. [Online]. Available: http://infoscience.epfl.ch/record/138586

[30] T. Elmas, J. Burnim, G. Necula, and K. Sen, "CONCURRIT: A domain specific language for reproducing concurrency bugs," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 153–164, 2013.

[31] et al. Neale, R., "Coupled simulations from CESM1 using the Community Atmosphere Model version 5: (CAM5)," *See also http://www.cesm.ucar.edu/publications*, 2012. [Online]. Available: http://www.cesm.ucar.edu/publications

[32] W.-c. Feng and P. Tinnakornsrisuphap, "The failure of TCP in high-performance computational grids," in *Supercomputing, ACM/IEEE 2000 Conference*. IEEE, 2000, p. 37.

[33] R. W. Floyd, "Algorithm 97: Shortest Path," *Commun. ACM*, vol. 5, no. 6, pp. 345—-, jun 1962. [Online]. Available: http://doi.acm.org/10.1145/367766.368168

[34] A. Ford, C. Raiciu, M. Handley, S. Barre, J. Iyengar, and Others, "Architectural guidelines for multipath TCP development," *IETF, Informational RFC*, vol. 6182, pp. 1721–2070, 2011.

[35] S. Fowler, S. Lindley, and P. Wadler, "Mixing Metaphors: Actors as Channels and Channels as Actors (Extended Version)," vol. 1, no. 11, pp. 1–11, 2017. [Online]. Available: http://arxiv.org/abs/1611.06276

[36] L.-Å. Fredlund and H. Svensson, "McErlang: a model checker for a distributed functional programming language," in *ACM SIGPLAN Notices*, vol. 42, no. 9. ACM, 2007, pp. 125–136.

[37] E. Gagnon and L. Hendren, "SableCC – an object-oriented compiler framework," *Proceedings of TOOLS 1998*, 1998. [Online]. Available: citeseer.ist.psu.edu/gagnon98sablecc.html

[38] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Abstraction and Reuse of Object-Oriented Design*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 406–431. [Online]. Available: http://dx.doi.org/10.1007/3-540-47910-4_21

[39] L. Gilman and R. Schreiber, *Distributed computing with IBM MQSeries*. John Wiley & Sons, Inc., 1996.

[40] Google Inc., "The Go Programming Language Specification," 2018. [Online]. Available: https://golang.org/ref/spec

[41] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, and D. Smith, "The Java Language Specification, Java SE 12 Edition," Tech. Rep., 2019. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se12/jls12.pdf

[42] Y. Gu and R. L. Grossman, "UDT: UDP-based data transfer for high-speed wide area networks," *Computer Networks*, vol. 51, no. 7, pp. 1777–1799, 2007.

[43] S. Ha, I. Rhee, and L. Xu, "CUBIC: a new TCP-friendly high-speed TCP variant," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 64–74, 2008.

[44] P. Haller and M. Odersky, "Scala Actors: Unifying thread-based and event-based programming," *Theoretical Computer Science*, vol. 410, no. 2-3, pp. 202–220, 2009. [Online]. Available: http://dx.doi.org/10.1016/j.tcs.2008.09.019

[45] S. Hanenberg, S. Kleinschmager, R. Robbes, É. Tanter, and A. Stefik, "An empirical study on the impact of static typing on software maintainability," *Empirical Software Engineering*, vol. 19, 2013.

[46] J. He, P. Wadler, and P. Trinder, "Typecasting Actors: From Akka to TAkka," *Proceedings of the Fifth Annual Scala Workshop*, pp. 23–33, 2014. [Online]. Available: http://doi.acm.org/10.1145/2637647.2637651

[47] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, A. Skou, M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson, *Formal Methods and Testing*, 2008, vol. 4949. [Online]. Available: https://dl.acm.org/doi/10.5555/1806209.1806212

[48] C. Hewitt, P. Bishop, and R. Steiger, "Session 8 Formalisms for Artificial Intelligence A Universal Modular ACTOR Formalism for Artificial Intelligence," in *Advance Papers of the Conference*, vol. 3. Stanford Research Institute, 1973, p. 235.

[49] P. Hintjens, *ZeroMQ: Messaging for Many Applications*. " O'Reilly Media, Inc.", 2013.

[50] C. A. R. Hoare, "Communicating sequential processes," in *The origin of concurrent programming*. Springer, 1978, pp. 413–443.

[51] M. R. Hoffmann, B. Janiczak, and E. Mandrikov, "Eclemma-jacoco java code coverage library," 2011.

[52] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, Inc, 1979.

[53] A. Igarashi, B. C. Pierce, and P. Wadler, "Featherweight Java: a minimal core calculus for Java and GJ," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 23, no. 3, pp. 396–450, 2001.

[54] S. Imam and V. Sarkar, "Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries," *Proceedings of the 4th International Workshop on Programming Based on Actors Agents and Decentralized Control (AGERE)*, pp. 67–80, 2014. [Online]. Available: http://doi.acm.org/10.1145/2687357.2687368

[55] P. Z. Ingerman, "Algorithm 141: Path Matrix," *Commun. ACM*, vol. 5, no. 11, pp. 556—-, nov 1962. [Online]. Available: http://doi.acm.org/10.1145/368996.369016

[56] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," Internet Engineering Task Force, Internet-Draft draft-ietf-quic-transport-24, Nov. 2019, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-24

[57] D. Jurafsky and J. H. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, ser. Practical Resources for the Mental Health Professionals Series. Prentice Hall, 2000.

[58] C. Kaiser and J.-F. J.-F. Pradat-Peyre, "Chameneos, a concurrency game for Java, Ada and others," *Computer Systems and Applications*, p. 8, 2003.

[59] M. Khan, "A Comparative Study of White Box, Black Box and Grey Box Testing Techniques," *International Journal of Advanced Computer Science and Applications*, vol. 3, no. 6, pp. 12–15, 2012.

[60] S. C. Kleene, "Representation of events in nerve nets and finite automata," DTIC Document, Tech. Rep., 1951.

[61] L. Kroll, P. Carbone, and S. Haridi, "Kompics Scala: Narrowing the Gap Between Algorithmic Specification and Executable Code (Short Paper)," in *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*, ser. SCALA 2017. New York, NY, USA: ACM, 2017, pp. 73–77. [Online]. Available: http://doi.acm.org/10.1145/3136000.3136009

[62] L. Kroll, J. Dowling, and S. Haridi, "Static Type Checking for the Kompics Component Model: Kola – The Kompics Language," in *First Workshop on Programming Models and Languages for Distributed Computing*, ser. PMLDC '16. New York, NY, USA: ACM, 2016, pp. 2:1—-2:10. [Online]. Available: http://doi.acm.org/10.1145/2957319.2957371

[63] L. Kroll, A. A. Ormenisan, and J. Dowling, "Fast and Flexible Networking for Message-Oriented Middleware," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, jun 2017, pp. 1453–1464. [Online]. Available: http://ieeexplore.ieee.org/document/7980084/

[64] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, may 1998. [Online]. Available: http://doi.acm.org/10.1145/279227.279229

[65] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," San Jose, CA, USA, Mar 2004, pp. 75–88.

[66] S. Lauterburg, M. Dotta, D. Marinov, and G. Agha, "A framework for state-space exploration of Java-based actor programs," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2009, pp. 468–479.

[67] S. Lauterburg, R. K. Karmani, D. Marinov, and G. Agha, "Evaluating Ordering Heuristics for Dynamic Partial-order Reduction Techniques," in *Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering*, ser. FASE'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 308–322. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12029-9_22

[68] M. Lienhardt, A. Schmitt, and J.-B. Stefani, "Oz/K: A kernel language for component-based open programming," in *Proceedings of the 6th international conference on Generative programming and component engineering*. ACM, 2007, pp. 43–52.

[69] Lightbend Inc., "Akka Documentation: Release 2.5.23," 2019. [Online]. Available: https://doc.akka.io/docs/akka/2.5.23/

[70] ——, "Akka TestKit: Release 2.5.23," 2019. [Online]. Available: https://doc.akka.io/docs/akka/2.5.23/testing.html

[71] ——, "Akka Typed Documentation: Release 2.5.23," 2019. [Online]. Available: https://doc.akka.io/docs/akka/2.5.23/typed/index.html

[72] S. Liu, T. Ba\csar, and R. Srikant, "TCP-Illinois: A loss-and delay-based congestion control algorithm for high-speed networks," *Performance Evaluation*, vol. 65, no. 6, pp. 417–440, 2008.

[73] N. Lynch, "Input/Output automata: Basic, timed, hybrid, probabilistic, dynamic,..." in *International Conference on Concurrency Theory*. Springer, 2003, pp. 191–192.

[74] N. A. Lynch, *Distributed algorithms*. Elsevier, 1996.

[75] N. A. Lynch and A. A. Shvartsman, "Tempo: A toolkit for the timed input/output automata formalism," VEROMODO INC BROOKLINE MA, Tech. Rep. 0704, 2008.

[76] T. Mackinnon, S. Freeman, and P. Craig, "Endo-Testing: Unit Testing with Mock Objects," *eXtreme Programming and Flexible Processes inSoftware Engineering - XP2000*, 2000.

[77] P. Maiya, R. Gupta, A. Kanade, and R. Majumdar, "Partial Order Reduction for Event-Driven Multi-threaded Programs," in *Proceedings of the 22Nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9636*. New York, NY, USA: Springer-Verlag New York, Inc., 2016, pp. 680–697. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-49674-9_44

[78] S. Mascolo, C. Casetti, M. Gerla, M. Y. Sanadidi, and R. Wang, "TCP westwood: Bandwidth estimation for enhanced transport over wireless links," in *Proceedings of the 7th annual international conference on Mobile computing and networking*. ACM, 2001, pp. 287–297.

[79] N. D. Matsakis and F. S. Klock II, "The rust language," in *ACM SIGAda Ada Letters*, vol. 34, no. 3. ACM, 2014, pp. 103–104.

[80] C. S. Meiklejohn, H. Miller, and P. Alvaro, "PARTISAN: Scaling the Distributed Actor Runtime," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 63–76. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/meiklejohn

[81] J. Mickos, "Design of a Network Library for Continuous Deep Analytics," master's thesis, KTH, School of Electrical Engineering and Computer Science (EECS), 2018. [Online]. Available: http://kth.diva-portal.org/smash/get/diva2:1232485/FULLTEXT01.pdf

[82] P. Nordwall, "50 Million Messages Per Second - on a Single Machine," 2012. [Online]. Available: http://letitcrash.com/post/20397701710/50-million-messages-per-second-on-a-single-machine

[83] H. Obata, K. Tamehiro, and K. Ishida, "Experimental evaluation of TCP-STAR for satellite Internet over WINDS," in *Autonomous Decentralized Systems (ISADS), 2011 10th International Symposium on*. IEEE, 2011, pp. 605–610.

[84] M. Odersky, P. Altherr, V. Cremet, G. Dubochet, B. Emir, P. Haller, S. Micheloud, N. Mihaylov, A. Moors, L. Rytz, M. Schinz, E. Stenman, and M. Zenger, "Scala Language Specification: Release 2.13," 2019. [Online]. Available: https://scala-lang.org/files/archive/spec/2.13/

[85] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "An Overview of the Scala Programming Language," *Technical Report 64*, 2004.

[86] B. C. Pierce and C. Benjamin, *Types and programming languages*. MIT press, 2002.

[87] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving datacenter performance and robustness with multipath tcp," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 266–277, 2011.

[88] Riker, "Riker Documentation: Release 0.3," 2019. [Online]. Available: https://riker.rs/riker03/

[89] B. Roscoe, "A theory of communicating sequential processes," *Journal of the ACM*, vol. 255 LNCS, no. 3, pp. 442–465, 1984.

[90] A. Scalas, N. Yoshida, and E. Benussi, "Effpi: Verified Message-passing Programs in Dotty," in *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala*, ser. Scala '19. New York, NY, USA: ACM, 2019, pp. 27–31. [Online]. Available: http://doi.acm.org/10.1145/3337932.3338812

[91] M. Shafique and Y. Labiche, "A systematic review of state-based test tools," *International Journal on Software Tools for Technology Transfer*, vol. 17, no. 1, pp. 59–76, 2013.

[92] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind, "Low extra delay background transport (LEDBAT)," *IETF draft*, 2010.

[93] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 1998.

[94] S. Tasharofi, M. Gligoric, D. Marinov, and R. Johnson, "Setac : A Framework for Phased Deterministic Testing of Scala Actor Programs," *Proc. of the Scala Workshop*, 2011.

[95] S. Tasharofi, M. Pradel, Y. Lin, and R. Johnson, "Bita: Coverage-guided, automatic testing of actor programs," *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings*, pp. 114–124, 2013.

[96] The Actix Team, "Actix Documentation: Release 1.0," 2019. [Online]. Available: https://actix.rs/docs/

[97] The Elixir Developers, "Elixir Documentation: Release 1.9.1," 2019. [Online]. Available: https://hexdocs.pm/elixir/1.9.1

[98] The JUnit Developers, "JUnit 5 Documenation," 2018. [Online]. Available: https://junit.org/junit5/docs/current/user-guide/

[99] The Kompics Developers, "Kompics Documentation: Release 1.1.0," 2019. [Online]. Available: https://kompics.github.io/docs/1.1.0/

[100] The Netty Project, "Netty," 2019. [Online]. Available: http://netty.io/

[101] The Orleans Developers, "Orleans Documentation: Release 2.0," 2018. [Online]. Available: https://dotnet.github.io/orleans/Documentation/index.html

[102] The Rust Developers, "Rust Language Reference," 2019. [Online]. Available: https://doc.rust-lang.org/stable/reference/

[103] I. W. Ubah, "A Language-Recognition Approach to Unit Testing Message-Passing Systems," master's thesis, KTH, School of Information and Communication Technology (ICT), 2017. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-215655

[104] I. W. Ubah, L. Kroll, A. A. Ormenisan, and S. Haridi, "KompicsTesting - Unit Testing Event Streams," *CoRR*, vol. abs/1705.0, 2017. [Online]. Available: http://arxiv.org/abs/1705.04669

[105] P. Van Roy, S. Haridi, P. Brand, G. Smolka, M. Mehl, and R. Scheidhauer, "Mobile Objects in Distributed Oz," *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 5, pp. 804–851, 1997. [Online]. Available: http://portal.acm.org/citation.cfm?doid=265943.265972

[106] G. Varghese and T. Lauck, "Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility," in *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, ser. SOSP '87. New York, NY, USA: ACM, 1987, pp. 25–38. [Online]. Available: http://doi.acm.org/10.1145/41457.37504

[107] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson, *Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer*, ser. Lecture Notes in Computer Science. Springer Verlag, jan 2008, vol. 4949, pp. 39–76. [Online]. Available: https://www.microsoft.com/en-us/research/publication/model-based-testing-of-object-oriented-reactive-systems-with-spec-explorer/

[108] B. Venners and Artima, "ScalaTest," 2009. [Online]. Available: http://www.scalatest.org/user_guide

[109] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Software Engineering*, vol. 10, no. 2, pp. 203–232, 2003.

[110] M. Wachs, F. Oehlmann, and C. Grothoff, "Automatic transport selection and resource allocation for resilient communication in decentralised networks," in *14-th IEEE International Conference on Peer-to-Peer Computing*. IEEE, sep 2014, pp. 1–5. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6934301

[111] D. X. Wei, C. Jin, S. H. Low, and S. Hegde, "FAST TCP: motivation, architecture, algorithms, performance," *IEEE/ACM Transactions on Networking (ToN)*, vol. 14, no. 6, pp. 1246–1259, 2006.

[112] C. Wikström, "Distributed programming in Erlang," in *the 1st International Symposium on Parallel Symbolic Computation (PASCO 94)*. World Scientific, 1994, pp. 412–421.

[113] K. Winstein and H. Balakrishnan, "Tcp ex machina: Computer-generated congestion control," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 123–134.

[114] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley, "Design, Implementation and Evaluation of Congestion Control for Multipath TCP." in *NSDI*, vol. 11, 2011, p. 8.

[115] H. Wu, Z. Feng, C. Guo, and Y. Zhang, "ICTCP: incast congestion control for TCP in data-center networks," *IEEE/ACM Transactions on Networking (TON)*, vol. 21, no. 2, pp. 345–358, 2013.

[116] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark : Cluster Computing with Working Sets," *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, p. 10, 2010.

[117] A. Zakeriyan, E. Khamespanah, M. Sirjani, and R. Khosravi, "Jacco: More Efficient Model Checking Toolset for Java Actor Programs," in *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, ser. AGERE! 2015. New York, NY, USA: ACM, 2015, pp. 37–44. [Online]. Available: http://doi.acm.org/10.1145/2824815.2824819

Acronyms

AMI Amazon Machine Image. 168

API application programming interface. 16, 17, 103, 155, 156, 160, 189, 197, 207, 212

AST abstract syntax tree. 72, 74, 79

BDP bandwidth delay product. 158, 171, 211

BNF Backus–Naur form. 215

CD continuous delivery. 4

CFG context-free grammar. xviii, 92, 94, 97, 100–102

CI continuous integration. 4

CPU central processing unit. 3, 53, 139, 145, 170, 195, 200

CSG context-sensitive grammar. 98

CSP Communicating Sequential Processes. 11

CST concrete syntax tree. 79

CSV comma-separated values. 117

CUT component under test. 89, 91, 92, 99, 101, 103, 104, 106

DFA deterministic finite automaton. 89, 92, 93, 95, 100, 167

DSL domain-specific language. xi, xviii, 3, 5, 6, 8, 26, 27, 32, 40, 41, 51, 52, 55, 59, 68, 69, 80, 81, 86, 87, 89, 94, 100, 102, 110, 111, 174, 178, 179, 181, 183, 185, 187, 189, 191, 209, 215, 219

eDSL embedded domain-specific language. 6, 8, 40, 69, 102, 119, 178

FIFO first-in, first-out. 13, 14, 20, 161, 162

FSM finite state machine. 63, 64, 83, 110

HPC high-performance computing. 169

HTTP Hypertext Transfer Protocol. 171

HVM hardware virtual machine. 168

I/O input/output. 160, 195, 196, 241

IDE integrated development environment. 210

JIT just-in-time. 82, 117, 130, 147

JST Java syntax tree. 79

JVM Java virtual machine. 41, 42, 71, 82, 114, 117, 120, 122, 136, 139, 140, 147, 150, 153, 155, 195, 200

LEDBAT Low Extra Delay Background Transport. 172

LIFO last-in, first-out. 97

LOC lines of code. 52, 80, 82, 124

MBT model-based testing. 109, 110

MPP message-passing performance. 51, 53, 114–116, 120–124, 136, 138, 178, 211

NAT network address translation. 159, 164

NFA nondeterministic finite automaton. 95

NIO non-blocking I/O. 160

OS operating system. 131, 195, 196

PCC Performance-oriented Congestion Control. 172

RPC remote procedure call. 12, 84

RSE relative standard error. 117, 120, 123, 145, 169

RTT round-trip time. 53, 166, 168–170

S2S source-to-source. 69, 78, 210

SCP secure copy protocol. 171

SMP symmetric multiprocessing. 139, 140, 147, 153, 155, 199

TCP Transmission Control Protocol. 131, 157, 158, 161–164, 166–169, 171, 172

UDP User Datagram Protocol. 158, 162–164, 166, 169

UDT UDP-based Data Transfer Protocol. 162–164, 166–169, 172

UI user interface. 106

URI Uniform Resource Identifier. 190

VM virtual machine. 14, 138, 153, 168

VPC Virtual Private Cloud. 168, 169