KTH Information and Communication Technology

Generic Distribution Support for Programming Systems

Erik Klintskog

A Dissertation submitted to
the Royal Institute of Technology
in partial fulfillment of the requirements for
the degree of Doctor of Technology

June 2005

The Royal Institute of Technology
School of Information and Communication Technology
Department of Microelectronics and Information Technology

TRITA-IT LECS AVH 05:03
ISSN 1651-4076
ISRN KTH/IMIT/LECS/AVH-05/03–SE

and

SICS Dissertation Series 39
ISSN 1101-1335
ISRN SICS-D–39–SE

To Malin, Hedda, Tilde and Estrid

Abstract

This dissertation provides constructive proof, through the implementation of a middleware, that distribution transparency is practical, generic, and extensible. Fault tolerant distributed services can be developed by using the failure detection abilities of the middleware. By generic we mean that the middleware can be used for many different programming languages and paradigms. Distribution for each kind of language entity is done in terms of consistency protocols, which guarantee that the semantics of the entities are preserved in a distributed setting. The middleware allows new consistency protocols to be added easily. The efficiency of the middleware and the ease of integration are shown by coupling the middleware to a programming system, which encompasses the object oriented, the functional, and the concurrent-declarative programming paradigms. Our measurements show that the distribution middleware is competitive with the most popular distributed programming systems (JavaRMI, .NET, IBM CORBA).

Acknowledgments

I would like to start by showing my gratitude to my supervisor professor Seif Haridi for his guidance and encouragement throughout this whole process. Seif introduced me to distributed programming in general and Mozart in particular and always engaged me in interesting and challenging research issues. Another person that has been of outmost importance for my work is Per Brand. Per has been acting as a supervisor, a mentor, and a close friend to me.

The work presented in this dissertation would not have been possible without the help of Zacharias El Banna. Not only was Zacharias a gifted colleague, he also became a close friend of mine. Another person that deserves gratitude from me is Anna Neiderud. The work we conducted together on the Mozart system was invaluable for my later research.

This dissertation builds heavily on the experiences from the Mozart project collected at the Distributed Systems Laboratory (DSL) at SICS. Therefore, I would like to thank the developers of Mozart. Particularly, I would like to thank Konstantin Popov and Andreas Sundström for the work we did together. In addition, I would like the members of DSL for creating such an productive environment, thanks goes to Dragan Havelka, Sameh El-Ansary, Fredrik Holmlund, Per Sahlin, and Nils Franzen.

Ali Ghodsi and Lars-Åke Fredlund have been most helpful during the writing of this dissertation. The feedback I received from them on structure, style, and language was invaluable for the final result. Moreover, I would like to show my gratitude to Sverker Janson, Frej Drejhammar, and Vicki Carlgren for proof reading later versions of my dissertation. I would like to thank my employer, SICS, for letting me pursue this work. The friendly and warm atmosphere at the SICS Uppsala office, where I spent most of my time, provided by Per Mildner, Markus Bylund, Stina Nylander, certainly simplified my every day research.

Finally, I would like to thank my wife Malin for her profound support during this process. Without her, I would never have finished this dissertation. I thank my parents for raising me to believe in my self and I thank my sister Ingrid for being such a great sister. Moreover, I thank my three daughters Hedda, Tilde, and Estrid for making every day life so wonderful.

List of Papers

This dissertation is composed of the following papers. In the summary they will be referred to as papers A through H.

A Erik Klintskog, Zacharias El Banna, Per Brand and Seif Haridi. The DSS, a Middleware Library for Efficient and Transparent Distribution of Language Entities. In *Proceedings of HICSS'37*, Hawaii, USA, 2004.

B Erik Klintskog, Zacharias El Banna, Per Brand and Seif Haridi. The Design and Evaluation of a Middleware Library for Distribution of Language Entities. In 8^{th} *Asian Computing Conference*, Mumbai, India, 2003.

C Erik Klintskog, Valentin Mesaros, Zacharias El Banna, Per Brand and Seif Haridi. A Peer-to-Peer Approach to Enhance Middleware Connectivity. In *OPODIS 2003:* 7^{th} *International Conference on Principles of Distributed Systems*, Martinique, France, 2003.

D Zacharias El Banna, Erik Klintskog and Per Brand. Securing the DSS Technical Report T2004:14, Swedish Institute of Computer Science, SICS, November 2004.

E Erik Klintskog, Per Brand and Seif Haridi. Home migration using a structured overlay network. To be submitted for review.

F Erik Klintskog, Anna Neiderud, Per Brand and Seif Haridi. Fractional Weighted Reference Counting. In *Proceedings of Euro-Par 2001*, Manchester, England, 2001.

G Erik Klintskog. Internal Design of the DSS. Technical Report T2004:15, Swedish Institute of Computer Science, SICS, 2004.

H Erik Klintskog. Coupling a Programming System to the DSS, a Case Study. Technical Report T2004:16, Swedish Institute of Computer Science, SICS, 2004.

Contents

Chapter 1

Introduction

This dissertation presents the design, implementation, and evaluation of the Distribution SubSystem (DSS), a middleware which provides efficient distribution support for programming languages. It supports the object oriented, the functional, and the declarative-concurrency paradigms. The development time of a distributed programming system can be significantly reduced by the use of the DSS. The distribution support provided by the DSS is customizable and efficient, which in turn results in efficient and functionally comprehensive implementations of distributed programming languages.

The contributions of the dissertation can be summarized as: (i) A programming paradigm independent interface based on an abstract model of language entities. (ii) A framework for consistency protocols which simplifies the development of new protocols and allows for fine grained customization of protocol properties. (iii) The design and implementation of an efficient messaging layer which allows for traversal of firewalls and handling of mobile processes. (iv) The development of protocols and methods which makes it possible to build decentralized and self-organizing distributed applications. As a proof of concept, the middleware has been integrated with to the multiparadigm programming system Mozart, which implements the functional, the declarative-concurrent and the object oriented programming paradigms.

This chapter presents the background and motivation for this disser-

tation. The first section describes distributed systems in general. The second section describes distribution support on the level of programming languages. Thereafter, a section devoted to motivating the work and the thesis follows. The chapter is concluded with a section that presents the contributions of this dissertation.

1.1 Distributed Systems

Computers of today are typically members of some sort of network. Consequently, the resources an application can harness are not necessarily restricted to one computer. Instead, an application can make use of a large set of resources located at many different computers.

A set of interconnected autonomous processes, referred to as *nodes*, constitutes a *distributed system*. Nodes are hosted by computers (sometimes referred to as machines) interconnected by a network that allows the nodes to exchange information. It is possible that more than one node of a distributed system reside on the same computer. If all nodes of a distributed system resides at the same computer (with potentially many processors), some of the characteristic challenges related to distributed systems are reduced, thus in such case, it is more correct to talk about a parallel or a concurrent system. Moreover, a system that consists of one single node is called a *centralized system*.

The rather general description of a distributed system as a set of interconnected nodes is inspired by Gerard Tel [126]. In this dissertation, we devote ourselves to realize a more restricted definition, described by Tanenbaum and van Steen. They state that for a system of nodes to classify as being a distributed system, the existence of autonomous nodes is transparent to users of the system:

> *"A distributed system is a collection of independent computers that appears to its users as a single coherent system."* [122]

A distributed system that adheres to this description appears to its users as a single computer system. A distributed system that appears as a single system is sometimes said to provide a single system image (SSI) [25]. As

noted by Tel [126], and described below, realizing this vision is a daunting task, if at all possible. Still, systems that realize this vision, even just partially, are simpler to use, maintain, and program than systems that do not provide the vision of a single system.

1.1.1 Benefits of Distributed Systems

The deployment of the Internet, cheap communication hardware, and the increased efficiency in computer hardware has made distributed systems so ubiquitous that users seldom recognize that they use a distributed system. Distributed systems are developed with a purpose to provide a service a centralized system cannot provide. Here we present a non-exhaustive list of reasons and motivations for why distributed systems are useful.

The interconnected property of a distributed system allows nodes to *exchange information*. Similarly, users of a distributed system can use the system to exchange information, exemplified by email and instant messaging systems such as ICQ [1]. The interconnected property also allows *resource sharing*, i.e. a node of a distributed system can access and make use of resources present at other nodes. Resources can be anything from physical devices such as sensors or printers to conceptual units such as a compute-server or an information storage facility. The latter type of resource leads to a further argument for distributed systems, the possibility to acquire computation power, and *increase performance* over a centralized application. A distributed system, if carefully designed, can handle node failures while still providing service, thus providing *high availability*. This is in contrast to a centralized system that is extremely vulnerable to failures, i.e. loss of the single node prevents further service.

1.1.2 Challenges

The challenges of developing distributed systems are related to their distribution properties. The nodes of a distributed system are connected by a network and communicate by message passing. Since remote resources are accessed by message passing, accessing a remote resource takes con-

[1] `http://www.icq.com`

siderably more time than accessing a local resource. The overhead in time caused by the messaging is called *latency*. Since the latency varies over time and between nodes of a distributed system, it is hard or even impossible to facilitate an exact global notion of time, i.e. a distributed system lacks a global clock.

A distributed system is subject to node failures. A node can be unavailable because of problems in the underlying network, because the machine that hosts the node has stopped, or because of a deliberate or non-deliberate halt. Unavailability of some of the nodes that make up a distributed system is called *partial failure* [118]. Any information and resources solely located at failed nodes are unavailable to the nodes that remain in the distributed system [23]. The latency in the underlying network makes it hard to differentiate between a failed node and a node to which communication currently takes a long time.

The final challenge related to distributed systems is to make them *scalable*. We adhere to the definition by Clifford Neuman [93] where he defines three dimensions of scalability. (i) A distributed system can be scalable in the sense that the number of nodes can grow without notable degradation of its performance. (ii) A distributed system can be scalable in terms of geographic stretch, that is, nodes can lie far apart. (iii) A distributed system can be administratively scalable, meaning that the system can span multiple administrative domains without becoming administratively impractical.

The challenges described stem from the desire that a set of interconnected nodes should provide the appearance of one single system. Ideally, it should be possible to develop a distributed system without considering details such as where data is located and how data is represented at different machines, and still achieve the same level of performance, scalability, and reliability as if all details of the distributed system were taken into consideration.

1.1.3 Transparency

A distributed system spans multiple machines interconnected by a network. Managing and using such a system is complicated if the underlying structure has to be taken into consideration. In order to simplify the usage of a

distributed system, the physical distribution of machines and resources are typically hidden from the user, the distributed system is said to be *transparent* if it appears to its users as a single computer system. Note that the definition by Tanenbaum and van Steen of a distributed system is according to Tel a transparent distributed system. Take the World-Wide-Web as an example, disregarding whether the contents of a webpage is located at one web-server or at multiple web-servers (unless one of the servers is unavailable), the contents are automatically downloaded and presented in a browser to the user. A non-transparent example would be to require the user to explicitly download the contents of a webpage, item for item, by explicitly connecting using the IP address of the destination server by using the version of the HTTP protocol supported by the web server hosting the content.

Transparency is a multifaceted property, taxonomies over the different dimensions of transparency can be found in the literature [122, 126, 35]. Here we have chosen to present a subset which is of interest for this dissertation.

A system that hides differences in how data is represented and accessed at different nodes is said to provide *access transparency*. If data can be accessed without having to know its physical location, *location transparency* is provided. A system that is both access and location transparent is often said to provide *network transparency*. A distributed system that allows resources to move between nodes without affecting how the resources are accessed implements *migration transparency*. Replication of data to nodes of a distributed system is a technique used to increase scalability and performance. If single replicas can be accessed as if there were just one instance of the data the system is said to implement *replication transparency*. Finally, a distributed system that hides that nodes fail, i.e. partial failures, provides *failure transparency*. Failure transparency is one of the hardest transparency properties to achieve, while access and location transparency is rather straightforward to implement, at least when not considering efficiency.

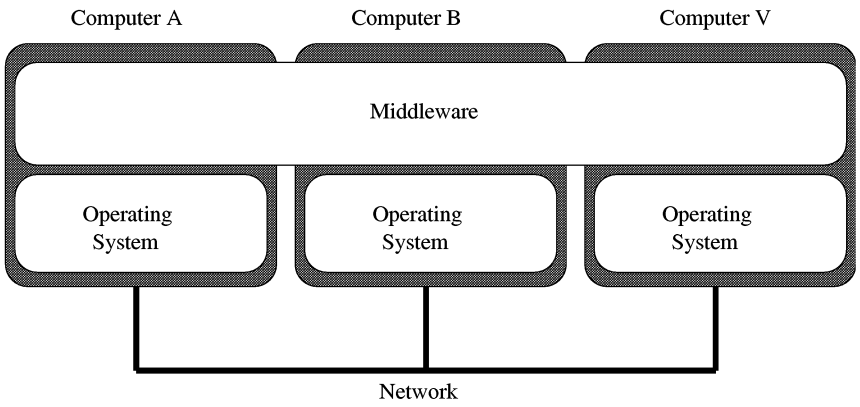


Figure 1.1: A distributed system of three computers, connected by an underlying network. On top of the (network) operating system is a middleware layer. The middleware hides details of the underlying network and provides the appearance of one system to the application programmer.

1.2 Programming Languages for Distributed Systems

Programming distributed applications as such is commonly done using a toolbox of abstractions. Typically a toolbox provides a high-level model of nodes and links that hides details of the underlying services. Almost all operating systems present today allow inter-machine communication and are thus sometimes called *network operating systems*. A network operating system does not per se hide the heterogeneity of a distributed system, instead primitives are provided for communication, i.e. sockets [119], and for remote access for resources, for example telnet and rlogin. The primitives provided are typically only access transparent by the use of standard protocols.

The abstractions provided by a network operating system do not provide the appearance of a single coherent system. *Middleware* [12] is an approach to overcome the limitations in the operating system, providing a platform on top of the operating system for the development of distributed

applications, see Figure 1.1. The purpose of the middleware is to provide a higher level of services than provided by the operating system. Typically, in the form of abstractions which provide location and access transparency. As depicted in the figure, middleware often provide the image of one system. More advanced middleware typically also provide replication and failure transparency.

A more high level type of tool for development of distributed applications is a programming language with integrated distribution support. Distribution services are integrated into the programming model of a programming language, allowing development of distributed systems using programming constructs well known to the programmer. Development of a distributed system is then done similarly to how a centralized system is developed, and consequently simplified, resulting in a shorter development time for distributed applications.

1.2.1 Distributed Programming Languages

The purpose of a distributed programming language is to minimize the complexity of distributed system development. This is achieved by allowing, as far as possible, development of distributed applications as if they were concurrent centralized applications.

We need to distinguish between a programming language and its implementation. The implementation of a programming language is called a *programming system*. A programming system can, for instance, consist of a compiler and a set of libraries as in the case of GCC [48]. A programming system can also be a compiler, a set of libraries and a virtual machine, as in the case of the Mozart [90] Java [52], and the Erlang [42] programming systems. Similar to the separation of a programming language from its implementation in the centralized case, we differentiate between a distributed programming language and its implementations, *distributed programming systems*.

The goal of the Emerald [96] system nicely summarizes the purpose of a distributed programming system: *"The primary goal of Emerald [20, 19] is to simplify distributed programming through language support while providing acceptable performance and flexibility both in local and distributed*

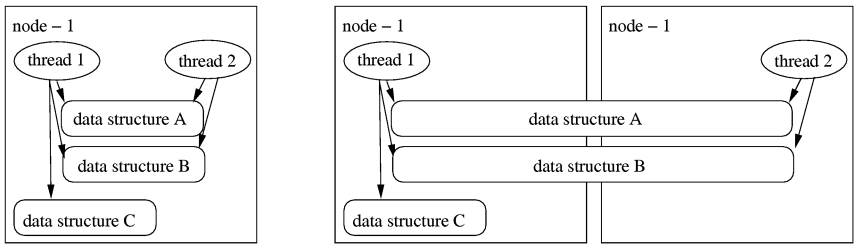


Figure 1.2: One program hosted by a centralized (a) and a distributed system (b). The figure depicts a distributed system that is both location and access transparent. The two threads *thread-1* and *thread-2* can access the data structures A and B in the distributed case similarly as in the centralized case.

environments." [96]. For a programming language based on threads and data structures, distributed language support means that threads of a programming system interact via operations on referred data structures, independently of the physical location of the threads. Figure 1.2(a) depicts an application consisting of two threads that share two data structures, A and B. Figure 1.2(b) depicts the same application distributed over two nodes. The two threads still, conceptually, share the data structures A and B as if the threads were located at the same process. Manipulation of the data structures by one of the threads is visible to the other threads in a similar way as in the centralized setting. Interaction between threads located at different nodes of a distributed system can with this model of shared data structures be treated similarly as interaction between threads in a centralized application.

For the remainder of this dissertation we will use the notion of a *language entity* to denote an instance of a data structure or a data type. The concept of a language entity is independent of programming paradigm and includes not only data structures, but also constructs such as classes and procedures. Moreover, a language entity can be a complex structure such as a data structure that encompasses other data structures. For example,

consider a vector of strings, which can be seen as one language entity that refers a set of other language entities. Or the vector can be seen as one language entity that encompasses the strings. The concept of a language entity does not restrict distribution of information to the granularity enforced by the programming model, but supports tailoring the granularity such that the unit of distribution matches the access patterns of the data.

From an implementation point of view we differentiate between a distributed language entity and a local language entity. A distributed language entity is referred from multiple nodes (by threads or other constructs that can hold references). Independently of how the distributed language entity is realized, there exists only one logical instance. The data structures A and B in Figure 1.2(b) are considered distributed language entities and data structure C is a local language entity. A language entity that is distributed under access, location, and replication transparency is said to provide single-instance equivalence [55], moreover, the language entity is said to be distribution transparent. Disregarding how a language entity is physically distributed, it should provide the same behavior. The semantics of an invocation on a distributed language entity should not differ if the language entity is located in the same process or at another process. Moreover, whether the language entity is replicated or not should not alter its semantics.

Distributed Language Entities

The programming model we assume is based on threads that communicate by accessing and manipulating language entities. A thread that holds a reference to a distributed language entity should be able to interact with the entity as if referring to a local language entity. Consider Figure 1.2(a) and 1.2(b), in both cases, if *thread-1* makes the data structure A point to data structure C, *thread-2* should be able to refer C by accessing A. Of course, this scenario assumes that data structure A can be made to point to other data structures and that data structure A allows access to what it points to. In a centralized setting, this is implemented by representing data structures as single instances in physical memory. All threads of a process (or node according to our notation) have access to the same physical memory and can thus observe modifications performed by other threads. Nodes of a

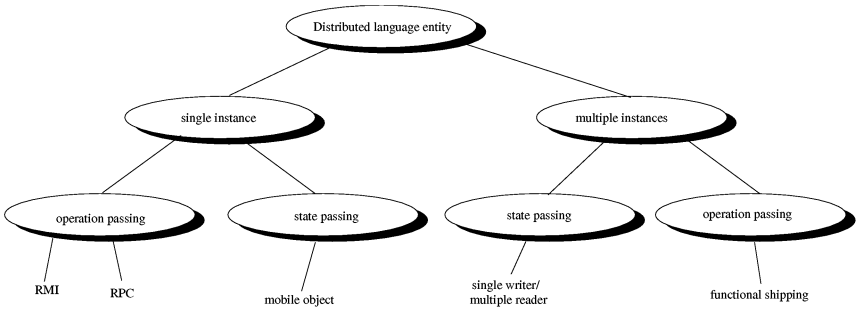


Figure 1.3: A taxonomy over distribution support of language entities with the purpose of achieving single-instance equivalence.

distributed application do not share physical memory, thus, threads located at different nodes cannot directly access the same instance of a distributed language entity. A protocol is required for threads to make use of references to distributed language entities as if the distributed language entity was a centralized language entity. Such a protocol is commonly called a consistency-protocol (see Section 1.2.1 for a discussion about consistency). However, we use the more general notion of a *distribution strategy*. A distribution strategy describes how operations on a distributed language entity are resolved in a distributed system, such that the appearance of one logical instance of the language entity is maintained.

Figure 1.3 depicts a taxonomy of different types of distribution strategies that can be used to maintain single-instance equivalence for a language entity. The different strategies can be further classified into either single-instance or multiple-instances types. The single-instance type of distribution strategies are identified by maintaining one single instance of the language entity at some of the nodes in the distributed system. One possibility is to pass operations to the instance of the language entity, Java RMI [89] and DEC RPC [17] are examples of this type of distribution strategy. Another possibility is to move the instance to where operations on the language entity are performed. This type of distribution strategy is commonly called mobile state and is the distribution strategy used for

objects in Mozart [132] and Aleph [65].

The right sub-tree of Figure 1.3 depicts the family of distribution strategies that maintain multiple instances of a language entity. As opposed to the single-instance type of distribution strategy, multiple instances exist of the same conceptual language entity. Typically threads perform operations on local instances. It is the role of the distribution strategy to ensure that the local instance is in a coherent state. One benefit of multiple-instance distribution strategies is that reading a local instance usually can be done without coordinating with other instances (or replicas) of the language entity. In addition, a multiple instance distribution protocol is potentially more robust to failure than a single-instance distribution strategy.

The challenge of multiple-instance distribution strategies is to keep the instances (or replicas) in a coherent state such that single-instance equivalence is maintained. Multiple-instances distribution strategies can be further characterized in how instances are updated, either by passing a new state description or by passing the operation to each instance. The state passing approach calculates the result of an operation at one location, typically at the node where the operation was performed by a thread. The new state of the language entity is sent to all instances to ensure that they describe the same state. Single-writer/multiple reader protocols [81] are examples of state passing multiple-instances distribution strategies. An operation passing distribution strategy keeps the replicas in a consistent state by performing every operation on each replica. Thus, instead of passing a new state description to each replica, a description of the operation is passed to and performed on each replica. Functional-shipping in ORCA [10] and in Manta [85] are two examples of such a distribution strategy. The approach is based on the observation that descriptions of operations generally can be expressed in fewer bytes than a language entity state description. However, the model is restricted in that operations must be side effect free. Consider what would happen if an operation is performed on 20 replicas and the operation is "send a document to a printer".

Figure 1.4 depicts the scenario from Figure 1.2 on a more detailed level. Language entity A is distributed by a single-instance operation passing distribution strategy (RPC). The instance is located at *node-1*. The location of the single instance is called the *home* of the distribution strategy. The

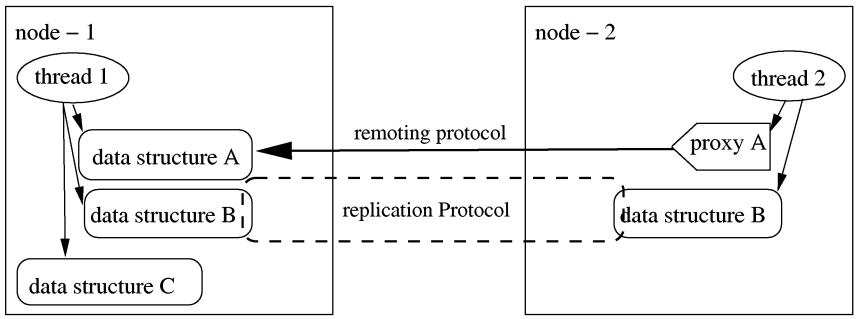


Figure 1.4: Data structures A and B distributed by the operation-moving and state moving approach.

home maintains an instance of the language entity that operations can be performed on. On any other node than the home, a reference to the language entity is represented by a proxy [113]. The role of the proxy is to pass operations to the home and wait for results of operations.

Language entity B is distributed by a multiple-instance state passing distribution strategy (single-writer/multiple-reader). Instance of B exist at both *node-1* and *node-2*. An operation performed on any of the instances representing B are executed by the calling thread at the local instance. If the operation results in a change of the state of the local instance, the new state is distributed to every instance representing the distributed language entity. For a node to acquire exclusive write access to the state every other instance has to be made unusable, this is called invalidation. It is the task of the distribution strategy to ensure that the instance is complete or complete enough such that the operation can be performed exactly as if the instance was a local language entity. Note that the instance is only required to be complete when an operation is actually executed on it. Typically, to ensure single-instance equivalence, a distribution strategy that supports local execution of operations suspends threads that try to perform an operation until the instance is in a complete and coherent state.

A distribution strategy that keeps replicas in a consistent state, no matter how, requires some type of arbitration to control access to the different

instances that represent the distributed language entity. The arbitrating functionality can be distributed among the nodes that hold references to a distributed entity. More commonly, the arbitrating functionality is located at a dedicated node, similar to the home of the operation moving approach.

Maintaining Single-Instance Equivalence

The purpose of a distribution strategy is to maintain single-instance equivalence for a shared language entity. Obviously, maintaining single-instance equivalence is straightforward for any strategy that maintains only one instance of the language entity (e.g. RPC and mobile state). However, for a distribution strategy that coordinates multiple replicas the single-instance equivalence can be violated if manipulations and access of the different instances are not properly coordinated.

First, consider a language entity distributed by a strategy that maintains multiple instances of the entity and updates the instances by sending new state descriptions. The language entity is represented by an instance at each node holding a reference. An operation on the entity is performed on the local instance. The operation does not have to be passed to a home as in an RPC protocol. Messaging is avoided when an operation on the language entity does not alter the state of the entity. We say that the operation *reads* the state of the entity. Following an operation that manipulates the state of the entity, called a *write*, the new state must be propagated to every instance of the entity. Due to delays in the network, the updates will not be reflected instantaneously in all instances. If no precautions are taken, different instances of the same language entity will be in different states at the same time. Concurrent writes from multiple processes can be observed in different order at two instances, i.e. different instances of the same conceptual language entity are potentially incoherent. For example, consider the two threads in Figure 1.4. Assume that *thread 2* first updates the state of data structure B and then updates the state of data structure A. Since data structure B is distributed using a replication type of distribution strategy, *thread 1* can first read the new value of A and later read the old value of B, something not possible at *node-2*. Consequently, single-instance equivalence is broken, in practice there exist two instances of data structure B in this scenario. The role of a distribution strategy is

to ensure that this does not happen, that incoherence does not occur.

A *consistency model* [1] can be seen as a description of how incoherent a single instance of a distributed language entity can be. On one hand, a restrictive consistency model, that permits little or no incoherence, restricts the concurrency in the system. On the other hand, a looser consistency model supports a high degree of concurrent invocations on the local instances. The drawback is that causally related reads at different nodes of a particular language entity can return different values (see example above). In addition, a more restrictive model generally requires more coordination between instances of a distributed language entity, thus a more bandwidth consuming and complex protocol. Below we discuss a subset of the existing consistency models of interest for distributed programming systems.

The *atomic consistency* model respects a global happened-before order between reads and writes. In other words, based on the notion of global time, a *read* of an instance should always return the effect of the last *write*. However, since achieving global time in a distributed system is costly, the atomic consistency model is impractical and is seldom used but for some special cases of distributed data.

A programming language is typical concurrent and threads concurrently access and manipulate the same language entities. The model of data manipulation is based on a notion of conceptually happened-before and not on time. The programming model allows an operation on an entity to be interleaved with two consecutive operations performed by another thread on the same entity. If it is necessary that two operations are not interleaved, the programmer instead explicitly synchronizes [123] when needed, for example using a lock. The *sequential consistency* model, defined by Lamport [81], encompasses the description of consistency of a concurrent programming language: *"The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operation of each individual processor appears in this sequence in the order specified by its program"*. For a detailed description of the consistency models we direct the reader to [122, 35]. Here we make use of examples of processes that manipulate data items to show the difference between the consistency models. *R(X)v* and *W(X)v* denote *read* and *write* operations to variable X with value v.

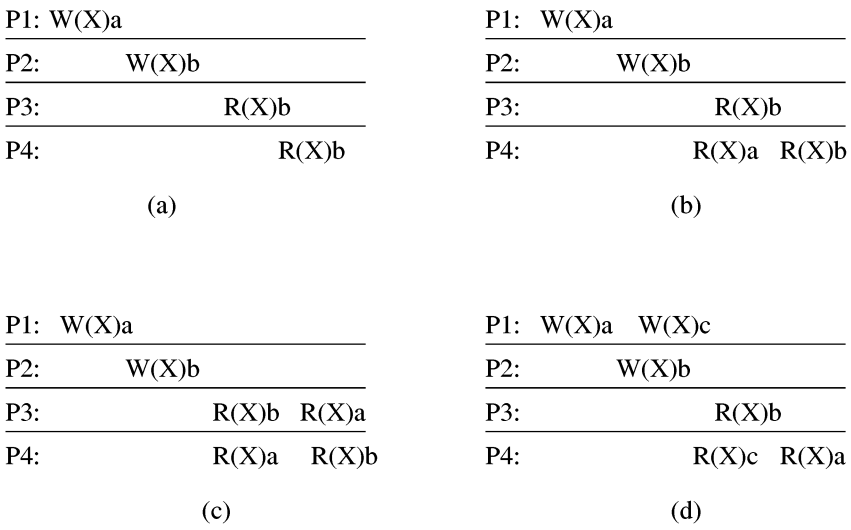


Figure 1.5: Four processes operating on one distributed data item X: (a) atomic consistency, (b) sequential consistency, (c) processor consistency, and (d) violating processor consistency (and thus also sequential and atomic consistency).

The operations on the variable X in Figure 1.5(a) are atomic consistent. Process *P3* and *P4* read the latest written value b. Figure 1.5(b) is an example of sequential consistency but not atomic consistency. Process *P4* reads the value a after *P2* has written the value b to X. This is not allowed in atomic consistency, but allowed in sequential consistency. The read operation of a is seen as if it had logically happened before the write of b to the variable X. Figure 1.5(c) is not atomic nor sequential consistent. *P3* and *P4* here observe the order of the two writes in different order, something that is not allowed in sequential consistency (and thus neither in atomic consistency).

The *processor consistency* model is weaker than sequential consistency, but nicely models asynchronous messaging in a system that experiences delay, it is sometimes referred to as *FIFO consistency* [104]. In short, the model guarantees that writes from one process are globally observed in the same order, but not that writes from different processes are globally observed in the same order. Consequently, Figure 1.5(c) is *processor consistent*, while Figure 1.5(d) is not since *P4* reads the writes of *P1* in the reverse order.

The three consistency models described above continuously maintain a particular coherence model. This has the advantage that shared data is always in a known coherent state, disregarding whether the data is used or not. If the distributed data is not used, keeping it in a coherent state is costly and simply a waste of bandwidth. The *release consistency* [50] and *entry consistency* [13] models are designed to overcome the above limitations. The two models are based on the notion of locks. Only when a lock is taken is the data guaranteed to be in a coherent state. Writes of data items while not holding the lock are not seen at other processes but only at the process where the update was performed. Any changes to the data are propagated after the lock has been released. If reads and writes of a distributed language entity are properly protected by locks, the two consistency models are equal to sequential consistency. Release consistency and entry consistency differ in that release consistency uses one lock for every distributed data structure of a system, while entry consistency supports multiple locks that monitor one or more data structures.

Distribution Strategies

An interesting observation is that different distribution strategies guarantee the same consistency model, e.g. both RMI and mobile state ensure sequential consistency. Thus, the semantics of a distributed language entity is not affected by choice of distribution strategy, as long as the strategy maintains the required consistency model. Changing between distribution strategies that adhere to the same consistency model maintains the functional properties of the language entity while altering the non-functional properties [108]. The functional property of a language entity is the service it provides. The non-functional property is how the service is provided. Security, fault tolerance, latency and bandwidth consumption are usually considered non-functional properties. As an example, an operation moving distribution strategy like RMI [89] requires a communication channel between the proxy and the home. Whether the channel to the home node is encrypted or not does not change the functionality of the remote object. However, it greatly affects the security of single remote operations. Encryption of the channel is a non-functional property.

The quote below nicely states why a distributed programming system should also support different types of replication distribution strategies:

> *"To maintain these copies [information manually replicated at multiple processes] in the face of distributed updates, programmers typically resort to ad-hoc messaging protocols that embody the coherence and consistency requirements of the application at hand. The code devoted to these protocols often accounts for a significant fraction of overall application size and complexity, and this fraction is likely to increase." [84]*

It is of great importance that a distributed programming system offers a large suite of distribution strategies to choose from when distributing an application [124, 111, 109]. If not, ad-hoc protocols will eventually be developed at application level to tune performance. Such solutions are generally not available outside the application. In addition, writing such protocols is a time consuming task that takes focus and resources from application development. The purpose of middleware is to remove the burden of im-

plementing network abstractions from the application developer, thus the service offered should be comprehensive enough.

1.2.2 The Underlying Network

The nodes of a distributed system are interconnected by an asynchronous network. For the remainder of this document, unless otherwise stated, we will assume Internet as the underlying network. In theory, the network is fully connected. Any two nodes can communicate by message passing disregarding their physical location. In practice this is not always true, as will be discussed later. On top of this core functionality, abstractions in the form of channels are provided. TCP is an example of such an abstraction. The channel abstraction hides details of message passing, including resending of lost or corrupted messages. The service is connection oriented and provides in-order delivery of messages. In addition, the channel abstraction can provide some type of monitoring mechanism that tells the status of a channel, e.g. connected, congested, destination-lost etc. Unfortunately some of the properties of the underlying network cannot be hidden, such as latency, since message delivery is not instantaneous. Depending on many factors, like the current network utilization, the physical distance between two nodes, and the size of the message, it will take a certain amount of time before a message sent to a node reaches its destination.

Despite the initial assumption on full connectivity, connectivity over Internet is in practice not always symmetric. Barriers in the form of administrative domain boundaries hinder connectivity. The possibility to establish a channel in one direction does not necessarily imply that a channel can be established in the opposite direction. This is commonly caused by NAT (Network Address Translation) and/or firewalls. For example, that node A is connected by channels to nodes B and C does not necessarily imply that a channel can be established between B and C. Moreover, the address of a node is not necessarily persistent during its lifetime and can change over time.

Secure communication over the underlying network is yet another challenge. In general, it cannot be assumed that all nodes of the network are friendly. Communication between two nodes that is routed over the

network can often be eavesdropped on by a malicious node. Moreover, malicious nodes can masquerade as being legitimate to acquire secret information. Encryption and authentication techniques can reduce the risk of information leakage to malicious nodes. Still, mistakes caused by the human factor, both during the development and the use of a distributed system can cause security "holes". These holes can be utilized by adversaries to cause harm to a distributed application.

In summary, the network environment, typically the Internet, is in theory fully connected, but in practice is not. Communication is asynchronous, subject to latency, and inherently not secure. In addition, little support for discovering link or node failure is provided.

1.2.3 Implementing Transparent Distribution

Ultimately, a distributed programming system should make programming a distributed application very much like programming a single processor application. If this is true, development, testing and deployment can be done in a controlled environment, i.e. at one machine, thus greatly simplifying development of distributed applications. For this to be true, the transparency must be functionally complete. That is, every first class data structure of a programming language must provide the same semantics when used in a distributed setting as when used centrally.

A distributed programming language that fails in providing transparency provides two computational models, one for centralized computing and one for distributed computing. This is inconvenient and also potentially dangerous. A distributed programming language that offers special constructs for distributed computing that syntactically resemble the constructs for centralized computing, but differ in their semantics, provides two models of language entities that look similar but behave differently. Such a programming model is hard to use and easily confuses the developer. In practice, a program developed, tested and proven to be correct at one machine, can contain errors that are hard to find when deployed in a distributed setting due to changed semantics of single language entities.

From the above discussion it is clear that transparent distribution is a beneficial property. However, it has been argued that transparent dis-

tribution is not possible to achieve [135], partial failures being the prime argument. Another argument is that hiding the underlying network results in inefficient applications [47]. Here we show that by introducing a concept of control, transparency can be provided, despite the two earlier arguments against transparency. Or, as pointed by Geihs, total transparency is not required [49] as long as the functional aspects of a programming system are preserved.

Partial Failures

Any distributed system deployed over a network is subject to failures. Nodes of the system become inaccessible because of node and link failures called partial failures. Partial failures are especially problematic in a system that provides a single system image, such as a distributed programming system that provides distribution transparency. Parts of the system are unavailable and any services located at the failed nodes are no longer available. This is clearly different from a centralized system that offers an all-or-nothing failure model, the system either provides service or the system does not. One approach to handle partial failures would be to impose an all-or-nothing failure model on a distributed system: if one node goes down, the whole system is taken down. However, such an approach would result in a system that scales poorly, as the chance that one of the nodes of the system fails increases with the size of the distributed system.

Another approach is to accept that a distributed system is exposed to partial failures and try to allow for the failures in the programming model. By exposing information to the programming level, actions can be taken to minimize the effect of failures and the application can continue to provide service. For example, consider how a person who searches information on the web handles a failed server. If the searched for web-server does not respond, another server is contacted that might provide the same or similar information [28]. We advocate that the same model should be provided when programming distributed applications, encompassing the benefit of partial failures: the whole application has not failed.

A distributed language entity is said to have failed when its associated distribution strategy has failed. In turn, a distribution strategy fails when one or more nodes necessary for the correctness of the protocol has failed.

For example, if the home node of a remote object fails, the remote object fails. Failure of entities should preferably be exposing to the programming level such that the application can react to the failure. Example of methods of signaling failed entities are throwing exceptions at invoked, or asynchronous signaling when a failure is experienced [110].

Efficiency

The choice of distribution strategies for the language entities of a distributed application is the dominant factor when considering the efficiency of an application [76, 8, 111]. To realize efficient distributed applications, the amount of interprocess communication and interprocess synchronization should be kept to a minimum. Because of the overhead of remote operations, operations should preferably be done locally. Consider a single-writer/multiple-readers distribution strategy: if the associated language entity is only read, the distribution strategy produces no messages. On the other hand, if the language entity is only updated, all replicas will be constantly invalidated and later updated. In such a case a remote execution type of distribution strategy is probably preferable.

A distributed programming system that transparently distributes language entities without the possibility to alter distribution strategies is known to hinder development of efficient applications [73]. Instead we argue that transparency should be relaxed, to a certain degree, and that a distribution strategy should be assigned on single language entity basis. This allows the programmer to assign a distribution strategy to a language entity based on expected usage pattern. If control over the distribution strategy can be introduced as an orthogonal aspect into a programming language, a distributed language entity may still be treated as a local language entity.

1.2.4 Distributed Programming System

Taking the view that a distributed programming system is a programming system integrated with a distribution support unit allows us to present a taxonomy of different distribution support techniques. Three different types of distribution support are recognized according to how a program-

ming system can be designed or adapted to host the distribution service. Hybrid systems can combine two (possibly all three) of these approaches. Moreover, a system that adheres to one approach can be used to implement another approach. Still we believe that a system can be predominantly classified as being of one type, with possibly some elements of another type.

The three approaches presented here bear some similarities to the taxonomy over distribution and concurrency support for object systems presented by Briot et al in [23]. Our taxonomy is organized from the perspective of transparent distribution support for open distributed systems. Furthermore, we are not only concerned with object oriented systems, but take a wider look at distribution support and also include other programming language paradigms.

The three identified models of distribution describe distribution support at three different levels of a programming system: (1) the *Shared Memory Approach* provides distribution from a level below that of the programming system, (2) the *New Entities Approach* augments the programming system on the application level, (3) the *Integrated Approach* extends a programming system from within.

The Shared Memory Approach

In this approach the nodes of a distributed application share a virtual memory space, similar to how multiprocessor-single-memory systems are organized. Conceptually each process reads from and writes to the shared memory as if the memory was local to the process. In practice, each node locally stores a replica of the shared memory; access to the shared memory is done on the local replica. A consistency protocol is executed to keep the replicas at the different nodes in a consistent state. For reasons of efficiency, the granularity of sharing is typically memory pages. To further improve performance, weaker consistency models than sequential consistency are commonly used, e.g. release consistency or entry consistency. A general assumption behind the shared memory approach is that sharing processes are part of a single concurrent program, running over homogenous hardware. The replication protocols used are communication intensive and require low latency and high bandwidth between the nodes of an application, i.e. typically over a bus or local LAN.

It is generally recognized that shared memory distribution support works well for some access patterns of shared data, but shows pathological performance degradation for other patterns [31]. The lack of information about program structure, i.e. data structures, at the level of distribution is the prime limitation of the Shared Memory model. This results in poor performance for distributing programming systems based on a structured programming model [70]. Furthermore, the model handles remote execution poorly and is primary limited to replication-type of distribution. Treadmarks [4] and InterWeave [84] are examples of systems that implement the shared memory approach.

The New Entities Approach

In this approach the programming system is extended using constructs available in the programming language, with data types and data structures that can be distributed. Typically, the original programming model is not altered, nor can it be distributed. A programming system extended by the approach results in a system that provides two programming models. The old model, that cannot be distributed and is primary used for centralized computing and the add-on to the old model that commonly resembles the old model syntactically, but differs in that it can be distributed. In effect, the approach makes a distinction between what is distributable and what is not distributable.

The advantage of the approach is that it makes it easy to implement and maintain a distributed programming system. Implementing the approach can be done at the application level, using the target programming system as platform. Moreover, access to the programming system internals is not needed. Thus development and maintenance can be done in a high-level programming environment. Dissemination is potentially simplified. If the programming system is operating system independent, the distribution support becomes operating system independent. The reason for the simplicity, the add-on characteristic, is also the major drawback of the approach. A programming system distributed by the new entities approach is a programming system with two programming models: one for centralized computing (the original set of language entities) and one for distributed computing (the new set of language entities).

The new entities approach is well suited for object oriented systems. The new entities are here represented by base classes. A user defined class is made distributable by inheriting from the distributable base class. Java RMI [89] is the prime example of an object oriented programming system that makes use of the new entities approach.

The Integrated Approach

The integrated approach can be seen as the middle ground between the shared-memory and extended-entity approaches. Distribution support is on the level of language entities, or more precisely, on operations on language entities. Implementing distribution support using the integrated approach requires modifying the programming system on the level of entity operations. Ultimately, if every operation on every language entity is supported for use in a distributed setting, the model supports potentially efficient transparent distribution of most of the programming system's language models.

The requirement to intercept operations on language entities and the requirement to interact with language entities is a potential drawback of the integrated approach. To be able to implement distribution support using the integrated approach, access to the target programming system internals is required. This is in difference to the new entities approach, which by definition can be constructed as an add-on to a programming system.

However, the model supports distribution of every language entity of a programming system, i.e. one single programming model for both local and distributed language entities. Whether a language entity is distributed or not can easily be expressed as a property. A language entity can start as a local entity and later be turned into a distributed language entity. Later still, the language entity can be made local (if possible without violating single-instance equivalence). Disregarding the status of a language entity (distributed or local) the language entity provides the same interface to the programmer. If associated with a distribution strategy that preserves the semantics, a distributed language entity provides the same semantics to the programmer as the local variant. Thus, the model can be used to implement a programming system that provides a common programming model

for local and distributed language entities. Mozart and Erlang are two distributed programming systems that are implemented using the integrated approach.

Distribution support on the level of language entity operations caters for efficiency. Context information regarding an operation on a language entity, i.e. read only or update of the entity state, can be used to optimize interaction with the associated distribution strategy. Moreover, since interaction between the programming system level and the distribution support level is not by reads and writes of memory (as in the shared memory approach), but by operations, operation passing protocols can easily be implemented (see Section 1.2.1).

1.3 Motivation and Thesis

From the previous section it should be clear that the realization of a distributed system is challenging not least because of the properties of the underlying network. We believe that many of the problems associated with distributed programming can be ascribed to limitations in the tools used, i.e. the distributed programming systems themselves:

> *"The increasing complexity of distributed software systems in the absence of corresponding advances in software technology fuel a perennial crisis in software."* [6]

How to distribute the object oriented paradigm is understood. The algorithms necessary to implement objects that can be accessed from multiple nodes are known; examples are remote method invocation and mobile state protocols. Still, few object oriented systems exist that implement programming models that provide transparent distribution. Java and C# are two examples of object oriented systems that provide two programming models, one for local and one for distributed objects. Except for a few research systems such as JavaParty [99] and Emerald [96], transparency does not seem to be the target of object oriented systems. CORBA and web-services are two attempts to standardize distributed object systems (remote objects) that focus on interoperability between different programming languages, and not on transparent distribution of objects.

We claim that even though distribution of the object oriented paradigm attained a lot of interest, the subject is far from fully explored. Other programming paradigms, such as the functional and the declarative-concurrent paradigms [133], have just partly been explored. We believe that the solution to *"... the absence of corresponding advances in software technology ..."* can be found in these and future, less explored, programming paradigms. To explore distributed programming using object oriented and other paradigms, new distributed programming languages and coordination mechanisms must be developed. The distributed logic variable [59] of Mozart for instance, that supports simple coordination of threads located at different nodes, is one example. The idempotent property of pure functions in Haskell [61], to create fault-tolerant remote execution [131], is another example. The possibility to distribute closures caused by lexical scoping introduced in Obliq [27] is yet another example of programming language constructs that was useful in a distributed setting.

This dissertation builds on the experiences from the distributed programming system Mozart [90]. Mozart showed that transparent distribution support could be achieved efficiently for a programming language. However, the monolithic nature of the implementation made maintenance and further development extremely time consuming. In addition was the distribution support in Mozart tightly integrated with the programming system, reuse of the distribution support code was impossible.

The goal of this dissertation is to provide generic tools in the form of distribution support for programming systems. The tools should not be restricted in what types of programming constructs can be supported, both constructs from the object oriented and constructs found in currently non- or poorly-supported programming paradigms should be supported. With such a tool at hand, we can compare the different models and evaluate and classify them according to what types of distributed programming problems they fit best. This will increase the number of available distributed programming systems and create knowledge of new efficient programming constructs for distributed programming.

To realize the vision of a generic tool of programming systems distribution support, distribution support should be clearly separated from the programming system implementation in the form of a middleware. The

middleware must provide a functionally complete model of distribution. That means that all language entity types found in the target programming languages should be supported. Moreover, the distribution support should be reasonably efficient to increase acceptance from application developers. In addition, the distribution support must provide features necessary for real-world applications, including automatic memory management and handling of network/node failures. Last, the tool must be simple to integrate with a programming system. This is summarized in our thesis:

> *Efficient multi-paradigm programming language distribution-support can be provided by a middleware.*

1.4 Contribution

This dissertation covers the design, implementation and evaluation of a Distribution SubSystem (DSS) middleware which is described by a number of research papers. The main contribution of the dissertation is, first, the design of a middleware that is complete enough to be used as a tool for creating efficient distributed programming systems offering transparent distributed programming models. Second, the middleware is designed to be coupled to, and integrated with, programming systems. The design of the interfaces of the middleware significantly reduces the effort of realizing a distributed programming system, compared to writing dedicated distribution support.

The efficiency of the middleware and the ease of integration are shown in an experiment where the middleware is coupled to a programming system. The resulting distributed programming system has shown good performance compared to other systems. We present evidence of the impact of the work presented in this dissertation on the research community in the form of published papers and use of the middleware as a tool for further research.

1.4.1 Scientific Contribution

The dissertation presents the design and implementation of a middleware for generic distribution support based on the notions of language entities,

threads, and their interaction. The novel concept of an *abstract entity* is presented. The abstract entity is based on the observation that different language entities, implementing different semantics in a localized computation, can be correctly distributed using the same type of distribution support. Objects of different languages (Java, C++, Ruby) even though semantically different can usually be distributed using the same distribution strategy. RMI is a good example of that. The differences that are of importance for distribution are captured in concepts of abstract entity types. Creating a distributed programming system is reduced to mapping language entities to the correct abstract entity types. The abstract entity interface makes the middleware generic, in that it can be coupled to any programming system.

A new approach to efficient distribution support of language entities is presented in a framework of protocols that implement distribution strategies. The functionality of a distribution strategy is separated into different aspects, called sub-protocols. A distribution strategy is composed of multiple sub-protocols. This design results in freedom of choice of sub-protocols. In addition, the clear separation of concerns simplifies extending the suite of protocols. Without this division, providing the functionality possible by sub-protocol composition as monotonic protocol implementations would result in a combinatorial explosion. The framework has made it simpler to realize numerous different distribution strategies, and thus implement efficient distributed applications.

The middleware design is fully decentralized and does not rely on external services. Nodes self-organize to facilitate services commonly located at dedicated servers, such as directory and yellow-page information. Since the middleware is not dependant on any infrastructure of services deployment of distributed applications is simplified. Firewall traversal, support for mobile nodes, and support for mobile language entities are examples of the services provided by the nodes of a distributed application. Techniques from the peer-to-peer domain are used. We show how structured and unstructured overlay network techniques can be used in a distributed programming system setting.

1.4.2 Proof of Concept

A middleware that implements generic distribution support for programming systems has been developed, the Distribution SubSystem (DSS). The DSS, is fully functional and implements the contributions described above.[2].

As proof-of-concept, the DSS has been coupled to an existing implementation of the multi-paradigm programming system Mozart which encompasses both the object oriented paradigm, the functional paradigm and the declarative concurrent (data-flow) paradigm. Mozart, even when not coupled with the DSS, implements transparent distribution of the data structures of the programming language Oz. In our experiment we have replaced the existing distribution layer of Mozart with the DSS. The result, the DSS extended Mozart version called OzDSS, can thus be compared with the original tightly integrated distribution support of Mozart. OzDSS is surprisingly efficient; the middleware approach only imposes an overhead in the range of a few percent. A significant benefit of OzDSS is that it supports customization of distribution strategy (of a large set) on a language entity level, something that is not possible in Mozart, which employs fixed distribution strategy for each language entity. The fine grained customization possible in OzDSS supports improvements in efficiency for some applications in the order of magnitudes compared to static allocation of distribution support [76].

1.4.3 Evidence of Impact

An implementation of the DSS middleware has been available for use as a research platform since autumn 2002.

- The programming system Mozart has been coupled to the DSS middleware resulting in the OzDSS [76, 74] system [3]. The OzDSS system shows that a distributed programming system created using the DSS can be efficient and provide a comprehensive distributed programming model.

[2]The implementation is freely available and can be downloaded from `http://dss.sics.se/`

[3]available for down load from `http://dss.sics.se/`

- An early version of the DSS has in an experiment been coupled to the .Net platform [91]. The objects of C# where successfully distributed using the abstract entity interface of the DSS and the reflective message-sink [103] interface of .NET.

- In close collaboration with researchers at UCL in Belgium the DSS has been extended with a communications infrastructure that can handle asymmetric connectivity [75].

- The success of the OzDSS prototype has initiated a project at UCL that will replace the distribution support in the official release of the Mozart system with the DSS. This is currently ongoing work being conducted by Boriss Meijas.

- The DSS is used as a functional component in one of the demonstrators for Pepito `http://www.sics.se/pepito`, a Fifth Framework EU FET project. In the same project a Java interface has been developed on top of the DSS middleware by researchers in Lousanne, making the distribution model presented in this thesis available to the Java community.

1.4.4 My Contribution

From an initial document that described the vision of generic distribution support [22] jointly written by Per Brand, Seif Haridi, Konstantin Popov, and myself, I have realized the vision in the form of the design of the DSS. I have been the main author of all but one of the papers included in this dissertation. A more detailed description of the contributions by me to each included paper can be found in Chapter 5. The implementation of the DSS middleware has been a joint effort by Zacharias El Banna and me.

1.5 Organization of the Dissertation

This Dissertation is based on eight papers: four peer-reviewed, one to be submited for review, and three technical reports. The papers are found as appendices. Chapters 2, 3, 4, 5, 6 serve as an introduction and overview

to the research presented in the attached papers. Chapter 2 presents an overview of existing approaches to distributed programming systems and to distribution support systems. The architecture of the DSS is described in Chapter 3. The resulting middleware is presented from a programmers point of view in Chapter 4. Chapter 5 introduces the attached papers and presents a short overview of how each single paper contributes to the overall design and description of the DSS. The dissertation is concluded in Chapter 6, which summarizes and highlights important experiences from the work on the DSS and also points at further research directions.

Chapter 2

An Overview of Distributed Programming Systems

This chapter gives an overview of systems that provide distribution support. We differentiate between distributed programming systems and distributed support systems. The distributed programming systems are compared according to completeness and transparency of their distributed programming model. Distribution support systems are considered from the perspective of using them as tools to create programming systems. For distribution support systems we have chosen to consider completeness in the distribution model as well as the convenience of integrating the system with a programming system.

For the purpose of this dissertation, interoperability between different programming languages is not considered. Ideally, code written in different programming languages can be executed at separate nodes connected as a distributed system. The appearance of one single system, without any language barriers, is achieved by providing access transparency through the use of the distributed objects. Programming language interoperability is important for applications that cannot be developed in one single programming language. However, our ultimate goal is to provide a middleware that can be used to create a distributed programming system based on any existing programming system. The first step towards this goal is to create distributed programming systems that can connect nodes executing

Design Approach	*Target Environment*		
	Cluster	*LAN*	*Internet*
Integrated	Erlang	Obliq	Mozart
New entities	JavaParty	Java RMI	Manta, Globe

Table 2.1: Situating the presented distributed programming systems

the same programming language. Programming language interoperability is considered as future work.

2.1 Distributed Programming Systems

For a programming system to classify as a distributed programming system distribution support should, to some degree, be integrated into the programming model of the programming language. Another definition of a distributed programming system is that distributed programs can be developed, to various degrees, without considering the distribution of data structures and threads, i.e. transparency.

Of the large number of existing distributed programming systems [102, 120, 131, 89, 10, 88, 80, 140, 66, 127, 96, 38], we have chosen a subset that is developed either by the new entities or the integrated approaches described in Section 1.2.4. Since the shared memory approach is intended for a cluster of homogenous workstation environments, we have deliberately not included any distributed programming system based on that approach. In addition we have chosen to differentiate between systems designed for clusters, LANs, and the Internet. Table 2.1 depicts the systems chosen and what type of network environment and design approach they are intended for.

To reason about different types of distributed programming systems we look at how well integrated distribution is into the programming model, i.e. how **transparent** the programming model is. For practical reasons, the distribution support must provide a comprehensive service, therefore we look at how functionally **complete** the distribution support is. Section 1.1.3 describes the concept of transparency in distributed programming

Distributed Programming System	*Transparency*	
	Network transparency	*Replication transparency*
Java RMI	partly	no
JavaParty	partly	yes
Globe	partly	yes
Erlang	fully	-
Mozart	fully	yes
Obliq	fully	yes
Manta	no	-

Table 2.2: Implemented transparency of the surveyed systems.

systems. According to the description found in Section 1.2.3, we exclude failure transparency in our requirements for a distributed programming system. Instead we argue that a distributed programming system should detect and report failures to the programmer and is therefore considered a completeness property. Table 2.2 summarizes how transparent the different evaluated systems are according to the following requirements:

Network transparency allows access to distributed language entities without explicitly considering their location and their exact representation.

Replication transparency means that replication of data is supported with preserved semantics. That is, single-instance equivalence is preserved even if local access of replicated data is allowed.

We use completeness to describe the service provided in the non-functional domain. For a distributed programming system to be useful as a tool for writing distributed applications that are deployed on the Internet, it should fulfill as many of the completeness requirements as possible. Table 2.3 summarizes the completeness of the evaluated systems according to the following requirements:

Failure Detection is a requirement to be able to develop distributed applications for networks that experience failures. A distributed programming system must first detect and differentiate between different link and node failures. Second, failures must be reported to the programming level such that actions can be taken at application level, see Section 1.2.3.

Internet Communication puts demands on the communication support of a distributed programming system. For a distributed programming system to qualify as being capable of handling Internet type of communication it should be able to establish and maintain connections in the face of temporary loss of connectivity, node migration and asymmetric connectivity, see Section 1.2.2.

Choice of Distribution Strategy for single language entities is the key to efficient distributed applications, as described in Section 1.2.3. Different distribution strategies should be provided, and ideally, it should be possible to implement new distribution strategies.

Dynamic Reference Handling is necessary for scalability of a distributed application. Only nodes that have an interest in a particular distributed language entity should participate in the consistency protocol of the entity. Nodes should learn about new distributed language entities by reference passing. In addition, distributed language entities should be garbage collected when no longer referred from any nodes.

2.1.1 Java-RMI

Java [52] has become a *de facto* standard for modern object-oriented distributed language design. The language model is extended with distribution support by the `serializable` and the Remote Method Invocation (RMI) interfaces [89]. Distribution support is integrated into the programming model using the new-entities approach. Thus, an ordinary Java object cannot be sent over the network. Only annotated objects can be used in a distributed setting. The two types of distributable objects, serializable and

Distributed Programming System	*Completeness*			
	Failure detection	*Internet communication*	*Reference handling*	*Distribution strategies*
Java RMI	simple	no	yes	no
JavaParty	simple	no	yes	yes
Globe	simple	no	yes	yes
Erlang	simple	no	no	no
Mozart	advanced	yes	yes	no
Obliq	no	no	yes	no
Manta	no	no	no	no

Table 2.3: Implemented completeness features of the surveyed systems.

remoting, alter the semantics of objects when used in a distributed setting. A serializable object is sent by value when transferred over the network, resulting in an uncoordinated replica of the original object. A remoting object, on the other hand, is passed by reference, preserving single-instance equivalence.

Java-RMI serves as a good example of the limitations of the new-entities approach. Only annotated objects can be used in a distributed setting, network transparency is only supported for a subset of the language entities. In addition, a design decision in Java has affected the programming model. Replicated objects are supported over the serializable interface, while replication transparency is not supported. In addition, support is missing for programming constructs such as feature access and reentrant locking [24]. Limitations in the distribution support of Java RMI make the system unsuitable for Internet type of distributed applications. Only one type of distribution strategy is supported, remote execution of methods on stationary objects. Node and link failures are detected, but there is no differentiation made when reported to the programming level. Still, the dynamic nature of Java that supports replacement of software modules and the choice of the new-entities approach make it possible to extend or replace the distribution support code [57, 101].

2.1.2 JavaParty

JavaParty is an efficient implementation of Java RMI for cluster environments. Of the many systems that improves the Java RMI model [128, 127, 43, 66, 39], JavaParty [99] is one of the more comprehensive systems. Java Party has been developed over many years with the goal of providing an efficient and coherent implementation of the remote object model of Java RMI. The system uses the new entities approach.

The serializing routines of Java RMI are reworked together with the remote execution code to increase performance [92]. Replication transparency that is lacking in Java is provided in JavaParty [63]. In addition, the concept of logical thread identities are introduced [62], allowing for reentrant locking and control of threads in a distributed setting. Distributed objects in JavaParty are not restricted to remote execution, but can be distributed by different distribution strategies, including different types of replication and mobile state.

In summary, JavaParty has the characteristic limitations in transparency associated with the new-entities approach, only a subset of the language entities are supported under network transparency. In contrast to the original Java-RMI implementation, replication transparency is provided. The system targets clusters and thus provides no support for Internet communication, one of our requirements for a complete system. The failure detection and reporting is similar to Java RMI.

2.1.3 Globe

The Java based system Globe [9] targets wide area computing. Globe is built on the observation that certain properties of how an application accesses shared data should be reflected in the choice of distribution support. This is presented to the programmer as the ability to specify the distribution behavior on object basis. Various aspects of an object's distribution strategy can be specified, including different types of replication protocol, fault tolerance and security.

Globe fulfills many of the requirements for a complete distribution programming system. However, the comprehensive distribution support is integrated into Java using the new entities approach. The remote objects

interface is replaced with the distributed objects interface, thus Globe has the same limitations when it comes to transparency as Java RMI.

2.1.4 Erlang

Erlang [42] is a functional and concurrent programming language primarily designed for telecom applications [5]. The language model supports processes that communicate by message passing. Concepts of distributed programming are elegantly included into the programming model [138]. Processes can communicate by message passing disregarding their physical location, i.e. network transparency.

The target environment, clusters of workstations, is reflected in limitations in the implementation of distribution support Erlang. Until the most recent version of Erlang a system of Erlang virtual machines could consist of a maximum of 256 nodes. In addition, the messaging implementation is not designed for the Internet. The failure detection does not differentiate between link and node failures.

The Erlang system is implemented by the integrated approach and the high degree of transparency is ascribed to this. Erlang serves as a good example of how a programming model without shared state can give network transparency.

2.1.5 Mozart

Mozart [90] is an implementation of the multi-paradigm programming language Oz [133] that simultaneously implements the functional, declarative-concurrent and object oriented programming paradigms. Mozart is designed for concurrent programming and provides an efficient implementation of threads, together with logic variables from the declarative-concurrent paradigm for data-flow synchronization.

Distribution in Mozart is implemented using the integrated approach and supports network transparency for all language entities found in the language Oz. Shared state abstractions, asynchronous message passing abstractions and data-flow abstraction, are supported by different protocols [59, 132]. In addition, Mozart implements various kinds of replication of immutable data structures and thus supports replication transparency.

Distribution support in Mozart is designed for the Internet. The runtime system differentiates between link and node failures. Moreover, asynchronous connectivity can to a certain degree be handled. The model of distributed language entity references fulfills the dynamic reference handling requirement. Different language entity types are assigned different distribution strategies. However, there is no support for changing distribution strategy on a single language entity basis. Nor is process migration supported.

2.1.6 Obliq

The distributed programming system Obliq [27] was, different from most other existing distributed programming systems, explicitly designed for distribution. Obliq was developed with the purpose of exploring how transparency could be included into the programming model of an object oriented programming language. The programming model is object oriented with the addition of lexical scoping, i.e. the possibility to dynamically create closures of data and code in runtime. Obliq implements network transparency for the complete programming model. Objects are distributed by the remote execution distribution strategy, which also support migration of objects. The immutable closures are replicated under replication transparency.

Obliq is built on top of the object oriented distributed programming system Modula-3 [18], which provides support for distributed objects. Obliq is implemented using the integrated approach. The use of Modula-3 facilitated the implementation of completeness of the distributed programming system. Modula-3 supports remote method invocation and automatic garbage collection of distributed objects. Still, Obliq lacks support for handling link or node failures. In addition, the distribution strategy is fixed for the different language entities. These limitations in completeness probably stem from limitations in the Modula-3 implementation.

Apart from being one of the first distributed programming languages to provide full network transparency at the programming level, the approach taken for implementing the system deserves extra focus. Instead of developing dedicated distribution support for Obliq, an existing system, Modula-3

network objects, was used. This relieved the development process from the challenges of developing distribution support. Instead, focus could be put on developing the programming model.

2.1.7 Manta

Manta [86] is a cluster computing system based on Java. Instead of distributing the model of the programming language, the Manta distributed programming system introduces the concept *distributed objects*. A distributed object is at creation replicated to all nodes of a distributed application. The replicas are kept consistent by method shipping described by the operation-passing multiple-instance model in Section 1.2.1. A method invoked on a distributed object is passed to, and executed at each replica, and is required to be to be side-effect free. To avoid side effects, a distributed object is not allowed to refer to other distributed objects.

For an object to be distributed it must be annotated. The programming model does not support transparency since local objects are differentiated from distributed objects. This is intentional as the Manta developers argue for the strength of non-transparency when developing efficient distributed applications [73]. Transparency is intentionally discarded in favor of efficiency.

Manta is implemented using the integrated approach. Communication primitives are provided by the Panda [15] middleware. The implementation of Manta is shown to be stable and efficient. However, the implementation fulfills few of our requirements for a complete system. Manta is designed for high-performance cluster computing. The distributed object protocol is built on the assumption of a low latency communication medium. No garbage collection exists for distributed objects. Finally, no support exists for detecting or reporting node or link failures.

2.2 Distribution Support Systems

In this section we present a selection of middleware that are of potential interest for use as distribution support systems. We have chosen systems that represent different paradigms of distribution support.

Distributed Suport System	*Supports*		
	Mutables	*Immutables*	*Streams*
MoM	no	no	yes
CORBA	yes	no	partly
Web Services	no	no	no
DotNET	yes	yes	no
Interweave	yes	yes	yes

Table 2.4: Expressiveness in distribution support of the surveyed distribution support systems.

The different paradigms are considered from how expressive their distribution support is. We use expressiveness to describe how well the distribution support provided matches the language entities of a programming system. For a system to be expressive, integrating it with a programming system should require little translation between distribution support units and language entities. To be able to reason about distribution support systems expressiveness, we look at how well three typical language entities are supported. The results are summarized in Table 2.4 and the three types of language entities are described here:

Mutables are data structures whose state can be altered, examples are the state of an object, or the value of a vector element.

Immutables are data structures whose state cannot be altered after creation. Classes in Java and atoms in declarative languages are examples of immutable.

Streams are data structures that describe a producer/consumer behavior. A producer adds items to the stream and the consumer reads items from the stream as items become available. Messaging abstraction like ports in Erlang is an example of stream type language entities.

Middleware that implements node to node messaging can be used to augment any programming system with distribution support. However, to

simplify development of distributed programming systems, the provided distribution support should at least directly support the above three types of language entities. The expressiveness of the different distribution support systems is our primary focus in the evaluation, but we also to some extent discuss the different systems completeness, according to the classification found in Section 2.1.

2.2.1 Messaging Oriented Middleware

Message Oriented Middleware (MoM) is a family of middleware that provide network transparent messaging abstractions. Messaging abstractions can be point-to-point as in the Message Passing Interface (MPI) or be of group communication type [105, 67]. With few exceptions, MoMs are designed with the focus on providing good performance and a high degree of customization. Performance can be measured as delivery time of a message [21], the size of a message or the number of times a message is copied between different internal buffers. Customization is commonly implemented by component systems, where communication protocols are constructed of sub-components [46, 45, 16]. In addition, there exists MoMs that provide relocation transparency, that is, connectivity can be provided even if nodes physically migrate [117, 141, 98].

MoMs are messaging wise complete. Typically they target multiple areas of communication, from high-speed intra-cluster communication to location-independent communication. However, the distribution service provided is restricted when considering the demands for an expressive distribution support system. Only the stream type of language entity is directly supported. Any other type of distribution support must be explicitly programmed on top of a MoM.

2.2.2 CORBA

The Common Object Request Broker Architecture (CORBA), is the specification of a distributed object system [97] that focuses on programming language interoperability. Different object oriented programming languages can use CORBA as an interoperability fabric. Remote objects is the usual technique to implement distributed objects in CORBA systems, although

implementations exists that provide replicated objects [34]. The CORBA standard is comprehensive, and can be seen as a wish-list of every functionality and conceivable operation that is possible with remote components [103]. The sheer amount of functionality that has to be implemented for a distributed programming system to be CORBA compliant is a strong argument against CORBA. However, C++ implementations, such as TAO [80], exist that can be used as generic distribution support middleware.

Classifying CORBA according to expressiveness and completeness is complicated since CORBA is a specification, and not an implementation. Moreover, numerous implementations exist, that extend the CORBA definition in various directions. Here we try to look at the specification only. Distribution support is based on objects, thus the mutable type of language entity is supported. Asynchronous, unreliable messaging is supported in the standard. Thus, to a limited extent, the stream type of language entity can be supported, with the requirement that the messaging is made reliable. However, no support exists for replication of data structures.

The CORBA specification is designed for interoperability and solves integration of different applications by remote execution. How replication or migrations of objects are handled is not defined in the specification. Thus, a basic CORBA system does not fulfill the completeness requirement of different distribution strategies for a language entity.

2.2.3 Web Services

Web Services [134] are an attempt to standardize application to application communication over the Internet. The unit of interaction is a service that has a known network location and exposes an interface of operations it can perform. Interaction between services is done by remote invocation using the SOAP (Simple Object Access Protocol) standards [115], with data encoded in XML (eXtensible Markup Language) [40]. The use of these standards makes it possible for different systems to interact, thus achieving interoperability.

A service, the unit of distribution, is a different concept than a language entity. The model is explicitly client/server oriented in that a SOAP object

models a server, its functionality is described in WSDL (Web Service Description Language). SOAP objects are stateless; invocations on a SOAP object should not change the object state.

Similarly to CORBA, web services are a standard. Looking at the standardization only, we find that web services have poor support for mutable language entities, the services are stateless. In addition, the standard fails to fulfill the completeness requirements of detection of link failures. HTTP does not differentiate between link and node failures. In addition, no support for distributed resource management in the form of garbage collection is described. In summary, web services are a framework for developing loosely coupled Internet applications, not for developing distributed programming systems.

2.2.4 Dot NET

The purpose of the .Net framework [88] is to provide a platform for object oriented programming over the Internet [44]. Automatic loading of objects and component descriptions at runtime is implemented as core functionality by the platform. The model of distribution, remote objects, supports both intra .Net distribution and seamless interaction with SOAP objects. Consequently a remote object can either exist at another node executing the .Net framework or be a SOAP object residing at a web-server.

Central in the .Net platform is the common language runtime (CLR). CLR implements core functionality such as memory management, thread management, remoting and management of language security. The CLR executes annotated byte code. Any programming language that is compiled into CLR byte code can be executed on the .Net platform. In addition to the core functionality, the .Net platform implements a class library available to any program compiled for the CLR.

To use the .Net platform as a generic distribution platform requires a programming language to be compiled to CLR byte code and executed on the .Net platform. As long as the model of a programming language matches the model provided by .Net, this is a viable approach. However, the platform is strongly geared towards statically typed, object oriented programming languages.

Distribution support is object oriented and resembles the model of JavaRMI. Two types of distributed objects are supported, remote objects and replicated objects. In addition asynchronous messaging is supported. Consequently, the provided distribution support can handle the three types of language entities required for a system to be expressive. However, the implementation is incomplete. Link and node failures are treated as one type of failure. Only one type of distribution support exists for distributed objects. Despite the limitation in completeness, the .Net platform is interesting because of its high degree of customization [88].

2.2.5 Software Distributed Shared Memory: InterWeave

Software Distributed Shared Memory(S-DSM) systems are primarily geared towards parallel computation over clusters of workstations located at a single LAN [4]. The type of distribution support, a shared virtual memory among the processes of the distributed system, fits programming languages such as C and FORTRAN. Still, the model has been applied to other programming languages, for example Java [140]. InterWeave [33] is a system that attempts to provide the S-DSM model of distribution for a wider group of programming languages and in other distributed environments than clusters of workstations.

Sharing memory between processes demands a common memory representation, thus S-DSM systems have traditionally been targeting homogenous systems, i.e. processes executing on the same type of hardware, on the same operating system. Interweave is designed to overcome this restriction associated with the S-DSM approach. The distributed shared memory is represented in a machine independent format. The machine independent structuring is on the level of primitive types, such as integers, pointers and floats. A layer between the application and the memory translates the generic format into a machine or software specific format [125].

A problem related to the S-DSM approach is *false sharing* [130] that stems from granularity differences between application data structure size and block size of shared memory. To overcome the problem of false sharing, InterWeave supports distribution of dynamically sized, fine-grained memory blocks called segments [33]. To further minimize false sharing, a second

level of memory access is provided, called views [31]. A view allows manipulation of parts of a segment and thus minimizes memory dependencies between different processes. Two processes can simultaneously access the same segment using non-overlapping views. InterWeave supports the definition of distribution strategy per segment and new distribution strategies can be added to the InterWeave system.

The concept of segments, views, and customizable distribution strategies for each view fulfills the completeness requirement of multiple distribution strategies. Despite these novel features of an S-DSM, InterWeave shares many of the limitations found in traditional S-DSM when it comes to Internet type distribution support. No support exists for failure detection or for proper reference maintenance. However, the distribution support is expressive. The appearance of a shared memory is provided. Any language entity that can be implemented in a single processor memory can be directly implemented in the distributed memory provided by InterWeave. Consequently, InterWeave supports the three different types of language entities that were required for a system to be expressive.

2.3 Conclusion

From the survey above we conclude that few of the systems implement transparency and a complete distribution model. This, we believe, is because of the sheer complexity of developing distribution support for a programming system. A natural approach would be to make use of an existing distribution support system. Again, none of the distribution support systems discussed is expressive enough to be coupled to arbitrary programming systems and provide a distribution support service complete enough for Internet type of distributed computing. The systems that exist are either specialized for a certain programming paradigm or not suited for the Internet. It is this limitation in complete distribution support for Internet applications this thesis remedies, the need for a generic distribution support system that can be easily coupled to any programming system and provide distribution support that can be deployed over Internet.

Chapter 3

Architecture of the Distribution SubSystem

To show the feasibility of our thesis *"Efficient multi-paradigm programming language distribution-support can be provided by a middleware."* we have designed and implemented the Distribution SubSystem (DSS) which provides efficient distribution support for programming systems. The DSS interface is based on abstract representations of programming language constructs, by abstract representations for language entities, operations on language entities, and programming system level threads. By using the DSS, development of a distributed programming system is reduced to connecting language entities and threads in the programming system with their abstract counterparts in the DSS. g This chapter presents an overview of the DSS middleware. References in the text point the reader to more elaborate descriptions found in the attached papers. Section 1.2.4 presents the background to the design of the DSS, including our assumptions and the reasons for choosing the integrated approach. Internally the DSS is structured in three layers, see Figure 3.1. Section 3.2 describes the *abstract entity layer*, which provides a generic data structure interface. Section 3.3 describes the *coordination layer*, which is responsible for coordinating movement of operations on, and replication of language entities according to a given consistency model. The different types of distribution strategies are implemented by this layer. Section 3.4 describes the *messaging layer* which

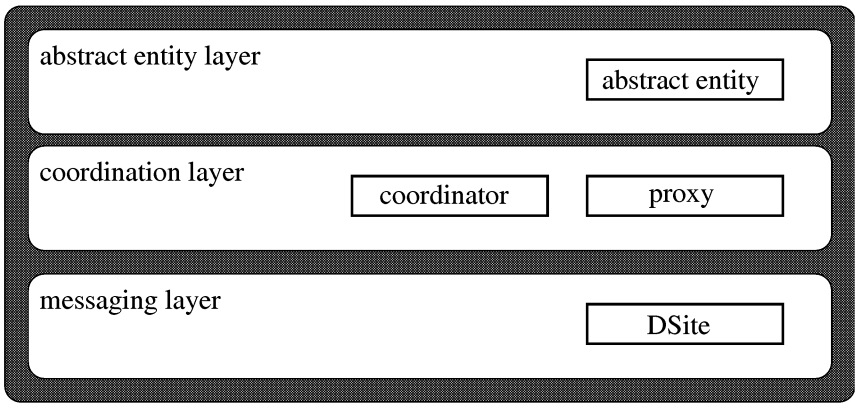


Figure 3.1: Internal layout of the DSS. The three conceptual layers are depicted together with the key component(s) of each layer

provides communication support for the middleware. The messaging layer handles nodes joining, leaving, and failing. Moreover, the layer is prepared for custom implementations of services such as connection establishment and failure detection.

3.1 Design Decisions

This section describes some of the design decisions of the DSS. The reason for the choice of type of distribution support is discussed. Assumptions on the targeted programming systems are also presented.

3.1.1 The Integrated Approach

Three different design approaches to distribution support for a programming system were presented in Section 1.2.4: the new entities approach, the integrated approach, and the shared memory approach. In order to argue for the approach chosen in the development of the DSS, the three approaches are compared side by side. The comparison is done with respect to three properties that we have identified as important for programming

system distribution support. First, how well does the approach support transparent distribution, which is equal to access, location, replication and migration transparency for the whole programming model. Second, to what degree is programming language level structuring preserved in a distributed setting, which is necessary to support instrumentation in the domain of non-functional properties on a single language entity basis. Third, how easily can an implementation of the distribution support be coupled to an existing programming system. Table 3.1 summarizes the findings.

Transparency: The shared memory and the integrated approaches both can support transparent distribution for every data structure of a programming language. The shared memory approach provides an easy to integrate and use distribution support model, if the memory is shared, every data structure, disregarding type or semantic model, is shared. The integrated approach requires the implementation of explicit distribution support for all data types of a programming system, but supports fine grained specialization of distribution behavior.

The add-on property of new entities approach hinders complete integration with the target programming language. Consequently, complete transparency is hard to achieve. As previously mentioned, the new entities approach usually renders two programming models, one for local computation and one for distributed computation.

Data structuring: The shared memory approach provides unstructured distribution support on the level of reads and writes to a shared memory, thus, application level structuring is not reflected at the level of distribution support. While this may work well for unstructured data, it has been shown that this limitation results in inefficient distribution of programming systems that are based on stronger, e.g object oriented, structuring principles [70].

Both the integrated and the new entities approaches provide distribution models based on language entities and supports simple realization of multiple distribution strategies for each single language entity.

Integration with a programming system: Unless the programming system provides reflective and introspective support that makes it

Design Approach	*Requirement*		
	Transparency	Data *Structuring*	*Integration*
New Entities	no	yes	simple
Shared Memory	yes	no	simple
Integrated	yes	yes	complicated

Table 3.1: Comparision of the three distribution support approaches

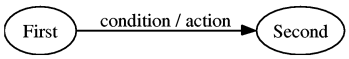


Figure 3.2: Notation for state diagrams.

easy to inspect the state of single language entities, the integrated approach requires fundamental rewrites of the programming system. The new entities and the shared memory approaches are, however, simple to integrate with an existing programming system.

The DSS supports the integrated approach. However, the DSS is supposed to provide a generic service, and the integrated approach does not cater for simple integration with a programming system. Thus, one of the primary challenges in the design of the DSS is the interface to programming systems. The interface must be specialized enough to capture programming system structuring and generic enough that the middleware can be coupled to different programming systems. While integrating a programming system with DSS can be expected to require considerable effort, the goal is minimize this effort.

3.1.2 Properties of Targeted Programming Languages

Ideally, the DSS should support all types of programming systems. This includes different implementations of programming languages of the same programming paradigm and implementations of programming languages that supports multiple programming paradigms.

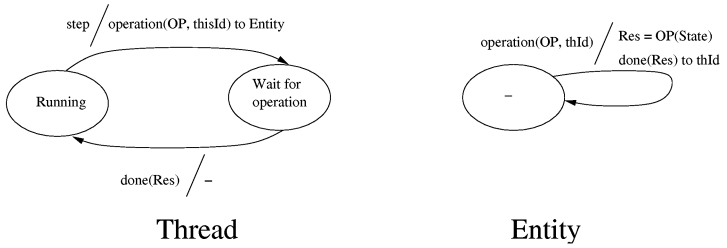


Figure 3.3: The interaction between a thread and a language entity modeled as a system of concurent units communicating by message passing.

We will use state transition diagrams to model the interaction of various entities. A state transition diagram is a finite state automaton. It consists of a finite set of states and transitions between the states. An entity starts in an initial state, and changes state as a result of a transition. A transition takes place when a message is received and a guard condition is true. As a result of a transition, messages can be sent and the internal state of the entity can be changed. Figure 3.2 shows the graphical notation. Each circle represents a state. Arrows between the circles represent transitions.

The DSS provides distribution support for programming languages which have threads (one or more). Threads execute instructions which result in operations on language entities. Figure 3.3 shows the state diagrams describing the interaction between a thread and a language entity. Internally, the thread maintains a program counter pointing to the current instruction. Some instructions make the thread perform operations on language entities. An operation on a language entity may or may not alter the state of the entity and may or may not return a value. In our model, an operation always returns a result. An operation not returning a result would be modeled as returning an empty result.

To enable efficient distribution support, the programming language should be *reference secure*. Reference passing should be explicit. A thread can only access data structures it has either created or received a reference to from another thread. A thread cannot guess or forge a reference to a data structure. A distributed programming system that is reference secure

can make a clear distinction between distributed data structures and local data structures. In a distributed programming language which is not referential secure, every data structure is potentially available to any thread of a distributed system. The distribution model required for such a programming language is a distributed shared memory, and not a middleware that supports the integrated approach.

3.2 The Abstract Entity Model

This section describes one of the main contributions of this dissertation, the interface of the DSS to a programming system. The interface is designed to simplify integration of data structures to distribution strategies and in the same time support detailed customization of non-functional properties. A key challenge for the DSS is the degree of specialization and expressiveness of the API. A more generic interface makes the middleware useful for a larger set of applications, but typically requires code that adapts an application to the middleware. A specialized interface, on the other hand, that on a detailed level models particular constructs makes it simple for a restricted set of applications to make use of the middleware. The DSS API simultaneously targets the properties of a generic and a specialized API. Many different programming systems should be supported and in addition, integration with the DSS should require little adaptation code.

3.2.1 Distributed References

The distribution model offered by the DSS is based on distributed language entities and references to distributed language entities. References are passed between nodes as a result of operations on distributed entities, arguments to operations on distributed language entities, or by explicit bootstrapping of a distributed system. A node learns of language entities by reference passing and maintains a set of references to distributed language entities referred to from the node. How the set is calculated depends on the type of programming system. For example, in a programming system that implements threads but no global variables, the set of distributed language entities is the union of distributed language entities referred from

the threads at the node.

A remote reference is represented at a node by an instance of the distributed language entity, called a *local entity instance*. The local entity instance acts as a proxy for the distributed language entity, implementing the appearance of an ordinary language entity is similar to fragmented objects [87] and truly-distributed objects in Globe [9]. An instance is created when a reference is received and removed when the reference to the distributed language entity is no longer needed at a node.

Guarded Replication

To maintain single-instance equivalence for a distributed language entity represented by multiple local instances, uncoordinated reads and writes of local instances should be prevented. To coordinate operations performed on local entity instances, which represent one distributed language entity, a conceptual guard is attached to each instance. The guard intercepts operations on the local entity instance and consults the DSS on how the operation should be resolved. We call this model *guarded replication*. The guarded replication model simultaneously supports both remoting and replication types of distribution strategies, see Section 1.2.1.

Figure 3.4 depicts the thread model found in Figure 3.3 enhanced by a guard. The language entity model is the same, a guard is introduced and the thread is extended with new states. Whenever a thread performs an operation on an entity, it has to query the guard associated with the language entity. The result from the guard tells the thread how to proceed. The model of the guard presented in Figure 3.5 is an abstraction. The purpose of the model is primary to show the interaction between the thread and the guard. How the guard decides to allow local access for a calling thread or to suspend a calling thread will be discussed in greater detail throughout this chapter.

The guarded replication model separates the distribution status from the state of the language entity and makes it possible to regard it as a non-functional property. The same representation suffices for a language entity whether local or distributed. A local language entity is connected to a guard that is in the `Local` state which always return `doLocal`. A local language entity is turned into a distributed language entity by an

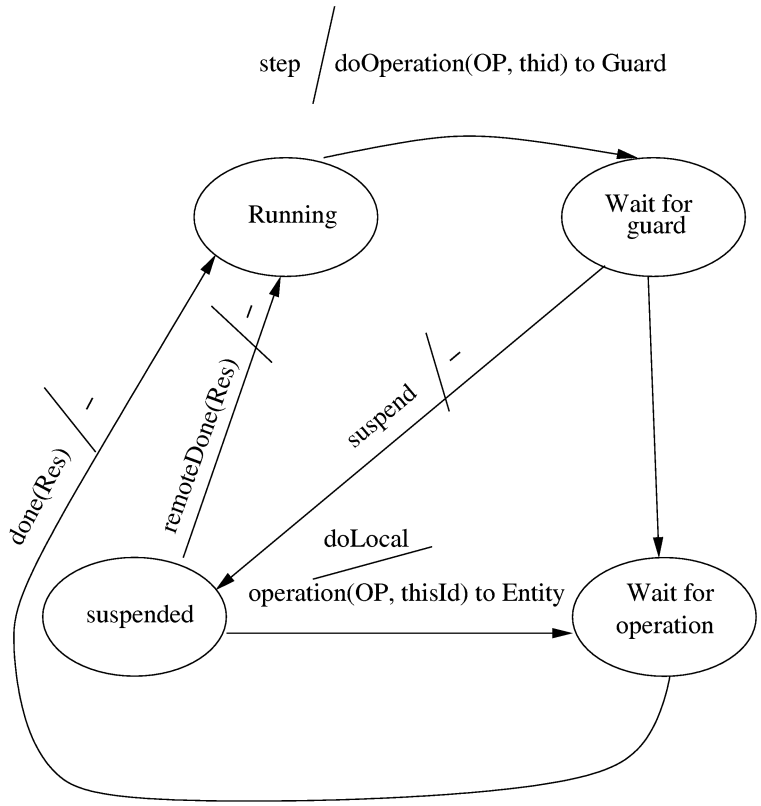


Figure 3.4: The state diagram for a thread that is extended to handle guarded replicated language entities.

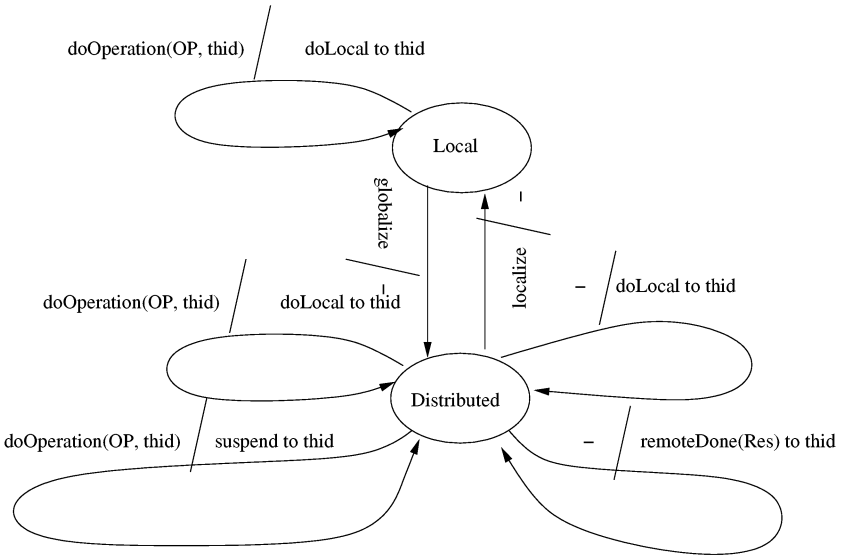


Figure 3.5: The state diagram for a guard.

operation called *globalization*. Globalization turns the guard into the state `Distributed`. A distributed language entity becomes a local entity by an operation called *localization* which turns the guard into the `Local` state.

Implementing Guards

To clarify the guard concept we will consider different implementation techniques used to realize guard constructs. We identify two requirements when realizing a guard, both related to efficiency:

1. The overhead imposed by the guard when performing an operation on a local language entity should be low.
2. The overhead of performing an operation locally on a distributed language entity should be low compared to performing an operation on a local entity.

If the first requirement is not met the overhead of distribution support will affect interaction with purely local language entities. Given that the non-distributed part of an application is typically larger than the distributed part, the effect of an inefficient guard implementation results in an inefficient programming system.

If the second requirement is not met it will be hard to realize efficient distributed applications. Different types of replication schemas typically supports consistent local access of language entities [55, 54]. If the overhead on operations on a distributed language entity is high, the benefit of local access diminishes. A coarse-grained model or a model where just a few dedicated language entities are distributable is not necessary affected, since fewer accesses to distributed language entities are performed and the guards are invoked seldom.

Locating distributed language entities at read/write protected virtual memory pages similarly to how shared memory systems detect access of shared memory solves requirement *1* [125]. However, the virtual memory page approach is expensive when accessing a distributed language entity since every access will result in a signal that halts the process. Moreover, unless virtual memory pages are of variable size it is hard to remove the guard when localizing a distributed language entity.

Another approach to implementing the guard is to explicitly annotate each language entity with a flag that indicates the distribution status of the entity. It has been shown that this is an efficient approach in programming systems which make use of tagged pointers, such as Mozart, Lisp, and Erlang [99, 32]. However, a requirement is that an extra tag can be added to the tagging scheme. For an alternative encoding it has been shown, in an extension to GnuJava [41], that adding an extra word to every object that is checked at method invocation incurs an overhead in the range of only tens of percents. A special type of flags is present in the .NET platform in the form of message sinks [103]. At creation, an object can be associated with an interceptor object, called a message sink. Any method call on an object with an interceptor is first directed to the interceptor which can choose to discard the operation or let the operation through to the object. Unfortunately, despite the elegance of the message sink model, experiments revealed that it imposes a high overhead [91], thus not fulfilling requirement *1* nor *2*.

3.2.2 The Abstract Entity

In order to provide distribution support for a programming system, the DSS should provide distribution support for the language entities found in the programming system. Most high-level programming languages offer a wide range of language entities. The union of all different language entities found in existing and future programming systems is very large. In addition, two language entities from two different programming languages might have the same name but from a programming point of view be semantically different. Objects, as found in different object oriented programming languages are examples of language entities that on a high level of abstraction describe the same type of entity, but on a more detailed level can differ considerably. For example, an object in Java and an object in C++ both have state and methods. However, the Java object is prepared for threads and can implement synchronized methods, something that the C++ object does not provide. However, from the distribution point of view, those differences can, to a large degree, be abstracted out and we are left with a smaller number of *abstract entity* types. Despite the differences between C++

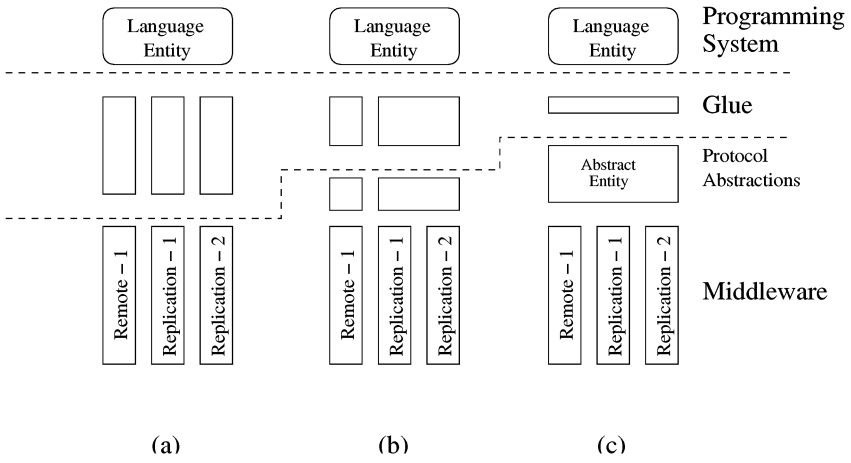


Figure 3.6: Alternative distribution strategy abstractions: (a) distribution strategies directly exposed, (b) distribution strategies classified according to distribution model (c) distribution strategies classified according to type of language entities supported.

and Java objects, the two objects can both be distributed using the same mechanisms, such as a remote execution type of protocol. The abstract entity model captures exactly this, not on the level of language level data structuring such as objects, but on the level of semantic behaviors.

To explain the abstract entity model, we give an overview of different possibilities of language entity distribution support. We use an example in which one language entity can be distributed equally well by three different distribution strategies, one remoting strategy (*remote-1*) and two different replication protocols (*replication-1* and *replication-2*). The actual choice of distribution strategy is decided in runtime. We assume that some mediating glue-code has to be written to couple the language entity to the middleware.

One possibility is for the middleware to expose the distribution strategies directly, as in Figure 3.6(a). In this case explicit code for each existing distribution strategy has to be written. In addition, the glue code is required to translate between operations on the language entity and explicit

protocol operations for each eligible distribution strategy.

An alternative is to present distribution strategies classified according to type of distribution model (see Section 1.2.1). The distribution strategies are organized according to distribution type, see Figure 3.6(b). The two replication protocols (*replication-1* and *replication-2*) can be exposed in one interface since they both provide read replica and write replica operations. Functionality is introduced in the middleware that translates from abstract protocol operations to proper protocol operations. The required glue code has been reduced since instead of supporting three different distribution strategies, two classes of distribution strategies are supported.

Continuing along this line of reasoning, we can raise the level of abstraction closer to the language entity. Given the observation that the three distribution strategies can be used to correctly distribute the language entity, there must be a common ground between the strategies. The abstract entity describes just this common ground, see Figure 3.6(c). One interface hides all eligible protocols for the language entity. Instead of writing multiple snippets of glue code, mediation between distribution support and language entity is done inside the middleware. Interaction with the abstract entity is done using operations that express the type of manipulation. The complexity associated with translating language operations into protocol operations is moved into the middleware, in the form of an abstract entity abstraction, resulting in a thinner glue layer.

The abstract entity model caters to extendibility. Consider adding a new distribution strategy to the scenarios in Figure 3.6. In case (a) a new piece of glue code has to be written. Unless the new distribution strategy belongs to the already supported classes of distribution support, the glue has to be extended in case (b) as well. However, in the case of an abstract entity, the extension is completely internal to the middleware.

3.2.3 Abstract Entity Interfaces

The abstract entity is the implementation of a guard. It is the responsibility of the programming system to direct operations to the abstract entity, and it is the abstract entitys responsibility to decide how an operation will be resolved. An abstract entity provides a number of operations that express

interaction of a shared data structure, called *abstract operations*. Similarly to how multiple language entities can be classified into the same kind of distribution support, multiple data structure operations can be classified into one *abstract operation*. For example, to efficiently distribute an object two types of distribution support is required: (i) for methods that alter the state of the object and (ii) for methods that access the state without altering it. This is expressed by the abstract entity interface by providing two types of abstract operations, one for altering data and one for just accessing data.

An abstract operation does not implement the actual operation of the language entity. Instead, an abstract operation is the means to tell the distribution strategy that a language level operation should be performed on the distributed language entity according to the guarded replication model described in Figure 3.4. The abstract operation type guides the distribution strategy on how the language level operation should be executed, for a more in detail discussion about abstract operations see attached paper *A* and *B*. In addition to abstract operations, the abstract entity interface defines a set of callbacks, necessary for executing operations remote, or transferring the state of a distributed language entity between different entity instances. The callbacks are defined and discussed in attached paper *A* and *B*.

Figure 3.7 depicts a state diagram describing the abstract entity. Note that in reality the abstract entity is a stateless interface. The decision of how an operation is to be executed is taken by the associated distribution strategy. In addition, the model is a simplification in that it can handle only one thread at a time and that the abstract entity has a notion of its state, `Skeleton` means that no operations can be performed on the local entity instance, while `Complete` means that operations can be performed. This information is in reality found in the distribution strategy associated with the abstract entity. An abstract operation performed on the abstract entity in a the complete state immediately returns `doLocal`. If the abstract entity is in the skeleton state, an abstract operation results in suspension of the calling thread. Depending on the distribution strategy the thread is either later told to do the operation locally, or handed the result of the operation. The `distOp` messages indicate state changes in the associated distribution strategy.

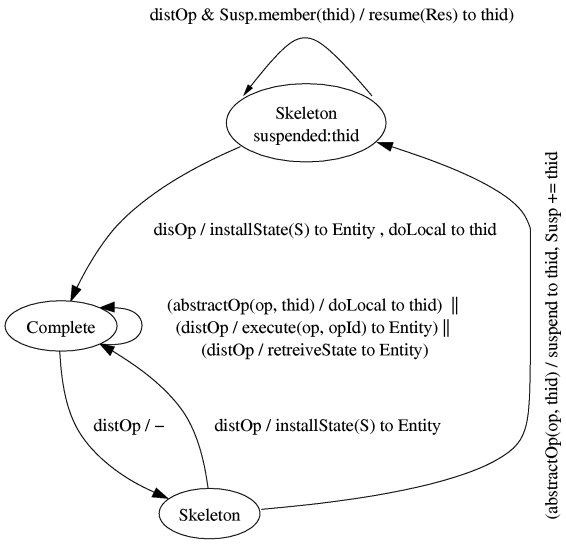


Figure 3.7: The state diagram for an abstract entity

Figure 3.8 depicts the state diagram for a language entity adapted to the abstract entity model. It has been extended with the three new operations, `executeOperation`, `installState`, and `retreiveState`. Sequence diagrams which describe the interaction between a thread and a distributed language entity, distributed using the abstract entity model can be found in both paper *A* and *B*. Diagrams are presented for both local access, state-passing local access, and remote access. In addition paper *A* presents pseudo code for an array mapped to an abstract entity. Attached paper *H* describes how language entities of Mozart are coupled to abstract entities. The implementation of abstract entities and the interfaces are described in attached paper *G*.

3.2.4 Abstract Threads

The DSS API supports an abstraction of threads, called *abstract threads*. An abstract thread serves two purposes. First, it is used to enable communication from an abstract entity with a programming system thread, that is

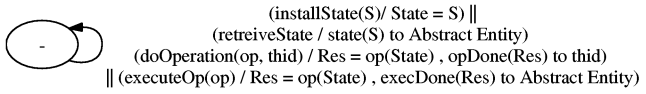


Figure 3.8: The state diagram for a language entity exteneded with interafces required by an abstract entity

resume a suspended thread when the operation has been executed remotely or when the operation can be executed locally. Second, the abstract thread gives a programming system thread a globally unique identity, necessary to implement location transparency for remote operations [62, 137]. Internally in the DSS, an abstract thread is represented by a global thread identity, called a *global thread* in attached paper *G*. For example, to preserve location transparency for an RPC the thread that executes the body of an RPC must have the same logical identity as the thread that called the RPC. To maintain the logical identity of a thread that performs remote operations it is necessary to properly implement constructs that are based on thread identities. A reentrant lock is an example of a construct that requires preservation of logical thread identities.

3.2.5 Different Types of Abstract Entities

A key observation for efficiency in distributed systems is that language entities that do not change their state can be replicated [93, 79, 64, 124]. Language entities that change their state can also be replicated, but to provide single-instance equivalence, the replicas have to be kept in a consistent state. This observation is central in the abstract entity model. Different interaction models are explicitly represented as abstract entity types. Each abstract entity type comes with a set of eligible distribution strategies. Moreover, the abstract entity types implement different abstract operation interfaces. Below follows the three abstract entity types currently supported by the DSS.

Mutable Abstract Entity This type of abstract entity has two abstract operations. *Update* indicates that the state is to be altered while

access means to read. The *mutable abstract entity* is preferably used by language entities that supports destructive updates, e.g. objects. Example of eligible distribution strategies for the mutable abstract entity are: remote-execution, mobile state [132], and read/write invalidation [81].

Immutable Abstract Entity Language entities that have static state, called *immutables*, can safely be distributed by replication. For efficiency reasons, immediate replication of the state of an immutable is not always the most optimal strategy [64]. The immutable abstract entity hides the replication strategy behind the *access* abstract operation. The DSS implements three different replication strategies for immutable data [112]. First, the immediate-replication distribution strategy which replicates a data structure every time a reference to it is received. Second, the eager-replication strategy which replicates only if no instance of the data structure exists at the receiving node. If an instance exists, no new instance is created and no description of the data structure is passed between the two nodes. Third, lazy-replication, which replicate the language entity first when the entity is read. Note that no coordination of the different replicas is required, since they do not change their state. Thus, after a replica has been constructed, no abstract entity is required. Examples of immutable data structures can be found in most programming languages, atoms in functional languages are one example, and strings in Java are another example.

Transient Abstract Entity The transient abstract entity describes the middle-ground between the *mutable* and *immutable* abstract entities. It starts in a mutable state, in which it can be updated by the *append* abstract operation. The mutable property can be terminated by the *bind* abstract operation. The name, transient, reflects the possibility to translate from mutable to immutable. One use of the transient abstract entity is to describe a stream to which items can be written. In addition the stream can be closed, and no more items can be written to it. The transient abstract entity is used to efficiently represent logic variables [59], futures [71, 83] and previous mentioned stream abstractions.

3.3 Distribution Strategy Framework

It is generally recognized that the choice of distribution strategy dominates the overhead for a distributed entity [76, 8, 111], see Section 1.2.1 and attached paper *B*. Here we try to illustrate the impact the choice of distribution strategy has on the overhead by a short example. Consider two processes A and B such that process B refers to an array initially located at A. If B accesses only one element of the array once, a remote execution protocol is the best choice. However, if B accesses the array multiple times, there is a point where it is more beneficial to replicate the array at B, despite how efficient marshaling or messaging techniques are used. Attached paper *B* discusses the importance of being able to choose distribution strategy for a distributed language entity motivated by presented results from evaluations of different distribution strategies.

The abstract entity interface of the DSS provides support for multiple distribution strategies. Within the functional bounds of an abstract entity there is a range of possibilities in choice of distribution strategy that becomes a tuning space; the best strategy depends on the application and the pattern of use.

The DSS is designed to simplify development and implementation of new distribution strategies. First, the abstract entity interface abstracts interaction between the programming system and the DSS. Thus, a distribution strategy does not communicate directly with a language entity, but with DSS defined interface routines. Second, internally, the DSS hosts a framework that divides distribution strategy functionality into *sub-protocols*. This division is based on the observation that distribution strategies often share the same functional components, such as a distributed garbage collector and the notion of a home. This has in our model been explicitly modeled as sub-protocols.

The sub-protocol framework captures the right level of abstraction, indicated by the large sub-protocol suite implemented in the DSS, see attached paper *B*. For example, implementation of the *pilgrim protocol* [56] in the DSS required a minimal effort, primarily because of the separation of concerns in the sub-protocol framework. By implementing the pilgrim protocol in the framework, functionality such as mobile home and distributed

garbage collection where automatically added to the protocol. These issues where not addressed in the original description of the protocol. Development of a new sub-protocol does not create just one new distribution strategy, but a large number of new strategies (all combination of existing sub-protocols combined with the new sub-protocol).

3.3.1 The Coordination Network

The nodes of a distributed application form virtual networks on top of the underlying network, caused by the distributed entities referred from the nodes. Such a virtual network is called a *coordination network*. A coordination network consisting of four nodes is depicted in Figure 3.9. The network includes the *reference set*, defined as every process that holds a reference to a language entity (nodes A, B, and C in Figure 3.9) and the *home* of the distributed language entity. At the home of a coordination network the distributed language entity is represented by a *coordinator*. In Figure 3.9, the coordinator is located at node D. Recall that the notion of a generic home for distribution strategies was introduced in Section 1.2.1.

The coordinator is a special entity in a coordination network that implements an arbitrating functionality for the distribution strategy. Examples are keeping information about the current location of a mobile object, or maintaining the set of read copies of a read/write invalidation distribution strategy. The coordination network maintains the invariant that every reference holder, called a proxy, can communication with the coordinator.

The coordination network is dynamic in its configuration. As references to a distributed language entity are passed between processes, new processes join the coordination network, and the reference set grows. Local loss of interest in a distributed language entity causes nodes to leave the coordination network, and the reference set shrinks. The size of the reference set is used to define the distribution status of a language entity. A language entity that has a reference set of size one is a local entity and reference set of size two or larger makes a language entity distributed.

A language entity starts as local, it is referred from one process only, and the reference set is of size one. By passing a reference to a remote node, the language entity becomes distributed. At this point the reference set is

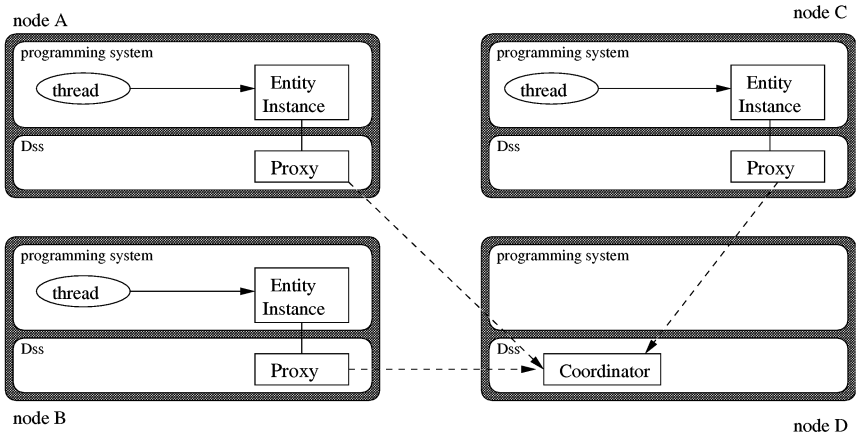


Figure 3.9: A coordination network that spans 4 nodes (A-D). Node D is member of the coordination network, even though it does not hold a proxy. This could for example be a remote object that is only referred from remote nodes, and not referred locally. The reference set consist of nodes A to C since no reference exists from node D.

of size two and the coordination set of size three, including the coordinator. The coordinator is created when a reference to a local entity is passed over the network, the entity is globalized. Later, if the size of reference set reaches one the coordinator can be removed, and the language entity is localized. .

Maintaining a correct view of the reference set is hard in the face of network and node failures. If a node that holds a reference is terminated without being removed from the reference set, the coordination network will never be dismantled. To handle this, the DSS makes use of time-lease based distributed garbage collection algorithms that can handle network failures and node failures [26, 53, 121].

3.3.2 Sub-protocols

The DSS implements a framework for distribution strategy development based on division of functionality into three sub-domains; each sub-domain is realized by a sub-protocol type. Each sub-protocol type implements a well defined service and together the three sub-protocol types provide the services necessary for a distribution strategy. Interfaces between different sub-protocol types make it possible to freely combine instances of the three sub-protocol types to implement dedicated distribution strategies. The three sub-protocol types are:

Consistency sub-protocol The functional-property of the distribution strategy, access to a distributed language entity under single-instance-equivalence is implemented by this sub-protocol. It is the consistency sub-protocol that interacts with the abstract entity and the programming system level language entity.

Coordination sub-protocol The sub-protocol implements two services, a location service of the coordinator, and migration primitives. The location service provides location transparent message passing from a proxy to the coordinator. The migration primitives allow relocation of a coordination network's coordinator from outside the DSS.

Reference sub-protocol Implements the distributed garbage collector that ensures that the coordination network is dismantled when the

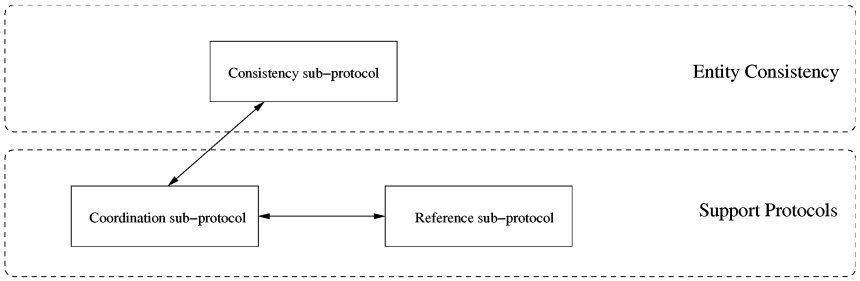


Figure 3.10: The internal organization of the different sub-protocol types. The coordination and the reference sub-protocols provide a support service for the consistency sub-protocol.

size of the reference set reaches one.

The three sub-protocol types together implement a distribution strategy. Figure 3.10 shows the organization of the different sub-protocols. The consistency sub-protocol provides the entity consistency services to the abstract entity. The coordination and the reference sub-protocols implement support for the consistency sub-protocol. The different sub-protocol types are described in attached paper *A* and *B*. Attached paper *G* describes the implementation of the coordination framework, and shows the interfaces of the different sub-protcol types.

Each sub-protocol is defined as two units, a *home-unit* and a *remote-unit*. The home-units of the three sub-protocols implements the coordinator and the remote-units the proxy. Interaction between a remote- and home-unit of a particular sub-protocol is defined by the sub-protocol implementation. Interaction between home- or remote-units of different sub-protocol types are defined by C++ interfaces, presented in attached paper *G*.

3.3.3 Implemented Sub-protocols

The DSS provides multiple consistency strategies, of both remoting and replication type. The set of available reference sub-protocols includes most reference counting type of distributed garbage collection algorithms [77,

14, 114, 100]. In addition, time lease based algorithms handling loss of proxies because of network and/or node failures are also implemented [53]. The DSS implements a novel framework for combining different types of distributed garbage collection algorithms [78] which is used to combine the Fractional Weighted Reference Counting algorithm presented in attached paper *F* with a time-lease algorithm. One stationary and two mobile coordinator sub-protocols have been implemented. One of the mobile coordinator protocols uses a classical forward pointer protocol to locate a migrating coordinator [114]. A more interesting approach, also implemented in the DSS and presented in attached paper *E*, is to make use of a distributed directory service implemented by a structured peer-to-peer overlay network to keep track of the current location of the coordinator.

Two of the 14 implemented protocols are presented in attached paper *E* and *F*. Here the different protocols are introduced together with pointers to where the protocols are described. The following consistency sub-protocols are implemented in the DSS.

Mobile state The protocol uses a token which is passed between the proxies in the coordination network. The proxy holding the token has sole access to the state of the distributed language entity. Migration of the token is controlled by the home-unit located at the coordinator. The base protocol as described in [132], was refined for better fault-detection.

Pilgrim A mobile state protocol inspired by the work in [56]. The proxies of the coordination network which needs to read or write to state of the distributed language entity form a ring. The token is constantly passed from proxy to proxy in one direction over the ring. A proxy gets sole access to the state of the distributed entity when the token is received and is supposed to pass the token to the next member of the ring when any operations on the state are completed. The role of the home-unit is to maintain the ring so that a proxy can find a member of the ring to connect to. The protocol outperforms the mobile state protocol in the special case when a small set of proxies reads and writes frequently to the state.

Read/write-invalidation The protocol maintains two types of tokens,

multiple read tokens and a single write token. The write token gives the proxy holding it sole access to the state of the distributed entity. Holding the read token allows a proxy to read the state of the distributed entity. The home-unit ensures that there is either one write token or a set of read tokens. When switching between token types, the existing tokens are said to be invalidated. The protocol is inspired by protocols presented in [81].

Remote execution The protocol is an implementation of the traditional RPC protocol. A proxy sends operations to the home unit which returns the result to the proxy.

Once only The protocol implements a one-shot broadcast service over the coordination network. Any of the proxies can initiate a broadcast that will reach every proxy, including the initiator. After one broadcast has been sent, no more broadcasts can be sent. This protocol is a generalization of the distributed unification protocol used for distributed logic variable in Mozart [59].

Replication The protocol implements lazy replication. When a proxy is created, it acts as a placeholder. The proxy can perform one operation, retrieve a state description of the associated distributed language entity. Description and evaluation of the implementation can be found in [112]. The protocol is similar to the replication protocols presented in [64].

The following reference sub-protocols are implemented in the DSS. Note that different reference sub-protocols can be combined to form more advanced distributed garbage collection algorithms [78].

Fractional Weighted Reference Counting An efficient reference counting protocol allowing references to be sent between nodes without the need for any third party communication. The protocol is described in attached paper *F*.

Time Lease A variant of the classic time-lease algorithm which can handle failure of proxies of the coordination network, and still detect when the reference-set becomes empty.

Reference Listing The home-unit maintains an explicit list of the proxies in the reference set [114]. When a reference is passed to a node, the node must be inserted into the set of existing proxies. Moreover, as proxies leave the reference set, the home-unit has to be informed.

Reference Counting A simple distributed reference counting algorithm which is based on the regular reference counting technique, adapted to the distributed environment [82]. The algorithm has a small memory footprint, but requires messages to be sent to the coordinator when new references are passed between nodes.

The coordination sub-protocol provides location-transparent access from proxies to the coordinator. A discussion about different techniques to maintain location-transparent access to a mobile coordinator is presented together with an evaluation of different techniques in attached paper *E*. The following coordination sub-protocols are implemented in the DSS:

Stationary The location of the coordinator is fixed to the node where the coordinator is created and each proxy statically know the location of their coordinator.

Forward Chaining The coordinator can be migrated to nodes holding a proxy. When migrating, a pointer is left at the node migrated from, pointing to the new location. Each proxy keeps a pointer to the last known coordinator location. Messages to the coordinator are sent to the last known node. If the coordinator has migrated, a message sent to the coordinator is routed over the pointer(s) pointing to more recent locations. The protocol is inspired by the forward chaining algorithm presented in [114].

DKS Directory The protocol is based on the directory-service approach (see attached paper *E*), realized by the structured peer-to-peer system DKS [2]. The DKS system is used to store the current location of a coordinator, updated at migration. Each proxy maintains a last known coordinator location which is used when communicating with the coordinator. If the coordinator has migrated, the DKS system

is queried for a more recent location. The algorithm is presented in attached paper *E*.

To emphasize the expressiveness of the model, consider the many possible distribution strategies which can be created using the above presented protocols. By combining the instances of the three different sub-protocol types we can get `6 * 5 * 3 = 90` different distribution strategies, this by just implementing 14 sub-protocols. In this calculation the possibility to combine reference sub-protocols is not taken into account. Comparing the work necessary for implementing 90 distribution strategies versus 14 sub-protocols should be a strong argument for the coordination framework of the DSS.

3.3.4 Examples of Consistency Sub-protocols

The interaction of a thread and a distributed language entity is shown in the form of sequence diagrams in attached papers *A* and *B*. The interaction is described on the level of implementation in attached paper *G*. In order to complement those descriptions, we present the state diagrams for two consistency sub-protocols, read/write invalidation and remote execution. Each protocol is described as two state machines, one for the remote unit that executes at a proxy and one as a home-unit executing at the coordinator. In order to simplify the descriptions, we have deliberately left out failure handling from the protocol descriptions.

The descriptions are interesting in that they depict the interaction of the threads and the entity instance at the programming system level. No information about internal structuring of neither language entities nor threads exist at the level of the consistency sub-protocol. The two protocols represent different classes of distribution strategies, the single-instance operation-passing and the multiple-instances state-passing type and show two ways to interact with the programming system from the DSS.

Read/Write Invalidation

The protocol maintains two types of tokens. One write token and a set of read tokens. There can at most be one write token in a coordination

network or as many read tokens as there are proxies. However, the write token can not exist simultaneously with read tokens. The two types of tokens provide different capabilities. Holding a read token allows for local reads of a distributed entity's state. Holding the write token allows for both read and writes operations on the local instance of a distributed language entity.

The home-instance of the sub-protocol is responsible for maintaining the invariant that there is at most one type of tokens in the coordination network. Switching from one type of token to the other type is called invalidated, since all read tokens are invalidating when going from read to write, similarly, the write token is invalidated when going from write to read. The state-transition diagram for the home unit is depicted in Figure 3.11. The home-unit maintains four values:

W pointer to the node holding the write token.

R a list of nodes holding read tokens.

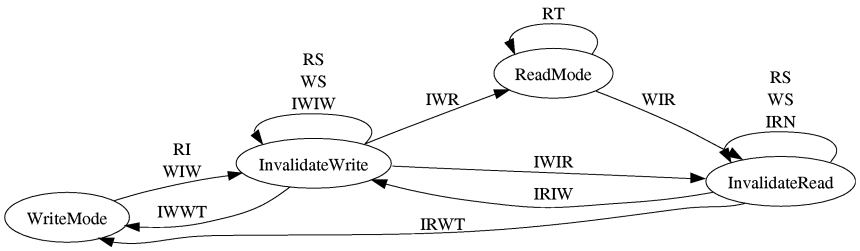
Q a queue of requests from proxies in the coordination network for either a read or write token.

S a pointer to the current state of the distributed language entity.

The remote-unit is responsible for ensuring consistent read and write access to the state of a local language entity instance. Threads interact with the remote instance by `read` and `write` operations, modeled as messages. The remote-instance interacts with the programming system by resuming threads, installing entity state descriptions, and retrieving entity state descriptions. Figure 3.12 depicts the state diagram of the remote-instance. The remote-unit maintains two values:

S a queue of threads suspended when trying to do either read or write operations.

O a list of ongoing operations conducted by threads at the programming system level.



WIR = write(N) / invalid to all in R, Q+=N
WIW = write(N)/Q+=N, invalid to W
WS = write(N) / Q+=N

RI = read(N)/Q+=N, invalid to W
RS = read(N) / Q+=N
RT = read(N)/ readToken(S) to N, R+=N

IWIW = invalidWrite(s) & Q.len < 1 & Q.1 isWrite / writeToken(s) to Q.1, invalid to Q.1, Q−=Q.1
IWWT = invalidWrite(s) & isWrite(Q.1) & Q.len ==1 / writeToken(s) to Q.1, W = Q.1, Q = nil
IWIR = invalidWrite(s) & containsWrite(Q) & Q.1 isRead / S = s, readToken(s) to all in untilFirstRead(Q),
Q−= untilFirstRead(Q)
IWR = invalidWrite(s) & containsNoWrite(Q) / S = s, readToken(s) to all in Q, Q=nil

IRN = invalidRead(N) & length(R) > 1 / R−=N
IRIW = invalidRead(N) & length(R) == 1 & length(Q) > 1 / writeToken(S) to Q.1, invalid to Q.1 Q−= Q.1
IRWT = invalidRead(N) & length(R) == 1 & length(Q) == 1 / writeToken(S) to Q.1 Q−=Q.1

Figure 3.11: State diagram for the home-unit of the read/write invalidation consistency sub-protocol.

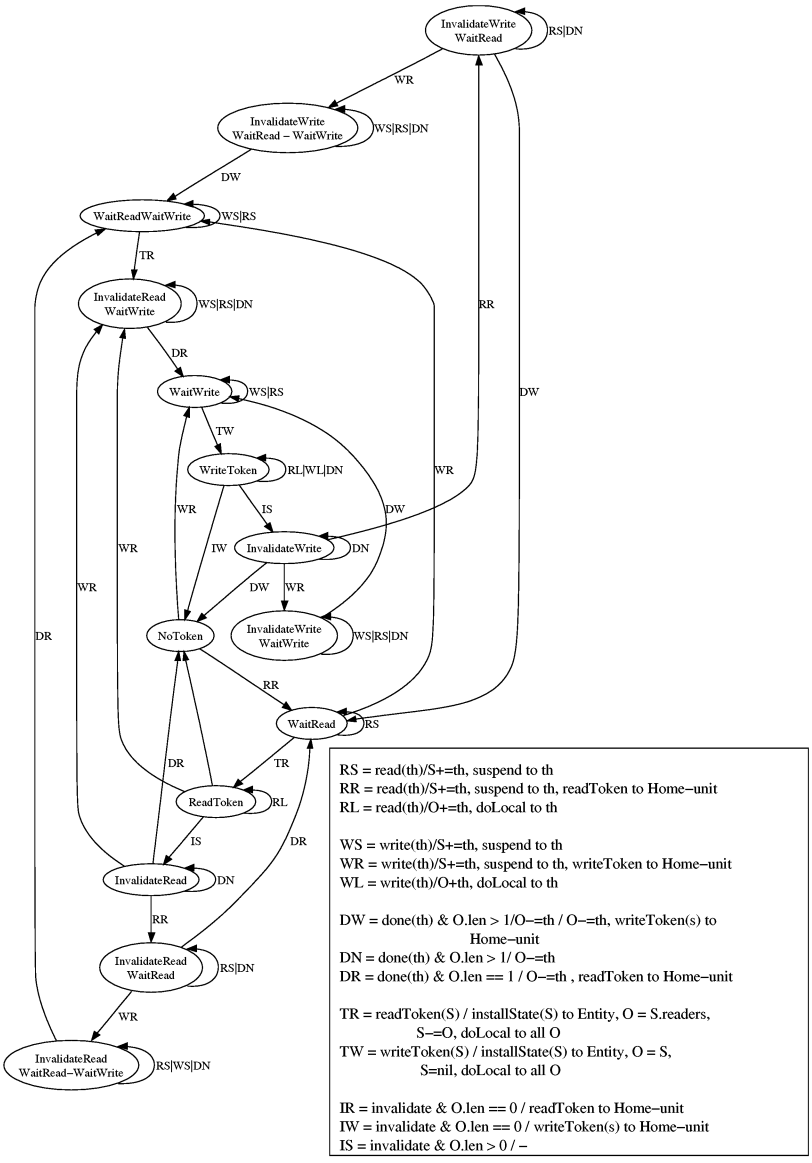


Figure 3.12: State diagram for the remote-unit of the read/write invalidation consistency sub-protocol.

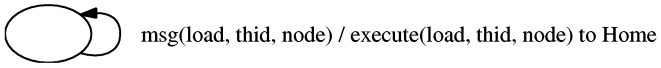


Figure 3.13: The state diagram for the home-unit of the remote execution consistency sub-protocol.

Remote Execution Protocol

The remote execution sub-protocol implements a generic single-instance operating-passing protocol. The protocol can be used to realize both RPC and RMI. The protocol is simple compared to the Read/Write invalidation protocol, and shows the strength of the coordination framework model. The single-instance of the language entity is collocated with one dedicated remote-unit. That remote-unit has the status *complete*. Every other proxy has the status *incomplete*. Operations performed on an entity instance associated with a proxy whose remote-instance is in the incomplete status are transported to the proxy with complete status and executed there. Messages are sent from proxies in the incomplete status to the home-unit of the consistency sub-protocol.

The home-unit of the remote-execution consistency sub-protocol acts as a relay unit. It has a pointer to the node where the complete proxy can be found, usually at the same node as the coordinator. The home-unit accepts one type of message `forward`, which is sent to the complete proxy. The state-transition diagram for the home unit is depicted in Figure 3.13. The home-unit maintains one value:

C a pointer to the node where the proxy with status complete can be found.

The status of a remote unit, complete or incomplete, is defined when it is created. Interaction with the remote-unit is done by the `write` operation. If the unit is in complete status, the operation is executed locally, otherwise remotely. The state-transition diagram for the remote-unit is depicted in Figure 3.14. The remote-unit maintains one value, only used by the instance in a complete state:

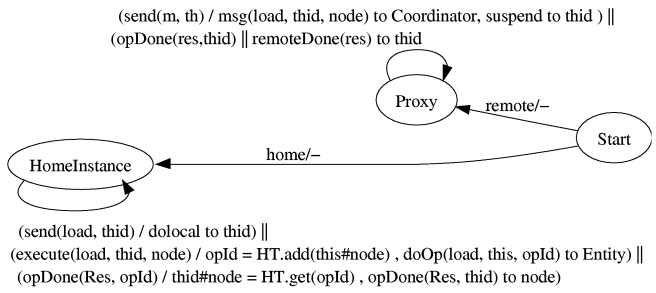


Figure 3.14: The state diagram for the remote-unit of the remote execution consistency sub-protocol.

HT a hash table maping operation identities to pairs of a thread and a node.

3.3.5 Referentially Secure Coordination Networks

In order to maintain referential security each coordination network is assigned a unique non-forgeable identity. A reference to a coordination network, required to be a member of a coordination network, can only be obtained by delegation from a process that is itself a member of the coordination network. Intra-coordination network messages are authenticated using the identity of the coordination network.

3.4 Messaging Layer

The messaging layer of the DSS provides a channel based communication mechanism for the coordination layer based on first class node references. The network abstraction hides issues regarding reliable delivery, serialization, failure detection, authentication, and encryption. Bidirectional channels are, as needed, automatically established to other nodes. Message passing is asynchronous, reliable and FIFO with respect to a bidirectional channel. To further simplify networking the messaging layer detects, classifies and reports failures on the first class node references.

Networking is inherently dependent on the applications deployment environment. First, communication primitives are usually operating system specific or virtual machine specific. To simplify the integration into programming systems, the messaging layer supports custom implementations of the low-end communication functionality. Second, connection establishment and failure detection is dependant on the network environment of an application. Consider the difference between machines located on a LAN and machines spread over the Internet. In the first case there are no firewalls, latencies are short, and node failures can be detected at the network level. The Internet case includes firewalls, processes that change their addresses, potentially very high latencies, and no reliable method to detect node failures. To cope with a multitude of network environments, connection maintenance (establishment and failure detection) is located in a second customizable module. This software design makes the messaging layer deployable in different environments.

3.4.1 First-Class Node Reference Model

Communication in the DSS is based on first class representations of nodes, in the form of node references. A node reference is used both as a channel for sending messages to the physical node it represents and as an identity. At any point in time a node holds a number of node references to other nodes, called the *known set*. During the lifetime of a distributed application, references to nodes are passed between nodes, thus the known set of a node potentially changes. At any one time a node needs to communicate with a subset of the nodes referred from the known set, this subset is also subject to change. It is possible that a node will never communicate with a subset of the nodes in the known set. A mark and sweep garbage collector is used to detect when node references are no longer needed as identity tokens or as communication channels.

Apart from being a channel to the node, a node reference also maintains information regarding the accessibility of the node. The nodes referred from the *known-set* are constantly monitored, and their accessibility is abstractly represented by a three state model. A state change of a node reference is automatically reported to higher layers where action can be taken. The

three states are:

No-problem. The node can be reached.

Communication-problem. The node is not accessible. However, it is possible, but not certain, that the node will be accessible again.

Crash-failure. The DSS-node has crashed and will never again be reachable from any DSS-node in the network.

Note that *communication-problem* is local to one node while *crash-failure* is global to all nodes of a distributed system. The local property of *communication-problem* makes it possible that parts of the nodes of a distributed system can communicate with a node that other parts of the same distributed system finds to be in a *communication-problem* state, a network partitioning has taken place. The global and monotonic property of *crash-failure* makes it possible to assume that no node can communicate with a node that is in the *crash-failure* state. It is possible that some of the nodes of a distributed system classifies a given node as being in the *communication-problem* state while others classifies the node as being in the *crash-failure* state.

Note that the use of the *crash-failure* state requires a failure detector that can correctly detect this state. This has been shown to be impossible in a system of processes communicating asynchronously [30]. However, there exists circumstances where it can be correctly detected, e.g. a node can potentially detect that another node located at the same machine has terminated, and sometimes even correctly detect that another node on the same LAN has terminated. Consequently, unless a reliable failure detector is provided, *communication-problem* will be the prevailing failure state associated with nodes that has crashed/terminated. As described in Section 3.4.3 the messaging layer is designed such that custom failure detectors easily can easily be added.

3.4.2 The DSite Interface

Node references are represented by DSite objects that act as proxies for the processes they represent. The DSite implements a seamless communication

interface and can be passed by reference between processes. Complicated tasks such as establishment, monitoring and termination of connections are hidden behind the asynchronous messaging interface provided by a DSite. A DSite provides one interfaces for passing messages to the process it represents and one interface for inquiring the status of the process it represents.

The messaging layer is a standalone middleware that is primary designed to provide a messaging service for the DSS, but is general enough to be used as a standalone messaging component. The sub-protocols of the coordination network use the DSite objects for communication and for identification. It is the sub-protocols that define the *known set*.

3.4.3 Internals of the Messaging Layer

The abstract node references in the form of DSite objects greatly simplify sub-protocol implementation in the coordination layer. The separation of messaging from failure detection/reporting caters for separation of concerns. However, maintaining the functionality of the DSite requires considerable support from the messaging layer. The messaging layer is the, in lines of code, single largest sub-component of the DSS. We will here highlight some of the features of the messaging layer.

Each DSS process is identified by a globally unique identity maintained by the messaging layer. The identity of a process is separated from its address; this is an important requirement [116] for supporting mobile processes. The identity of a node is static and guaranteed to be unique. Instead of representing addresses according to a given standard (e.g. IP number and port) an address is represented as a function of how to establish a connection to the process. Thus, a DSite that represents a process consists of an identity and a function that establishes a connection. The address function is subject to customization and can be changed during the lifetime of a process.

The separation of address from identity is explicitly represented in the design. The tasks of I/O handling and connection maintenance are located in separate modules. The *connection module* is responsible for establishing connections and detecting problems on established connections. Interfacing towards the operating system is delegated to the *communication module*.

Both modules are represented by well defined interfaces to facilitate custom implementation. The design of the messaging layer is described in attached paper *C*.

Inter-DSS process communication is made secure by non-forgeable DSite instances together with encrypted channels. To establish contact with a DSS-node, a correct DSite instance is required. Making DSite references non-forgeable prevents processes from connecting to a DSS-node without holding a correct reference to that process, i.e. knowing or guessing the physical address is not enough. Furthermore, a correct reference to a given DSS-node can only be created by a DSS-node itself. Other nodes learn of a DSS-node (in the form of a DSite) by reception of DSite references. Encryption of the communication channel prevents eavesdropping on communication between DSS nodes, and allows safe distribution of non-forgeable node-references in the form of DSites. Attached paper *D* further describes security aspects of the DSites.

Chapter 4

The Programmer's view of the Distribution SubSystem

This chapter describes the DSS from the point of view of a user of a distributed programming system developer. Section 4.1 describes failure handling in a programming system integrated with the DSS. Section 4.2 describes the standalone and decentralized properties of the DSS middleware. Finally, an overview of the evaluation of the DSS is presented in Section 4.3. Attached paper *H* presents the integration of the programming system Mozart with the DSS on the level of C++.

4.1 Practical handling of Failures

This section presents some practical aspects of handling failures in a distributed programming system integrated with the DSS. The DSS is designed to provide location, access, and replication transparency, but not to provide failure transparency. Failures affecting the functionality of the DSS are exposed to the programming system level, thus the decision of how to handle a given failure is delegated to the programming system.

4.1.1 Failed Coordination Networks

Section 1.2.3 and 1.2.3 presents the failure model implemented by the DSS. Failure of a node in the coordination network can fail the consistency sub-protocol which in turn prevents further interaction with a distributed language entity. Similarly to how nodes in a distributed system can fail, distributed language entities can fail.

The DSS detects node failures, and classifies the impact of a node failure on the existing proxies. The failure model used to describe DSites is used to describe the status of a coordination network:

No-problem. The coordination network functions correctly.

Communication-problem. The coordination network does not function correctly. It may be the case that the coordination network later will function correctly.

Crash-failure. Nodes necessary for the functionality of the coordination network has crashed, the network will never be able to provide service.

The DSS reports changes in failure status of the coordination networks the existing proxies belongs to. An example of how to handle fault callbacks based on a notion of *watchers* [110] is presented in attached paper *H*. A watcher consists of a condition and an action and is associated with a distributed entity. The condition describes on which types of fault callbacks the action should be triggered on. The action is represented as a programming fragment, typically a procedure. At a fault callback that matches the condition en a fault callback the action is executed and the watcher is removed. The model is inspired by the fault model of Mozart [110].

4.1.2 Time Lease and Partitioning

Reference based distributed garbage collection algorithms are vulnerable to failures. Failure of a node holding a reference or loss of a message containing a reference prevents future reclamation of the data the algorithm manages. Time lease is a simple algorithm which overcomes these problems. The algorithm is based on a notion of leases which has to be periodically

renewed. Each reference, both at a node an in a message, is a lease. For a lease to be renewed, a message has to be sent to the home-unit for a lease renewal request. A lost reference will not issue a lease renewal, and thus not prevent reclamation of the managed data.

Link failures can temporarily prevent a reference to renew its lease. If the link failure disappears and the managed data has been removed during the link failure, the reference becomes a dangling reference. The DSS is able to detect dangling references. A dangling reference is reported to the programming system as a special type of crash-failure.

4.2 Decentralized Distribution Support

A design decision of the DSS is that a distributed application should be self contained. This means that the services required to maintain the abstract entity distribution model should be present on the DSS nodes that make a distributed application. Thus, the DSS does not require the existence of any external services such as directory services or class repositories as in Java, Globe [7], and CORBA.

We believe that this independence on external services simplifies development and deployment of small to medium scale applications. Moreover, if large applications are developed, the DSS does not prevent the usage of external services. This section describes three design choices that make the DSS provide a decentralized service.

4.2.1 Bootstrapping a Distributed Application

Distribution models based on distributed language entities can use distributed language entities to introduce new distributes entities. For example, a method of a remote object can return a reference to a new remote object, thus initiate a new contact between callers and callees. This model of language entity dissemination requires an initial distributed entity. Naturally it cannot be used to connect two nodes that does not share any distributed entities. This problem of how to connect processes that do not share any channel of communication and is commonly called the bootstrapping problem. In systems like Java and CORBA, dedicated directory

services located at known and fixed addresses are used. In the DSS we have chosen a solution that does not require any dedicated services.

A reference to a distributed language entity consists of a description of how to construct the language entity instance and a reference to the coordination network of the distributed language entity. Such distributed language entity references are automatically passed between processes as a result of the operations on distributed language entities. In addition, the DSS supports writing distribute language entity references to secondary storage. A distributed language entity references can be written to disk and passed between processes. This mechanism is expressive enough to implement both directory services, and capability based ticket mechanisms such as in Mozart.

4.2.2 Establishing Connections

Asymmetric connectivity caused by fire-walls and/or network address translators (NAT), is a hindrance to connection establishment on the Internet. Connectivity is asymmetric when a connection can be established in one direction but not in the other. It is an administrative problem that has yet not been properly addressed.

Machine migration, be it physical or conceptual, can force a node to change its address. That is, the machine changes its address and the node must change its address as well, at least if we assume the de facto protocol of Internet, TCP. Until the information of the new address of a node has propagated in the distributed system, a node that changes address manifests itself as asymmetric connectivity. The nodes of the distributed system cannot establish connections to the node that has changed address, but the migrated node can connect to the nodes of the system.

Traditional solutions to the asymmetric connectivity problem includes rendezvous servers [98] and administratively opening firewalls for dedicated traffic. However, we propose using paths in the overlay network that is automatically created by the coordination networks of a distributed application. The network consists of established connections created to provide the connectivity requirements of the coordination network. The overlay network can sometimes show higher connectivity than the underlying net-

work. Even if it is impossible to establish a connection from node A to node B, there might exists a connection from node C to both A and B that if discovered could be used. In attached paper C we describe how a Gnutella [107] like flooding algorithm can be used to find paths of existing connections between two DSS nodes. This is an example of the strength of separating process identity from address. The flooding algorithm can be used to find indirect paths between nodes as well as discover the proper address of processes whose machines have changed their addresses.

4.2.3 Finding a Relocated Coordinator

A coordination sub-protocol which supports migration of the coordinator can be used to relocate a suboptimal located coordinator. A suboptimal located coordinator is known to render pathological performance for some replication protocols [36, 68, 94]. Moreover, a coordinator can be relocated when its current process is to be taken down. Disregarding the reason to migrate a coordinator, it is a requirement that the proxies of the coordination network are able to discover the new location of the coordinator.

Different techniques exist for locating migrating entities. The use of a home for keeping the current location of a coordinator is ruled out because of the dependency on the home. Forward pointing techniques is simple to implement, old coordinator locations points at more recent locations [114]. However, the pointers from old locations must all be present for old references to the mobile entity to be valid, thus causing strong dependencies on availability of old coordinator locations. Another approach is to make use of external services such as directory services or distributed data bases [7] to keep information about the current location of the coordinator. This approach, even though shown to be efficient and reliable, requires a dedicated, external infrastructure not really applicable to small and medium sized applications.

Instead, we make use of a structured peer-to-peer overlay system [3] that is inherently fault tolerant, self organizing and scalable. The structured peer-to-peer system DKS [2] is used to distribute the contents of a directory service over the nodes of a distributed application. Thus, instead of relying on an external service, the nodes themselves become the

service. Information regarding the current location of a migrating coordinator is stored in the directory service. The solution is further described in attached paper *E*.

4.3 Validating the Approach

To validate the applicability of the design, the DSS has been coupled to the multi-paradigm programming system Mozart [90]. The DSS has also been used to implement distribution support in the form of a C++ library, according to the new entities approach. The library was primary developed to explore how the abstract entity model confined with POSIX threads [69].

Integrating a programming system with the DSS was done to evaluate the generality and completeness of the middleware design. Moreover, to validate the efficiency of the design (and its implementation); the two DSS based system are compared against other distributed programming systems. Since most programming systems of today offer some means to achieve distribution, it is possible to measure the cost of incorporation of the DSS to a programming system.

4.3.1 Integrating the DSS with a Programming System

The multi-paradigm distributed programming system Mozart is an interesting subject for integration with the DSS. If the multi-paradigm programming model can be supported, programming systems which adheres to just a subset of the paradigms supported by Mozart could make use of the DSS. Moreover, since Mozart already implements a well maintained distribution support library, the DSS extended version of Mozart can be compared to the original Mozart to evaluate the overhead of the DSS approach. In the evaluation we looked at how well the programming model could be distributed and if the resulting system was reasonable efficient.

The Mozart version coupled to the DSS is called OzDSS (this relates to the programming language Oz that Mozart implements). Implementation of OzDSS first required removal of the native distribution support implemented by Mozart. The language entities and threads of Mozart had to be extended with interfaces which allowed for interaction with the DSS. Sur-

prisingly, the resulting implementation has fewer interaction points with the DSS than the original native distribution support had with the programming system. The language entities where associated with abstract entities that correctly distributed the semantics of the language entities. The abstract entity model was expressive enough to handle the data structures found in the language Oz. A new concept was introduced in OzDSS that was not present in Mozart, annotation of distribution strategy. The clean interface between the programming system and the DSS made it possible to enable distribution for some language entities that where not supported in Mozart. The integration is described in detail in paper *H*. The coupling was done by two persons, both had expertise in the DSS, but only one had experiences with the internals of Mozart, and the first prototype was up and running within a couple of months.

The Mozart system is implemented by a virtual machine which provides services such as light weight threads and garbage collection. Moreover, programming language level entities in the virtual machine are explicitly represented as C++ objects which have introspective properties. In order to broaden the evaluation and see how the DSS can work as a distribution support library the C++DSS prototype was developed. C++DSS implements a data structure library on top of the DSS which supports distributed programming in C++. Moreover, C++DSS makes use of OS-level POSIX threads.

4.3.2 Evaluation

Evaluation of the DSS is simultaneously an evaluation of the implementation and an evaluation of the presented approach to distribution support. By showing that the implementation is efficient, we indicate that the approach can be implemented efficiently, and thus is viable. To evaluate both design and implementation we have conducted three different evaluations. First, the performance of the DSS, how fast the middleware can send messages is measured. Second, the impact on the performance of a distributed application when the distribution strategy can be changed. Third, how much overhead does the DSS library impose on a programming system compared with a native closely implemented solution.

The basic messaging mechanism of the DSS, excluding the encryption mechanism, imposes an overhead of 26% compared to raw sockets. This is a small overhead when considering the difference in the abstraction provided by a DSite and a socket. The benchmarks are described in detail in attached paper *B*.

Remote invocation has been used when testing the performance of different distributed programming system. For object oriented system remote invocation is equal to a remote method invocation. The overhead of the DSS was measured by executing an Oz program that performed remote execution on Mozart and OzDSS. The overhead was surprisingly small, only 10 % compared to the performance of Mozart. This figure must be seen in the light of OzDSS, two components coupled by a glue layer, and Mozart, one optimized distributed programming language. Comparing the performance of OzDSS with other distributed programming systems reveals that the library is efficient. Different versions of Java are more than 3.5 times slower, and .Net remoting is almost 6 times slower. See attached paper *H* for a more elaborate description of the benchmark.

To evaluate the benefit of being able to choose distribution strategy for distributed language entities, a concurrent distributed Oz application was used. A set of threads, physically distributed over a set of processes, concurrently manipulates a set of mutable language entities. The application was used to evaluate distribution strategies for different configurations of number of nodes and number of threads per process. The test shows that the difference in performance between a distribution strategy that "matches" the usage case and one that does not is in the order of magnitudes. The benchmarks are described in detail in attached paper *B*.

4.3.3 Summary

The evaluation of the DSS middleware indicated that the design is sound and that it is possible to implement the design efficiently. Moreover, coupling the middleware to a programming system, either according to the integrated approach or new entities approach, results in an efficient and highly customizable distributed programming systems. The resulting programming systems showed better performance than any of the well known

systems such as Java and .NET.

The middleware was successfully integrated with the Mozart programming system and did not require expertise in programming system internals. The coupling was finished within months, compared to the development of the distribution support for Mozart that was an ongoing activity from 1996 to 1999 involving up to 5 persons working simultaneously. The abstract interface provided by the DSS was generic enough to capture the semantics of the language entities of the multi-paradigm programming language Oz. Thus, the DSS can support the functional, the declarative-concurrent and the object oriented paradigms.

Finally, the choice of distribution strategy was shown to be the dominant factor when optimizing a distributed application. This indicates that our approach to allow choosing distribution strategy on single entity level is crucial for high performance distributed computing.

Chapter 5

Summary of the Papers

This chapter introduces the papers included in this dissertation. Each paper is summarized and its relation to the realization of generic distribution support for programming systems is explained.

5.1 Paper A: The DSS, a Middleware Library for Efficient and Transparent Distribution of Language Entities

> Erik Klintskog, Zacharias El Banna, Per Brand and Seif Haridi. The DSS, a Middleware Library for Efficient and Transparent Distribution of Language Entities. In *Proceedings of HICSS'37*, Hawaii, USA, 2004.

The paper describes the design and implementation of the DSS on a conceptual level. The Focus of the paper is on the DSS's interface to a programming system. The abstract entity model is introduced together with a description of the abstract operations interface. The interaction between a programming system thread and an abstract entity is explained by examples. Finally, an example of how a simple data structure is coupled to an abstract entity is presented in pseudo C++ code. Moreover, the internals of the DSS is briefly described. Elements of the coordination layer and the

messaging layer are introduced. This paper presents an overview of the DSS and serves as an introduction to our approach to generic distribution support.

My Contribution The design and development of the abstract entity model and the distribution strategy framework was done by me, with help from Zacharias El Banna. The paper was jointly written by me and Zacharias El Banna under supervision from Per Brand and Seif Haridi.

5.2 Paper B: The Design and Evaluation of a Middleware Library for Distribution of Language Entities

> Erik Klintskog, Zacharias El Banna, Per Brand and Seif Haridi. The Design and Evaluation of a Middleware Library for Distribution of Language Entities. In 8^{th} *Asian Computing Conference*, Mumbai, India, 2003.

The second paper describing the DSS focuses on the support for efficient distribution support by choice of distribution strategy. The paper contributes an evaluation of the DSS implementation and the DSS approach. The OzDSS system is introduced and benchmarked in respect to basic messaging and capability to handle different types of distribution scenarios. More precisely, the ability to distribute language entities over various numbers of nodes and various degrees of concurrency (i.e. threads per process) is measured. The evaluation clearly shows that the right choice of distribution strategy for a distributed data structure can improve the performance in the order of magnitudes compared to a protocol that does not match the use case of a data structure.

The paper can be seen as a complement to paper *A*, together the two papers give a complete description of the DSS on a conceptual level. However, still the messaging layer has only been briefly described. The evaluation of the DSS shows that the DSS is efficient and indicates that the ability to choose distribution strategy is the most important factor when creating efficient distributed applications.

My Contribution. The development of the OzDSS system was primary done by me. The Oz applications used to evaluate the performance of the OzDSS system was written by me. The paper was jointly written by me, Zacharias El Banna and Per Brand, under supervision from Seif Haridi.

5.3 Paper C: A Peer-to-Peer Approach to Enhance Middleware Connectivity

Erik Klintskog, Valentin Mesaros, Zacharias El Banna, Per Brand and S. Haridi. A Peer-to-Peer Approach to Enhance Middleware Connectivity. In *OPODIS 2003: 7^{th} International Conference on Principles of Distributed Systems*, Martinique, France, 2003.

Providing a single system image over a set of nodes that communicate over the Internet requires connectivity between the nodes. However, administrative domains, such as NATs and firewalls hinder connectivity. In addition, processes can change their network addresses and thus hinder connection establishment until the new network address has been propagated in the distributed system. This paper shows how message flooding can be used to find indirect communication paths in the implicit overlay network created by a distributed application. The model is based on unique node identifiers together with volatile network address information for each node of a distributed system.

The internals of the messaging layer of the DSS is described with focus on the modular design which supports simple customization for various environments, both operating system and networks. An example of a custom connection maintenance module is presented; the example implements message flooding over existing connections between processes. The implementation is compared to a raw socket application to measure the overhead imposed by the functionality of the messaging layer when it comes to messaging.

Apart from DSS specifics, the paper contributes insights into how processes self organize into a stable system in the presence of partial failures.

Together with paper *A* and paper *B* the paper gives a conceptual description of the DSS, from the programming system interface down to the operating system interface.

My Contribution. The idea of using peer-to-peer techniques to achieve higher connectivity was originally mine. The design of a peer-to-peer connection establishment protocol was done by me and Valentin Mesaros. The implementation in the DSS was done by Valentin Mesaros under my supervision. The messaging layer is designed by me and jointly implemented by me and Zacharias El Banna. The paper is a joint work primary by me and Valentin Mesaros, with help from Zachrias El Banna and Per Brand. The work was supervised by Seif Haridi.

5.4 Paper D: Securing the DSS

> Zacharias El Banna, Erik Klintskog and Per Brand. Securing the DSS. Technical Report T2004:14, Swedish Institute of Computer Science, SICS, November 2004.

The model of distributed language entities supported by the abstract entities of the DSS is referentially secure. A reference to a distributed language entity cannot be forged or guessed; only by proper delegation can a thread acquire access to language entities originating at remote processes. Referential security is a requirement for secure distributed applications. By programmatically restricting access to distributed language entities to trusted nodes, a distributed application can be made secure. However, for this to be true, referential security must be supported on the level of the implementation. This paper describes how the DSS is made secure, how the referential secure model is preserved in a distributed setting.

Different types of attacks are described together with the means to prevent them. For instance, a process can attempt to acquire unauthorized access to a coordination network and thus access information that by the language security-model it should not have. Alternatively, the process can merely try to disrupt the service of a coordination network of which it is not a member.

Extensions required to the DSS to cope with the identified security breaches are presented on the level of design and implementation. A secure messaging layer, with non-forgeable process references, secure connection establishment protocols, and encrypted communication channels, is the key to secure data distribution system. The paper shows how the messaging layer presented in paper *C* is made secure according to the latter proposed extensions.

My Contribution The design of the security support for the DSS was done by Zacharias El Banna and me. The implementation was primary done by Zacharias El Banna. The paper was written by Zacharias El Banna and me under the supervision of Per Brand.

5.5 Paper E: Home migration using a structured overlay network

> Erik Klintskog, Per Brand and Seif Haridi. Home migration using a structured overlay network. To be submitted for review.

Any home-based protocol in general and the coordination network in particular are vulnerable to loss of the home or a suboptimal placed home. A natural solution is to allow migration of the home to move from nodes that are to be shut down or to nodes that are better located. However, the challenge is to locate the home after it has moved. The paper describes the design and implementation of a migrating home protocol that makes use of a structured peer-to-peer system, the DKS, to implement a distributed directory service which stores correct home locations. The solution presented is DSS specific, but could easily be applied to any home-based protocol.

The structured peer-to-peer overlay system DKS is integrated in the DSS middleware and is provided as a basic service to the sub-protocols of the coordination layer. The DKS system is used to organize the nodes of a distributed application such that a highly available, fault tolerant, distributed directory service is realized.

In addition, the paper shows the strength of the sub-protocol model. Only one new coordination sub-protocol is needed to be able to migrate the coordinator. This can be used in conjunction with all the existing consistency, and reference sub-protocols to provide an additional large set of useful distribution strategies.

My Contribution. The idea to make use of a structured peer-to-peer system to store locations of mobile homes was mine. The design and development of the DKS based home migration protocol was done by me. The paper was written by me with help from Per Brand, and supervised by Seif Haridi.

5.6 Paper F: Fractional Weighted Reference Counting

Erik Klintskog, Anna Neiderud, Per Brand and Seif Haridi. Fractional Weighted Reference Counting. In *Proceedings of Euro-Par 2001*, Manchester, England, 2001.

The paper presents a scalable version of the Weighted Reference Counting distributed garbage collection algorithm, called Fractional Weighted Reference Counting (FWRC). The FWRC algorithm introduces no extra messages when references to distributed data structures are passed between processes. The FWRC is implemented in the coordination framework of the DSS and complemented with a time-lease mechanism, resulting in an efficient distributed garbage collector which tolerates failures.

My Contribution The original idea of improving Weighted Reference Counting was mine. The design and initial implementation in the Mozart system was done by me and Anna Neiderud. The implementation in the DSS was done by Zacharias El Banna, under my supervision. The paper was jointly written by me and Anna Neiderud, under supervision from Seif Haridi and Per Brand.

5.7 Paper G: Internal Design of the DSS

Erik Klintskog. Internal Design of the DSS. Technical Report T2004:15, Swedish Institute of Computer Science, SICS, 2004.

This technical report is solely dedicated to the implementation of the DSS middleware with focus on the design of the abstract entity interface and the coordination layer. Key concepts are highlighted and described, on the level of C++ classes. Examples from paper *A* and paper*B* that where described on a conceptual level are revisited and described at the level of the C++ implementation.

This paper, together with the conceptual description found in paper *A*, paper *B* and paper *C*, gives the complete picture of the DSS. This paper shows how the concepts are realized in practice. The paper is not a straightforward interface description, but puts the focus on how selected classes of the middleware interact to provide service.

My Contribution I wrote the technical report. The described implementation was jointly done by me and Zacharias El Banna.

5.8 Paper H: Coupling a Programming System to the DSS, a Case Study

Erik Klintskog. Coupling a Programming System to the DSS, a Case Study. Technical Report T2004:16, Swedish Institute of Computer Science, SICS, 2004.

The technical report describes the integration of the DSS to the programming system Mozart. The result, OzDSS, is described in detail. Essential when coupling a programming system to the DSS is how the internal model of threads and language entities are mapped to the abstract entities of the DSS. The model of threads and language entities of Mozart is described at a detailed level to explain the design choices made when developing the code that couples the DSS to Mozart.

Similarly to the description of the DSS found in paper *G* the description is on the level of C++ classes. Thus the description is a complement to the conceptual descriptions of how to couple a system to the DSS found in paper *A* and paper *B*. To show the challenges associated with different thread implementations, the C++DSS system is introduced. C++DSS is a C++ library which uses the DSS to implement different types of distributed language entities in the form of C++ classes. Mozart emulates threads, thus there is no risk of multiple threads accessing the DSS simultaneously. C++DSS, on the other hand, makes use of POSIX threads, thus simultaneous access to the DSS from multiple POSIX threads can happen. The fundamental differences in how threads are treated in a system that emulates threads (Mozart) to a system that make use of native-threads (C++DSS) is discussed.

The paper is concluded by a performance comparison between the OzDSS system and other distributed programming systems. We see that the OzDSS system outperforms "industry grade" Java-RMI and Java-CORBA implementations.

My Contribution I wrote the technical report. The OzDSS system was primary designed and implemented by me. C++DSS was jointly developed by me and Zacharias El Banna. The evaluation of different distributed programming systems was done by Zacharias El Banna with help from me.

Chapter 6

Experiences and Conclusions

The attached papers which are included in this dissertation contain discussions and conclusions, however, those discussions are restricted to the subject of each paper. In this chapter we discuss the work presented in this dissertation on a higher level. Section 6.1 describes the DSS from a software development perspective. Section 6.2 is devoted to related work and presents comparision of key issues of the design of the DSS to what is provided by other systems. Future work is presented in Section 6.3. The dissertation is concluded in Section 6.4.

6.1 The Distribution SubSystem in Perspective (Lessons Learned)

The DSS project has lasted three years and along the way we have acquired a fair amount of experience. The project has in some sense been going on for a longer time if the experiences contributed from Mozart are accounted for. In this section we highlight some of the experiences we have acquired and some design choices we have faced during the design and development of the DSS.

6.1.1 History

The idea of generic distribution support for programming systems originates from the successful implementation of transparent distribution support for Mozart [60]. Distribution support for Mozart was tightly integrated in the runtime system. After extending the language entities of the programming langue Oz with distribution capability, the focus swerved towards system aspects, such as failure handling, failure detection, and efficient messaging. The reason for the choice of focus was twofold. First, to be able to deploy dynamic distributed applications over Internet messaging had to be robust. Second, which actually was the inspiration to this dissertation, the tight coupling between the distribution and the Oz virtual machine made working on any level but the messaging level complicated. Distribution strategies where tightly integrated in the virtual machine. Extensive knowledge in the Mozart virtual machine internals was required to work, develop and maintain the distribution support.

The lessons learned from the Mozart system was, with financing from Microsoft, formulated into the report *An Architecture for Distributed Programming Platforms* [22]. The report presented the vision of a middleware that provides transparent distribution with control of non-functional aspects. The vision would later be realized as the Distribution SubSystem.

Within the two projects Pepito (Peer to Peer in Theory and Practice) [129], and PPC (Peer to Peer Computing), the model was refined into a design that finally lead to the implementation of the DSS middleware. Despite the origin as apart of the Mozart system, surprisingly little code could be reused. The tight coupling of the distribution support in Mozart to the virtual machine made technology transfer, except for the messaging layer, impossible. Even the messaging layer had tight couplings to Mozart, but some parts where generic enough to be used[1]. The requirement for a programming system independent interface resulted in the abstract entity model. Years of arrested ideas, because of the practical impossibility to work on the level of distribution protocols in Mozart, resulted in a component based framework for consistency protocols. The framework let us

[1]There are a less than 500 lines of the 25000 lines of C++ code in the DSS retained from the Mozart implementation.

experiment with different protocol choices; finally we could easily implement and integrate new protocols, since the DSS allows development of consistency protocols without altering the interface to the programming system.

6.1.2 The Importance of Abstractions

The purpose of network transparency is to hide the underlying network from the application programmer. Ultimately, how an application is deployed, on one machine or over multiple machines, should not change the behavior of the application. As pointed out in Section 1.2.3, we neither believe full transparency to be possible nor wanted for applications that are intended to execute in Internet. Instead we argue for a model which introduces control over non-functional aspects of distribution and reveals selected details of the network to the programmer. However, network information should not be exposed in its rawest form; instead it should be filtered and classified. Ultimately, the programmer who uses a distributed programming system should be able to reason about a distributed system on an abstract level.

As far as possible, the DSS tries to hide errors. However, at a certain point, the errors that stems from failures at the networking level cannot be concealed, instead the errors are reported to the programming level. Each of the three internal layers of the DSS classifies detected errors into failures exposed to higher layers. A failure at a lower layer is reported as an error to a higher layer. For example, given that one of the channels established by the messaging layer is lost because of a network failure. This is an error, the processes the channel is connected to is currently unavailable. The error is classified into a failure of a DSite object and reported to the coordination layer. The proxies and coordinators are checked to see if they are affected by the failed DSite. If the functionality of the coordination networks which the proxies and coordinators are members of are affected by the failed DSite, the error has caused a failure. Each proxy whose coordination network has failed reports an error to the abstract entity layer. The abstract entity reports the failure at the coordination layer as an error to the programming level as a failed language entity. At programming system, the error is experienced as a failed language entity which may or

may not cause an error in the application. The strength of this model is that the programming level will only be informed of network problems that affect the distributed language entities to which the process refers.

Internally, the DSS is structured by at least one explicit object per conceptual layer (messaging, coordination, abstract entity). The sequence described above, from discovered network perturbation to report language entity failure requires at least one object indirection per conceptual level. This structuring implies a certain overhead, compared with applications that perform socket communication only; the DSS imposes an overhead of approximately 100% compared to a simple C++ program which communicates over raw sockets. However the explicit internal structuring of the middleware caters to separation of concerns. The service provided by the messaging layer has greatly simplified development of the coordination layer. Moreover, the division of a distribution strategy into sub-protocols has simplified development of sub-protocols. The overhead is insignificant in the light of the functionality provided by the DSS, especially when the OzDSS is compared to other distributed programming languages. Moreover, as shown in attached paper *B* the dominant factor for efficient entity distribution is in many cases not messaging speed, but a well chosen distribution strategy.

6.1.3 The Concept of an Abstract Entity

A prime requirement of the DSS was to support multiple distribution strategies for a single language entity. For example, given a distributed object, it should be possible to choose between remote execution and different kinds of replication protocols. The development of generic language entity distribution support was an incremental procedure which finally resulted in the abstract entity and abstract thread model. At an early stage, distribution support was presented as abstractions of the distribution strategies. Interaction between a language entity and the distribution strategy was conducted by protocol operations, "send a remote request" or "transfer the state" to this process. This design was ruled out as being too low-level and would have made integration to a programming system complicated. Instead a more abstract type of distribution support was developed based

on a notion of consistency models, which finally resulted in the abstract entity model.

6.1.4 In Search for the Third Abstract Entity

Three types of abstract entities are implemented, the mutable, the immutable and the transient abstract entity. The mutable and immutable abstract entities where derived from the language entities in target languages and known consistency protocols. However, there were both language entities and protocols that were not described by the immutable and mutable abstract entities. Asynchronous messaging abstractions, such as ports in Mozart and processes identifiers in Erlang, were one example. Futures and data flow variables, found in Mozart and in some functional languages, was another example. Neither the mutable nor the immutable abstract entities are an ideal match for distributing these language entities. The mutable abstract entity could of course be used use to *program* the required behavior at the glue layer. However, this would violate the whole idea of an abstract entity, to cater to simple integration between distribution support and language entities.

The data flow variable starts as unbound which means that it has no value. The variable can be bound to a value o making it indistinguishable from o. After binding, the variable is said to be *bound* or *determined*. This monotonic behavior, first starting as a mutable and after being bound becoming an immutable, describes a middle ground between the two hitherto identified abstract entity types. The new abstract entity was called *transient*. To cater to streams we extended the transient abstract entity with an asynchronous update operation. The resulting abstract entity is a construct which can be updated asynchronously, until the value is fixed (or bound). With the three abstract entities the model is expressive enough to efficiently distribute entities from the object oriented, the functional, and the declarative concurrent programming paradigms.

6.2 Related Work

An overview of existing systems is already presented in Chapter 2, instead of repeating the discussion, this section highlights important details of the DSS and compares them to what exists in other systems. Focus is on the Abstract Entity model, the explicit notion of a coordination network and the framework of sub-protocols which supports custom protocol implementation. This related work section serves as a complement to the related work sections found in the attached papers. Here, we try to provide a greater view of the technology found in the DSS, while the related works in the papers focus on the contributions of the particular paper.

6.2.1 Abstract Entity Model

The abstract entity, or in a more general term, the abstract entity model represents the interface between a protocol and a language entity. We will in this section compare the abstract entity model with other interface models between distribution support systems and programming systems. Other models will be compared with the prominent features of the abstract entity model: (1) a distributed language entity is represented by conceptual equal instances at each process that refers the entity. (2) the interface supports both remote and local access, i.e. functional shipping and various replication protocols can be used over the same interface. To simplify coupling, (3) different semantic interfaces are provided.

FOG [51], a language extension to C++ implements distributed objects in the form of *fragmented objects* [87]. Similarly to the abstract entity model, a fragmented object is physically distributed to each process which refers to the object. How a fragmented object is kept consistent is open to customization, the distribution strategy is defined by a per object replaceable *connective-object*. The two models, fragmented objects and abstract entities are similar in their purpose, to separate interface from implementation and thus achieving code reuse. However, the two interface models differ in target languages. Fragmented objects solely targets the object oriented programming systems, and thus only supports the mutable abstract entity. Globe, discussed in the Chapter 2, implements an object model similar to fragmented objects. The proxy instance of Globe [8] resembles the abstract

entity both in the service provided and in internal structuring. One object implements the interface to the programming system object. No assumption is made on distribution strategy, however, the authors argue strongly for replication. Similarly to the distribution support of fragmented-objects only objects are supported. Globe, FOG/C++, and the DSS provide similar type of distribution support, physically distributed language entities that can be kept consistent with different distribution strategies. The DSS differs in that it is a general support system and not a distributed programming system; moreover, the abstract entity model is not restricted to distributed object only.

POEMS [136] is a distributed programming system strongly oriented towards clusters-of-workstations that has a construct similar to abstract entities. Distribution is on the level of objects. A distributed object is replicated at each process of the cluster. Each replica is a complete object, and operations can either be executed remotely or locally. How an operation is resolved is defined by a *load-balancing* strategy. The distributed object interface of POEMS implements some of the functionality the abstract entity implements. However, the model is solely devoted to objects, and thus does not show any of the generality found in the abstract entity model.

Actor [6] is a middleware which separates distribution aspects from the functionality by the use of *policies*. A policy describes a distribution strategy, but not its implementation. The model is similar to the model described in Figure 3.6(a) on page 60, thus on a lower level of abstraction than the abstract entity model.

The TACT toolkit [139] is a middleware which provides distribution of a data store on the level of data structures. The system provides an interface construct called a *conit*, which similarly to the abstract entity has no knowledge of the internal structure of programming system data. The two systems differ in the type of distribution support provided. The DSS strives for transparency by distribution under stronger consistency models, while TACT strives for efficiency by distribution under weaker consistency models.

To our knowledge, no system represents semantically different distribution support as the abstract entity model does, i.e. mutable, immutable

and transient. Immutable data is usually just replicated between processes of a distributed system. However exceptions exists, Java Party system identifies immutable objects [63], the DISCWorld [64] system identifies the need to pass immutable data structures by reference, and replicate on need-to-use basis. Moreover, the Emerald [19] programming system allows an object to be annotated as immutables and thus freely replicated among the nodes that hold references to it. However, immutability is just a hint and not a property enforced by the runtime system, different replicas can intentionally or unintentionally diverge.

6.2.2 Coordination Networks

The concept of a coordination network is often non-existing in systems which are restricted to remote execution distribution support [89]. References to data structures are represented as proxies [113] whose only purpose is to direct operations to the data structure, much similar to RPC [17]. However, systems which distribute data structures by replication require knowledge of every replica to keep each replica in a consistent state when the value of the distributed data structure is changed.

Coordination networks can either be static or dynamic. A static coordination network has a fixed number of participants, known at creation. A dynamic coordination network shrinks and grows as nodes joins and leaves the network. The DSS maintains dynamic coordination networks. DSM systems [95, 37], which execute on clusters of workstations, typically have static coordination networks. The nodes which refers a memory page is the nodes of the cluster, known when the application starts.

The protocols executed in the coordination networks of the DSS all shares the notion of a home or a coordinator. The protocols which keep replicas in a consistent state are similar to the home-based protocols implemented by the DSM systems [95, 29, 125]. As shown above, remote execution protocols can also be expressed as a set of proxies and a home, thus both RPC [17] and JavaRMI [89] implement the notion of a home. However, the abstract representation of a home in the form of a coordinator is unique to the DSS. The coordination-protocol model implemented by the DSS separates the notion of a home from the task of keeping a dis-

tributed data structure consistent. This allows keeping migration of data structure state orthogonal to migration of the home. Similarly to DSM systems which supports migration of the home [95, 68, 37] the DSS implements sub-protocols that migrate the coordinator of a coordination network.

6.2.3 Protocol Choice

The "one-size-fits-all" argument does not hold for data structure distribution support [72]. There exists not one protocol which provides distribution support for any arbitrary data structure with acceptable performance. Performance includes functional and non-functional properties, where the latter includes fault-tolerance, security, response-time (i.e. latency), bandwidth utilization, memory utilization and number of messages.

Traditionally, distribution strategy is assigned to data structures based on type. Java RMI [89] is one example of this, objects are distributed remote execution only[2]. Even though Mozart [60] implements different distribution strategies, each language entity is statically assigned one distribution strategy. This simple approach results in inflexible system because the type of a data structure does not necessary reflect how the data structure will be used. InterWeave [33] overcomes this limitation by allowing for assignment of distribution patterns, not based on type, but on expected usage pattern. However, as pointed in [23], allowing for custom protocol allocation can result in unpredictable semantics.

The Javanaise [57] system allows custom implementations of consistency protocols for distributed objects. An Object is replicated and kept consistent by a read-write-invalidation protocol. The system supports custom implementations of consistency protocols. The model is limited in comparison to the DSS in that it does not support remote execution. Moreover, no support is provided to simplify protocol development, such as the sub-protocol framework of the DSS.

Customization of consistency protocols is built into the design of Khazana [29] a DSM system which allows nodes to join and leave at runtime.

[2]In fact, an object can by inheritance be defined as serializable or remoting. However, we do not consider the replication possible by serialization to be a proper distribution strategy.

Consistency of memory page replicas is separated from the actual replication mechanism. Thus custom consistency models can be implemented to, primarily, capture different non-functional aspects of distributed shared memory. The DSS and Khazana show similarities in that both systems are programming system independent, and support customization, but differs considerably in the interface to the programming system. The DSS provides distribution support over abstract entities which capture single data structure semantics, Khazana on the other hand provides just shared memory pages.

Constructing protocols of sub-components is common for point to point protocols, as well as for group communication protocols. The Horus system [106] is the prime example of protocol composition. However, in difference from the sub-protocol composition in the DSS which is concerned with high level issues as garbage-collection and home-location, the components of Horus provide encryption, ordering of messages, and fragmentation of large messages.

6.3 Future Work

This dissertation has shown that, by the use of the DSS, distribution support on the level of programming language data structures can be provided by a generic middleware; however, the work presented here is just a first step in the direction of generic distribution support. To further evaluate the approach described in this dissertation the DSS should be coupled programming systems. This section presents future research direction related to the DSS middleware.

Couple other Programming Systems to the DSS

We have shown how the DSS is integrated to a programming system which is based on a virtual machine. How to couple the DSS to the compiler of a programming system has not been explored. Conceptually, the two approaches to integration should be the same, but most likely there are practical considerations.

As compiler techniques and hardware gets more efficient, programming

system development is no longer solely restricted to programming languages such as C and C++. .NET is a generic platform for object oriented programming languages, and Java has been used as target platform for distributed programming language projects [58, 11]. Both Java and .Net have attractive features such machine independent byte-code instructions sets, automatic memory management and support for threads. Integration of the DSS into Java and .Net is probably the preferable approach to support distributed programming language development in the future. It would be beneficial to implement the features of the DSS in Java or a .Net language (read C#).

Security

Attached paper *D* presents initial work being done on securing the DSS. This work focused on protecting a distributed system from attacks by maintaining reference security in a hostile environment. This is the first step towards realizing secure distributed programming systems. Concepts of security should be introduced on the level of single distributed language entities. One possibility we see is that security is included in the sub-protocol framework as a fourth dimension. This would enable defining the security properties for a distributed language entity as a non-functional property.

Programming a DSS-Enabled System

The DSS offers novel features when it comes to customization of distribution strategies for language entities. This opens a new dimension to the programmer. How the possibility to customize should be exposed in a convenient way to the programmer. What are the abstractions needed to simplify development of efficient distributed applications using the full strength of the DSS.

Automate Customization of Distribution Strategy

The abstract entity model is unique in its flexibility. The abstract entity interface allows customization of the protocol which implements coherent distribution of a data structure at runtime. No matter what protocol is

chosen, the abstract entity guarantees a certain consistency model. The abstract entity interface simplifies coupling of a programming system, very little has to be known about how to distribute data. Just the semantics of the data has to be known. However, when customizing the distribution strategy for a distributed entity the model provides little in the form of abstractions or classifications. Instead of expressing the type of distribution support required, the abstract entity interface requires specification of the particular protocols to be used.

Quantifying non-functional properties of distribution support have been tried in other distributed systems [136, 72]. Following this approach, the choice of distribution strategy would then be expressed by parameters that describe the wanted degree of security, acceptable latency, bandwidth usage, and fault tolerance. Internally, a composition of sub-protocols are put together to match the demands. This model would relieve the programmer of a distributed application from learning and understanding the behavior of each sub-protocol component.

Programming Language Interoperability

The DSS middleware component makes no assumption on the internal structuring of a programming system it is coupled to. Distribution support is on the level of abstract entities and operations on the abstract entities. An abstract entity represents a language entity which can be interacted with according to a semantic model. The different types of abstract entities are basic data types found in programming languages. The abstract entity model could be used, similarly to how the object abstraction is used in CORBA, as means for language interoperability. This would probably require a conceptual layer on top of the DSS which specifies interaction in more detail, something similar to the IDL of CORBA.

Are there more than three Dimensions?

The sub-protocols framework the DSS implements describes a distribution strategy in three dimensions. The benefits of the model should be clear. It supports extendibility because of simple sub-protocol development and expressiveness due to the composition of sub-protocols into distribution

strategies. We do not believe the non-functionality to be restricted to just three dimensions. Clearly, the model would benefit from instrumentability in the security dimension. This question, can a distribution strategy be described as separation in more than three dimensions, requires a considerable research effort.

Formalize the DSS

This dissertation has mainly presented practical and experimental work. A future area of work would be to formalize the DSS internals and interfaces. A proper formalization of the abstract entity interface could be used as a basis to prove that a particular language entity can be distributed using a particular abstract entity. Formalization of the consistency sub-protocols of the coordination network is necessary to show that an abstract entity provides its defined service.

6.4 Conclusion

This dissertation has described the design and implementation of a middleware which provides distribution support for programming systems. The middleware, called the Distribution SubSystem (DSS), has been coupled to the multi-paradigm programming system Mozart and C++. The Mozart coupling, called OzDSS is realized according to the integrated approach, and the C++ coupling, called C++DSS, according to the new entities approach. The DSS and the OzDSS distributed programming system have shown that our thesis is sound:

> *Efficient multi-paradigm programming language distribution support can be provided by a middleware.*

On a more detailed level we have shown that, to support the thesis, the implemented middleware is generic and is efficient. Generic implies that the middleware can be used to support distribution for multiple programming models or paradigms. Generality is provided by the abstract entity model that can express distribution support for three types of language entities, that is mutable, immutable, and transient language entities.

Two factors contributed to make the DSS efficient. First, the efficient messaging layer of the DSS. Second, the freedom in choosing distribution strategy on the language entity level, which makes it possible to choose the most optimal distribution strategy for a given language entity. Taken together, the two factors make the DSS efficient, as shown in the performance evaluations of the middleware.

With the purpose of showing the soundness of the design and implementation of the DSS, the DSS has been used to distribute the multi-paradigm programming language Oz. Successfully coupling the DSS to Mozart, a programming system which implements the programming language Oz, shows that (1), the DSS can be used to distribute a programming system, (2) the DSS can handle both the object-oriented, the functional and the declarative-concurrent programming paradigm, (3) the coupling of Mozart to the DSS, OzDSS, is an efficient distributed programming language which provides distribution support. The system is efficient when compared to other systems, and implements novel features such as indirect routing to traverse fire-walls. Moreover, the programming system provides a novel feature in that the distribution strategy can be chosen at runtime on single language entity basis.

The OzDSS system shows that a distributed programming system based on the DSS middleware results in a highly customizable system, which is a strong argument for generic distribution support.

Bibliography

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12), 1996.

[2] Luc Onana Alima, Sameh El-Ansary, Per Brand, and Seif Haridi. Dks (n, k, f): A family of low communication, scalable and fault-tolerant infrastructures for p2p applications. In *3rd IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, 2003.

[3] Luc Onana Alima, Ali Ghodsi, and Seif Haridi. A framework for structured peer-to-peer overlay networks. In *LNCS volume 3267 of the post-proceedings of the Global Computing*, 2004.

[4] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2), 1996.

[5] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang.* Prentice-Hall, second edition, 1996.

[6] Mark Astley, Daniel Sturman, and Gul Agha. Customizable middleware for modular distributed software. *Communications of the ACM*, 44(5), 2001.

[7] Arno Bakker, E. Amade, Gerco Ballintijn, Ihor Kuz, P. Verkaik, I. van der Wijk, Maarten van Steen, and Andrew S. Tanenbaum. The globe distribution network. In *Proceedings of the USENIX Annual Conference*, 2000.

[8] Arno Bakker, Maarten van Steen, and Andrew S. Tanenbaum. From remote objects to physically distributed objects. In *Proceedings of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, 1999.

[9] Arno Bakker, Maarten van Steen, and Andrew S. Tanenbaum. From remote objects to physically distributed objects. In *Proceedings of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, 1999.

[10] Henri E. Bal, Raoul Bhoedjang, Rutger Hofman, Ceriel Jacobs, Koen Langendoen, Tim Rühl, and M. Frans Kaashoek. Performance evaluation of the Orca shared-object system. *ACM Transactions on Computer Systems*, 16(1), 1998.

[11] Jorge Luis Victria Barbosa and Adenauer Corra Yamin. Holoparadigm: a multiparadigm model oriented to development of distributed systems. In *9th International Conference on Parallel and Distributed Systems*, 2002.

[12] Philip A. Bernstein. Middleware: A model for distributed system services. *Commun. ACM*, 39(2), 1996.

[13] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The midway distributed shared memory system. Technical report, Carnegie Mellon University, 1993.

[14] D. I. Bevan. Distributed garbage collection using reference counting. In *PARLE (2)*, 1987.

[15] Raoul Bhoedjang, Tim Ruhl, Rutger Hofman, Koen Langendoen, Henri Bal, and Frans Kaashoek. Panda: A portable platform to support parallel programming languages. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, 1993.

[16] Ken Birman, Robert Constable, Mark Hayden, Jason Hickey, Christoph Kreitz, Robbert van Renesse, Ohad Rodeh, and Werner

Vogels. The horus and ensemble projects: Accomplishments and limitations. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, 2000.

[17] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. In *Proceedings of the ACM Symposium on Operating System Principles*, 1983.

[18] Andrew Birrell, Greg Nelson, Susan S. Owicki, and Edward Wobber. Network objects. *Softw., Pract. Exper.*, 1995.

[19] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distributed and abstract types in emerald. *IEEE Transactions on Software Engineering*, 13(1), 1987.

[20] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the emerald system. *SIGPLAN Not.*, 21(11), 1986.

[21] Luc Boug, Jean-Franois Mhaut, and Raymond Namyst. Efficient communications in multithreaded runtime systems. In *Parallel and Distributed Processing. Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP '99)*, 1999.

[22] Per Brand, Seif Haridi, Konstantin Popov, and Erik Klintskog. An architecture for distributed programming platforms. Internal SICS report, 2001. URL: http://dss.sics.se/.

[23] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Lohr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3), 1998.

[24] G. Brose, K. Lohr, and A. Spiegel. Java does not distribute. In *Technology of Object-Oriented Languages and Systems, 1997. TOOLS 25*, 1997.

[25] R. Buyya, T. Cortes, and H Jin. Single system immage(ssi). *The International Journal of High Performance Computing Applications*, 2001.

[26] Pei Cao and Chengjie Liu. Maintaining strong cache consistency in the world wide web. *IEEE Transactions on Computers*, 47(4):445–457, 1998.

[27] Luca Cardelli. A language with distributed scope. In *Conference Record of POPL '95: 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.*, 1995.

[28] Luca Cardelli and Rowan Davies. Service combinators for web computing. *IEEE Trans. Software Eng.*, 25(3), 1999.

[29] John B. Carter, Anand Ranganathan, and Sai Susarla. Khazana: An infrastructure for building distributed services. In *International Conference on Distributed Computing Systems*, 1998.

[30] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for asynchronous systems. In *PODC*, 1991.

[31] D. Chen, C. Tang nad B. Sanders, S. Dwarkadas, and M. Scott. Exploiting high-level coherence information to optimize distributed shared state. In *Proc. of the 9th ACM Symp. on Principles and Practice of Parallel Programming*, 2003.

[32] DeQing Chen, Alan Messer, Dejan S. Milojicic, and Sandhya Dwarkadas. Garbage collector assisted memory offloading for memory-constrained devices. In *5th IEEE Workshop on Mobile Computing Systems and Applications*, 2003.

[33] DeQing Chen, Chunqiang Tang, Xiangchuan Chen, Sandhya Dwarkadas, and Michael L. Scott. Multi-level shared state for distributed systems. In *ICPP'02*, 2002.

[34] Gregory Chockler, Danny Dolev, Roy Friedman, and Roman Vitenberg. Implementing a caching service for distributed corba objects. In *Middleware*, 2000.

[35] Randy Chow and Theodore Johnson. *Distributed Operating Systems & Algorithms.* Addison Wesley, 1997.

[36] Jae Woong Chung, Kyu Ho Park, and Daeyeon Park. Moving home-based lazy release consistency for shared virtual memory systems. In *1999 International Conference on Parallel Processing*, 1999.

[37] Jae Woong Chung, Kyu Ho Park, and Daeyeon Park. Moving home-based lazy release consistency for shared virtual memory systems. In *International Conference on Parallel Processing*, 1999.

[38] Sylvain Conchon and Fabrice Le Fessant. Jocaml: Mobile agents for Objective-Caml. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, 1999.

[39] M. Dahm. Doorastha: a step towards distribution transparency. In *JavaGrande*, 2000.

[40] W. A. Domain. Extensible markup language (xml). http://www.w3c.org/XML.

[41] Frej Drejhammar. *Flow Java: Declarative Concurrency for Java.* Licentiate thesis, Royal Institute of Technology, Stockholm, Sweden, 2005. TRITA-IT-LECS AVH 04:15.

[42] Ericsson AB. Open source erlang.

[43] K. E. Kerry Falkner, P. D. Coddington, and M. J. Oudshoorn. Implementing Asynchronous Remote Method Invocation in Java. Technical Report DHPC-072, 1999.

[44] D. Q. M. Fay. An architecture for distributed applications on the internet: Overview of microsoft's .net platform. In *17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*, 2003.

[45] Ronaldo A. Ferreira, Christian Grothoff, and Paul Ruth. A transport layer a straction for peer-to-peer networks. In *3rd IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, 2003.

[46] Jeffrey M. Fischer and Milos D. Ercegovac. A component framework for communication in distributed applications. In *Proceedings*

of the 14th International Parallel and Distributed Processing Symposium (IPDPS'00), 2000.

[47] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5), 1998.

[48] The GCC Team. The gnu compiler collection. Web page, 2002. URL: http://gcc.gnu.org/.

[49] Kurt Geihs. Middleware challenges ahead. *IEEE Computer*, 34(6), June 2001.

[50] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip B. Gibbons, Anoop Gupta, and John L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA*, 1990.

[51] Yvon Gourhant and Marc Shapiro. Fog/c++: A fragmented object generator. In *Proceedings of the USENIX C++ Conference*, 1990.

[52] Mark Grand. *Java Language Reference*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, January 1997.

[53] Cary G. Gray and David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP*, 1989.

[54] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *SIGMOD Conference*, 1996.

[55] Rachid Guerraoui and Andr Schiper. Software-based replication for fault tolerance. *Computer*, 30(4), 1997.

[56] H. Guyennet, J-C. Lapayre, and M. Trehel. Distributed shared memory layer for cooperative work application. In *Proc. of the 22nd Conference on Local Computer Networks (LCN'97)*, 1997.

[57] D. Hagimont and D. Louvegnies. Javanaise: distributed shared objects for Internet cooperative applications. In *Middleware'98*, 1998.

[58] Michael Hanus. Distributed programming in a multi-paradigm declarative language. In *Principles and Practice of Declarative Programming*, 1999.

[59] S. Haridi, P-V. Roy, P. Brand, M. Mehl, R. Scheidhauer, and G. Smolka. Efficient logic variables for distributed computing. *ACM Transactions on Programming Languages and Systems*, 21(3), 1999.

[60] Seif Haridi, Peter Van Roy, Per Brand, and Christian Schulte. Programming languages for distributed applications. *New Generation Computing*, 16(3), 1998.

[61] Haskell. Haskell language resources, December 2002. http://www.haskell.org.

[62] Bernhard Haumacher, Thomas Moschny, Jrgen Reuter, and Walter F. Tichy. Transparent distributed threads for java. In *5th International Workshop on Java for Parallel and Distributed Computing in conjunction with the International Parallel and Distributed Processing Symposium (IPDPS 2003)*, 2003.

[63] Bernhard Haumacher and Michael Philippsen. Exploiting object locality in JavaParty, a distributed computing environment for workstation clusters. In *CPC2001, 9th Workshop on Compilers for Parallel Computers*, 2001.

[64] K. A. Hawick, H. A. James, and J. A. Mathew. Remote Data Access in Distributed Object-Oriented Middleware. Technical report, 1998.

[65] Maurice Herlihy and Michael P. Warres. A tale of two directories: implementing distributed shared objects in Java. *Concurrency: Practice and Experience*, 12(7), 2000.

[66] Ophir Holder, Israel Ben-Shaul, and Hovav Gazit. Dynamic layout of distributed applications in fargo. In *International Conference on Software Engineering*, 1999.

[67] Hung-Yun Hsieh and Raghupathy Sivakumar. On transport layer support for peer-to-peer networks. In *Proc. of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS)*, 2004.

[68] Weiwu Hu, Weisong Shi, and Zhimin Tang. Home migration in home-based software DSMs. In *Proc. of the 1st Workshop on Software Distributed Shared Memory (WSDSM'99)*, 1999.

[69] IEEE Computer Society. *Portable Operating System Interface (POSIX)—Amendment 2: Threads Extension (C Language)*. 345 E. 47th St, New York, NY 10017, USA, 1995.

[70] Suresh Jagannathan and Richard Kelsey. On the interaction between mobile processes and objects. In *Seventh Heterogeneous Computing Workshop*, 1998.

[71] H. A. James and K. A. Hawick. Data Futures in DISCWorld. In *Proc. of High Performance Computing and Networks (HPCN) Europe 2000*, 2000.

[72] Bo Norregard Jorgensen, Eddy Truyen, Frank Matthijs, and Wouter Joosen. Customization of object request brokers by application specific policies. In *IFIP/ACM International Conference on Distributed systems platforms*, 2000.

[73] Thilo Kielmann, Philip Hatcher, Luc Boug, and Henri E. Bal. Enabling java for high-performance computing. *Communications of the ACM*, 44(10), 2001.

[74] E. Klintskog. Coupling a programming system to the DSS, a case study. Technical Report T2004:16, Swedish Institute of Computer Science, SICS, 2004.

[75] E. Klintskog, V. Mesaros, Z. El Banna, P. Brand, and S. Haridi. A peer-to-peer approach to enhance middleware connectivity. In *OPODIS 2003: 7^{th} International Conference on Principles of Distributed Systems*, 2003.

[76] Erik Klintskog, Zacharias El Banna, Per Brand, and Seif Haridi. The design and evaluation of a middleware library for distribution of language entities. In *Advances in Computing Science - ASIAN 2003*, 2003.

[77] Erik Klintskog, Anna Neiderud, Per Brand, and Seif Haridi. Fractional weighted reference counting. In *LNCS 2150*, 2001.

[78] Erik Klintskog, Anna Neiderud, and Zacharias El Banna Per Brand Seif Haridi. Component-based distributed garbage collection. http://dss.sics.se, 2002.

[79] Kenji Kono, Kazuhiko Kato, and Takashi Masuda. Smart remote procedure calls: Transparent treatment of remote pointers. In *International Conference on Distributed Computing Systems*, 1994.

[80] Fred Kuhns, Douglas C. Schmidt, and David L. Levine. The design and performance of a real-time i/o subsystem. In *IEEE Real Time Technology and Applications Symposium*, 1999.

[81] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 1979.

[82] C.-W. Lermen and Dieter Maurer. A protocol for distributed reference counting. In *Conference Record of the 1986 ACM Symposium on Lisp and Functional Programming*, 1986.

[83] B. Liskov and L. Shrira. Promises. Linguistic support for efficient asynchronous procedure calls in distributed systems. In *In Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, 1988.

[84] S. Dwarkadas M. L. Scott, D. Chen and C. Tang. Distributed shared state. In *Intl. Workshop on Future Trends in Distributed Computing Systems*, 2003.

[85] Jason Maassen, Thilo Kielmann, and Henri E. Bal. Efficient replicated method invocation in java. In *Java Grande*, 2000.

[86] Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. Bal, and Aske Plaat. An efficient implementation of java's remote method invocation. In *Principles Practice of Parallel Programming*, 1999.

[87] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, and Marc Shapiro. *Fragmented objects for distributed abstractions*. 1994.

[88] Microsoft Corporation. Microsoft .net development. msdn.microsoft.com/net/, 2002.

[89] Sun Microsystems. Java remote method invocation specication, rev. 1.50, 1998.

[90] Mozart Consortium. http://www.mozart-oz.org.

[91] Anna Neiderud. Implementing transparent remoting for .net using the distribution subsystem. Internal SICS report, 2002. URL: http://dss.sics.se/.

[92] Christian Nester, Michael Philippsen, and Bernhard Haumacher. A more efficient RMI for java. In *Java Grande*, 1999.

[93] B. Clifford Neuman. Scale in distributed systems. *Readings in Distributed Computing Systems*, 1994.

[94] M. C. Ng and W. F. Wong. Orion: An adaptive home-based software distributed shared memory system. In *Seventh International Conference on Parallel and Distributed Systems (ICPADS'00)*, 2000.

[95] M. C. Ng and W. F. Wong. Orion: An adaptive home-based software distributed shared memory system. In *Seventh International Conference on Parallel and Distributed Systems (ICPADS'00)*, 2000.

[96] Hutchinson Norman C, Rajendra K. Raj, Andrew P. Black, Henry M. Levy, and Eric Jul. The Emerald programming language. Technical Report 87-10-07, Seattle, WA (USA), 1987.

[97] Object Managament Group. The common object request broker: Architecture and specification, revision 2.4.2. OMG Document formal/00-02-23, 2001.

[98] C. Perkins. *Mobile IP: Design Principles and Practices.* Addison-Wesley, 1997.

[99] Michael Philippsen and Matthias Zenger. JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11), 1997.

[100] José M. Piquer. Indirect reference counting: a distributed garbage collection algorithm. In *PARLE'91—Parallel Architectures and Languages Europe*, 1991.

[101] Kordale R., Krishnaswamy V., Bhola S., Bommaiah E., Riley G., Topol B., Torres-Rojas F., and Ahamad M. Middleware support for scalable services. In *Proceedings of the Fourth International Workshop On Community Networking*, 1997.

[102] P. W. Trinder R. F. Pointon and H-W. Loidl. The design and implementation of Glasgow Distributed Haskell. In *Lecture Notes in Computer Science*, 2001.

[103] Ingo Rammer. *Advanced .Net Remoting.* Apress, 2002.

[104] M. Raynal and A. Schiper. A suite of formal definitions for consistency criteria in distributed shared memories. In *Proceedings Int Conf on Parallel and Distributed Computing (PDCS'96)*, 1996.

[105] Robbert Van Renesse, Kenneth P. Birman, Bradford B. Glade, Katie Guo, Mark Hayden, Takako Hickey, Dalia Malki, Alex Vaysburd, and Werner Vogels. Horus: A flexible group communications system. Technical Report TR95-1500, 23, 1995.

[106] Robbert Van Renesse, Kenneth P. Birman, Bradford B. Glade, Katie Guo, Mark Hayden, Takako Hickey, Dalia Malki, Alex Vaysburd, and Werner Vogels. Horus: A flexible group communications system. Technical Report TR95-1500, 23, 1995.

[107] M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing Journal*, 6(1), 2002.

[108] Bert Robben, Bart Vanhaute, Wouter Joosen, and Pierre Verbaeten. Non-functional policies. In *Proceedings of the Second International Conference on Meta-Level Architectures and Reflection*, 1999.

[109] Lus Rodrigues. The road to a more configurable and adaptive communication and coordination support. In *9th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS 2003)*, 2003.

[110] Peter Van Roy. *On the separation of concerns in distributed programming: Application to distribution structure and fault tolerance in Mozart.* World Scientific, Tohoku University, Sendai, Japan, July 1999.

[111] Giovanni Russello, Michel R. V. Chaudron, and Maarten van Steen. Exploiting differentiated tuple distribution in shared data spaces. In *Euro-Par*, 2004.

[112] Per Sahlin. Efficient distribution of immutable data structures in the distributed subsystem middleware library. Master thesis, available for download from http://www.sics.se/s̃ahlin.

[113] M. Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *6th International Conference on Distributed Computer System*, 1986.

[114] Marc Shapiro, Peter Dickman, and David Plainfoss. SSP chains: Robust, distributed references supporting acyclic garbage collection. Technical Report 1799, inria, 1992.

[115] J. Snell and K. MacLeod. *Programming Web Applications with SOAP.* O Reilly, 2001.

[116] A. Snoeren, H. Balakrishnan, and M. Kaashoek. Reconsidering internet mobility. In 8^{th} *Workshop on Hot Topics in Operating Systems*, 2001.

[117] Alex C. Snoeren, Hari Balakrishnan, and M. Frans Kaashoek. Reconsidering internet mobility. In *Proc. 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, 2001.

[118] Paul Stelling, Cheryl DeMatteis, Ian T. Foster, Carl Kesselman, Craig A. Lee, and Gregor von Laszewski. A fault detection service for wide area distributed computations. *Cluster Computing*, 2(2), 1999.

[119] W. Richard Stevens. *UNIX Network Programming.* Prentice Hall, 1990.

[120] Volker Stolz and Frank Huch. Implementation of port-based distributed haskell. http://www-i2.informatik.rwth-aachen.de/Research/distributedHaskell/ i 2001.ps.gz.

[121] Sun Microsystems. Jini distributed leasing specification – revision 1.0.1, 1999.

[122] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall PTR, 2001.

[123] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall PTR, 2001.

[124] C. Tang, D. Chen, S. Dwarkadas, and M. Scott. Integrating remote invocation and distributed shared state. 2004.

[125] Chunqiang Tang, DeQing Chen, Sandhya Dwarkadas, and Michael L. Scott. Efficient distributed shared state for heterogeneous machine architectures. In *ICDCS'03*, 2003.

[126] Gerard Tel. *Introduction to Distributed Algoritms.* Cambridge University Press, 1994.

[127] E. Tilevich and Y. Smaragdakis. Nrmi: Natural and efficient middleware. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, 2003.

[128] Eli Tilevich and Yannis Smaragdakis. J-orchestra: Automatic java application partitioning, 2002.

[129] PEPITO PEer to Peer: Implementation and TheOry. http://www.sics.se/pepito.

[130] Josep Torrellas, Monica S. Lam, and John L. Hennessy. False sharing ans spatial locality in multiprocessor caches. *IEEE Trans. Computers*, 43(6), 1994.

[131] P. Trinder, R. Pointon, and H. Loidl. Runtime system level fault tolerance for a distributed functional language. In *Proceedings of Scottish Functional Programming Workshop, SFP'00*, 2000.

[132] Peter Van Roy, Per Brand, Seif Haridi, and Raphaël Collet. A lightweight reliable object migration protocol. In *Internet Programming Languages, ICCL'98 Workshop*, 1999.

[133] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming.* MIT Press, March 2004.

[134] W3C. Web services activity. http://www.w3.org/2002/ws/.

[135] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. In *Mobile Object Systems: Towards the Programmable Internet.* Springer-Verlag: Heidelberg, Germany, 1997.

[136] Jie Wei, Cai Wen Tong, and Stephen John Turner. A parallel object-oriented system - its dynamic load balancing strategies and implementation. In *HPC Asia*, 2001.

[137] Danny Weyns, Eddy Truyen, and Pierre Verbaeten. Distributed threads in java. *Sci. Ann. Cuza Univ.*, 11, 2002.

[138] C. Wikstrom. Distributed programming in erlang. In *Proceedings of the First International Symposium on Parallel Symbolic Computation (PASCO'94)*, 1994.

[139] Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of Fourth Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[140] Weimin Yu and Alan L. Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency - Practice and Experience*, 9(11), 1997.

[141] Victor C. Zandy and Barton P. Miller. Reliable network connections. In *Proceedings of the eighth annual international conference on Mobile computing and networking*, 2002.

Swedish Institute of Computer Science

SICS Dissertation Series

01: Bogumil Hausman, *Pruning and Speculative Work in OR-Parallel PROLOG*, 1990.
02: Mats Carlsson, *Design and Implementation of an OR-Parallel Prolog Engine*, 1990.
03: Nabiel A. Elshiewy, *Robust Coordinated Reactive Computing in SANDRA*, 1990.
04: Dan Sahlin, *An Automatic Partial Evaluator for Full Prolog*, 1991.
05: Hans A. Hansson, *Time and Probability in Formal Design of Distributed Systems*, 1991.
06: Peter Sjödin, *From LOTOS Specifications to Distributed Implementations*, 1991.
07: Roland Karlsson, *A High Performance OR-parallel Prolog System*, 1992.
08: Erik Hagersten, *Toward Scalable Cache Only Memory Architectures*, 1992.
09: Lars-Henrik Eriksson, *Finitary Partial Inductive Definitions and General Logic*, 1993.
10: Mats Björkman, *Architectures for High Performance Communication*, 1993.
11: Stephen Pink, *Measurement, Implementation, and Optimization of Internet Protocols*, 1993.
12: Martin Aronsson, *GCLA. The Design, Use, and Implementation of a Program Development System*, 1993.
13: Christer Samuelsson, *Fast Natural-Language Parsing Using Explanation-Based Learning*, 1994.
14: Sverker Jansson, *AKL - - A Multiparadigm Programming Language*, 1994.
15: Fredrik Orava, *On the Formal Analysis of Telecommunication Protocols*, 1994.
16: Torbjörn Keisu, *Tree Constraints*, 1994.
17: Olof Hagsand, *Computer and Communication Support for Interactive Distributed Applications*, 1995.
18: Björn Carlsson, *Compiling and Executing Finite Domain Constraints*, 1995.
19: Per Kreuger, *Computational Issues in Calculi of Partial Inductive Definitions*, 1995.
20: Annika Waern, *Recognising Human Plans: Issues for Plan Recognition in Human-Computer Interaction*, 1996.
21: Björn Gambck, *Processing Swedish Sentences: A Unification-Based Grammar and Some Applications*, June 1997.
22: Klas Orsvärn, *Knowledge Modelling with Libraries of Task Decomposition Methods*, 1996.
23: Kia Höök, *A Glass Box Approach to Adaptive Hypermedia*, 1996.
24: Bengt Ahlgren, *Improving Computer Communication Performance by Reducing Memory Bandwidth Consumption*, 1997.
25: Johan Montelius, *Exploiting Fine-grain Parallelism in Concurrent Constraint Languages, May*, 1997.
26: Jussi Karlgren, *Stylistic experiments in information retrieval*, 2000
27: Ashley Saulsbury, *Attacking Latency Bottlenecks in Distributed Shared Memory Systems*, 1999.
28: Kristian Simsarian, *Toward Human Robot Collaboration*, 2000.
29: Lars-Åke Fredlund, *A Framework for Reasoning about Erlang Code*, 2001.
30: Thiemo Voigt, *Architectures for Service Differentiation in Overloaded Internet Servers*, 2002.
31: Fredrik Espinoza, *Individual Service Provisioning*, 2003.
32: Lars Rasmusson, *Network capacity sharing with QoS as a financial derivative pricing problem: algorithms and network design*, 2002.
33: Martin Svensson, *Defining, Designing and Evaluating Social Navigation*, 2003.
34: Joe Armstrong, *Making reliable distributed systems in the presence of software errors*, 2003.
35: Emmanuel Frecon, *DIVE on the Internet*, 2004.
36: Rickard Cöster, *Algorithms and Representations for Personalised Information Access*, 2005
37: Per Brand, *The Design Philosophy of Distributed Programming Systems: the Mozart Experience*, 2005
38: Sameh El-Ansary, *Designs and Analyses in Structured Peer-to-Peer Systems*, 2005