2017 IEEE 37th International Conference on Distributed Computing Systems

Fast and Flexible Networking for Message-oriented Middleware

Lars Kroll, Alexandru A. Ormenișan, and Jim Dowling
Department of Software and Computer Systems, School of ICT
KTH Royal Institute of Technology, Stockholm, Sweden

***Abstract*—Distributed applications deployed in multi-datacenter environments need to deal with network connections of varying quality, including high bandwidth and low latency within a datacenter and, more recently, high bandwidth and high latency between datacentres. In principle, for a given network connection, each message should be sent over the best available network protocol, but existing middlewares do not provide this functionality. In this paper, we present KompicsMessaging, a messaging middleware that allows for fine-grained control of the network protocol used on a per-message basis. Rather than always requiring application developers to specify the appropriate protocol for each message, we also provide an online reinforcement learner that optimises the selection of the network protocol for the current network environment. In experiments, we show how connection properties, such as the varying round-trip time, influence the performance of the application and we show how throughput and latency can be improved by picking the right protocol at the right time.**

***Keywords*-transport protocols; middleware; machine learning**

I. Introduction

Recently, distributed systems have undergone a shift from local deployments at a single site to geo-distributed multi-datacenter deployments. This trend is caused by the requirement to have data, that is generated globally, locally available for consumption at low latency (e.g., videos on YouTube and Netflix, or music on Spotify) or analysis (e.g., scientific data in Genomics, Climate Science, or Particle Physics) in a timely manner. Some of these systems even try to exploit resources that reside at the logical extremes of a network, employing a number of approaches loosely clustered under the term *edge computing*. These kinds of environments feature a variety of network conditions ranging from low latency, high bandwidth intra-rack connections, over high latency, high bandwidth intercontinental links, to high latency, low bandwidth peers at the edges, making it increasingly challenging to design middleware solutions that perform well for every aspect of such distributed systems.

In addition to these geospatial interconnection considerations, there has also been rapid growth in the production, storage, and processing of unstructured data, that has led to the development of high-performance distributed systems to both manage, transport, and process these huge volumes of data. For example, big data analytics frameworks like Apache Spark [1], [2] and Apache Flink [3] have the challenging task of both moving and processing large quantities of data while providing low latency control over the executing tasks. The key to performance in such systems is exploiting the inherent parallelism between control flow and data flow, as well as within the data flow itself.

Message passing frameworks have shown themselves to be efficient at exploiting parallelism while avoiding many of the programming issues that come with synchronisation. Both Spark and Flink, for example, use Akka [4] a runtime for reactive applications written for the Java Virtual Machine (JVM). Another such framework is Kompics, a framework for building stateful, concurrent, message-passing components [5]. Both Kompics and Akka have long supported a message-oriented middleware layer to allow for distributed deployments. In Akka's case the subsystem is called `akka-remote` and its default protocol is the Transmission Control Protocol (TCP), although it also provides support for the User Datagram Protocol (UDP) and the Secure Sockets Layer (SSL) using Netty [6] as a networking library. Similarly, Kompics has moved to Netty. However, these existing message transport implementations have a serious limitation: They exhibit unsatisfactory performance on links with a high bandwidth delay product (BDP) (such as intercontinental connections), and links with high packet loss rates (such as wireless links). This limitation is prohibitive for internet scale multi-datacenter deployments, or for systems that employ peer-to-peer (P2P) methods.

Of course, this issue can be solved at the application layer using the UDP transport implementations instead. For example, Low Extra Delay Background Transport (LEDBAT) [7] has been implemented on top of Kompics/Netty/UDP before. However, the implementation of congestion control protocols at the application level in message-passing frameworks without priority queues (Kompics, Akka) results in inconsistent performance due to their timing sensitive nature. There are alternative, more performant, solutions already implemented in user-level or kernel-level libraries. Among those are Aspera FASP [8], the UDP-based Data Transfer Protocol (UDT) [9], and the Performance-oriented Congestion Control (PCC) [10]. While FASP is commercial, and PCC is too recent to have a well tested implementation, UDT support has been part of Netty since version 4 from 2013. Support for protocols such as UDT at the networking level opens up new possibilities when building distributed systems, and, in particular, message-based systems where fine-grained control over the protocol an individual message

should use as a network transport is possible.
In this paper we present a messaging middleware layer for the Kompics component framework, called *KompicsMessaging*, which uses Netty to provide transport via UDP, TCP and UDT. The choice of transport protocol is available on a per-message basis at runtime, providing the maximum flexibility to adapt to changing network conditions. Additionally, we present an adaptive algorithm that uses online reinforcement learning techniques to shift traffic between two peers automatically between TCP and UDT, as these protocols give very similar guarantees. We introduce message interfaces that allow for flexible designs, such as addressing of "virtual nodes".
We evaluate our messaging middleware on Amazon EC2, both within and across data centres, to investigate the effects of round trip time (RTT) on throughput. We demonstrate the gains in responsiveness and throughput that can be made by picking the right transport protocol for the right message.

The paper is structured as follows. Section II introduces the Kompics model and Java framework, as well as the Netty framework, and reinforcement learning. In section III we present our solution for the messaging middleware layer and give concrete examples for working with the provided APIs and abstractions. We then describe in section IV the implementation of the adaptive transport selection system. We evaluate our solutions in section V, and describe related work in section VI. Finally, we summarise the paper and give suggestions for future work in section VII.

II. Background

A. The Kompics Component Model

Since many of the design decisions for KompicsMessaging are directly related to the way the Kompics component model [5] itself works, we will give a short introduction here. More details can be found in [11].
Semantics. Kompics is a programming model for distributed systems that implements protocols as event-driven *components* connected by *channels*. Kompics provides a form of type system for events, where every component declares its required and provided *ports* – they can be thought of as "services" –, which in turn define which event-types travel along the channels that connect them and in which direction. On a port type, the "service specification" for a port, events are declared as either *indications* or *requests*. Within a component that *provides* a port P with indication event I and request event R, only instances of I can be *triggered* ("sent") and only instances of R (or their subtypes) can be *handled* (see below). Conversely, within a component that *requires* P only instances of R can be *triggered* and only instances of I (or their subtypes) can be *handled*.
The channels connecting ports provide first-in-first-out (FIFO) order exactly-once (per receiver) delivery and events are queued up at the receiving ports until the component is scheduled to execute them.
Scheduling. A component is guaranteed to be only scheduled on one thread at a time and thus has exclusive access to its internal state without the need for further synchronisation. Different components, however, are scheduled in parallel in order to exploit the parallelism expressed in a message-passing program. When a component is scheduled, it handles one event at a time, and keeps handling events until either there are no more events queued at its ports or a configurable maximum number of events to be handled is reached. After the component has finished handling events, it will be placed at the end of the FIFO queue of components waiting to be scheduled. Tuning the configurable maximum number of events to be handled enables developers to tradeoff increased throughput, where higher values maximise cache reuse through fewer component context switches, against fairness, that is avoiding starvation of components with fewer queued events.
Event-handling. In contrast to Actor systems like Akka [4] or Erlang [12], events in Kompics are not addressed to components in any way, but are instead published on all connected channels. In this way the same event can be received by many components. The components themselves decide which events to handle and which to ignore by subscribing event *handler*s on their declared ports. Note that ignored messages are silently dropped, which is necessitated by the channel broadcasting model, that is to say, as opposed to Erlang and Akka, in Kompics it is often completely correct to simply ignore a large number of events.
Matching of events to handlers is based on the events' type-hierarchy, although there are some Kompics extensions that provide pattern matching as well.

Note that in addition to the default Java library implementation of Kompics, there are alternatives written in Scala and Python as well, which are not considered in this paper.

B. Netty

Netty [6] is a non-blocking I/O (NIO) network application framework for the JVM. It supports the rapid development of maintainable high-performance and high-scalability protocol servers and clients. Additionally, Netty provides a native byte buffer library that minimises unnecessary copying of data. Netty is a perfect match for Kompics, because it is event-driven and asynchronous, making the interface between the two systems both seamless and efficient. In order to provide non-blocking operations Netty has a highly customisable concurrency model employing constructs such as thread pools and futures, as well as a configurable channel handler pipeline.
Netty is well tested and already used in a number of large projects such as Apache Spark [1], [2] and Apache Flink [3], as well as Akka [4] and at companies like Facebook, Google and Spotify.[1]

[1]Source http://netty.io/wiki/adopters.html.

C. Reinforcement Learning

Reinforcement learning is an approximate dynamic programming approach from machine learning, where an agent executes actions based on its own state and the state of its environment so as to maximise some kind of numerical reward [13]. It is an unsupervised learning algorithm. Instead of being told which actions to take, agents explore their environment to take (initially random) actions, and over time they learn a policy that tells them which action is expected to give the best reward given its current state and the state of its environment. Especially in the context of control systems, there is a focus on online learning while maximising reward, by employing 'good' strategies. This is referred to as *exploitation*, and an often difficult balance must be maintained between exploring new actions to take and exploiting the learnt policy to maximise reward.

A reinforcement learning problem consists of the following elements:

1) A *policy* π prescribes the agent's behaviour in form of a strategy by mapping perceived states to actions that are to be taken while in that state. It is the policy's task to balance exploration and exploitation as appropriate. A policy that always exploits is called *greedy*, while a policy that always explores would simply be random. A common policy used to balance the two extremes is called *ϵ-greedy*, which explores with probability ϵ and otherwise always exploits. Similar to the approach of *simulated annealing*, it is reasonable to start with a relatively high ϵ, and then gradually reduce it to a certain minimum value, in order to eventually let the learner converge to the desired degree.
2) The *reward function* describes the learning objective and maps states (or state-actions pairs) to numerical rewards, which the agent is trying to maximise. The true reward function is typically unknown (and possibly unknowable, if it can change over time), and each reward R must be observed by exploration.
3) A *value function* V describes the desirability of a certain state, based on its expected long term reward. While the reward function is given by the environment, the value function must be learned and possibly adapted to changing environments by the agent. The value function is also dependent on the policy, and thus often written V^{π} for a policy π.
4) Additionally, an agent might have a *model* of the environment which predicts states that result from actions, and can be used for *planning*. Without a model, a state-value function V is often not useful, as the next state resulting from an action can not be known in advance. In such circumstances it is necessary to learn an action-value function Q instead, or as before Q^{π} for a specific policy.

One of the most important reinforcement learning methods is known as *temporal difference*(TD) learning, and specifically TD(λ) [13], which uses the notion of temporal distance to discount the influence of reward samples in learning a value function.

```java
public class Network extends PortType {{
 request(Msg.class);
 request(MessageNotify.Req.class);
 indication(Msg.class);
 indication(MessageNotify.Resp.class);
}}
```

Listing 1. Kompics Network Port.

```java
public interface Msg<H extends Header> extends
     KompicsEvent {
 public H getHeader();
}
```

Listing 2. The Msg interface.

III. Messaging in Kompics

A. API

The core of the KompicsMessaging API is a Kompics component called `NettyNetwork` which provides Kompics' network port, shown in listing 1 and interfaces the Kompics runtime with Netty's execution model. The network port allows messages that implement the `Msg` interface (listing 2) to travel on both directions on network channels and it allows to request notification of a message's delivery status with a `MessageNotify.Req`, which will be answered with a `MessageNotify.Resp` that indicates whether the message was sent successfully. If no notification is requested messages are simply "fire and forget".

As can be seen in listings 2 and 3 both `Msg` and `Header` allow subtypes of their generic parameters and neither those nor `Address` (listing 4) provide any implementation. In fact, there are default implementations for all of these interfaces in the `NettyNetwork` package, but the interfaces specify the minimum features the actual network implementation requires to work. The idea behind this design is, to allow

```java
public interface Header<Adr extends Address> {
 public Adr getSource();
 public Adr getDestination();
 public Transport getProtocol();
}
```

Listing 3. The Header interface.

```java
public interface Address {
 public InetAddress getIp();
 public int getPort();
 public InetSocketAddress asSocket();
 public boolean sameHostAs(Address other);
}
```

Listing 4. The Address interface.

```java
public class RoutingHeader<Adr extends Address>
    implements Header<Adr> {
 private final BasicHeader<Adr> base;
 //Forwardable Trait
 private Route<Adr> route = null;
 @Override
 public Adr getSource() {
  if (route != null) {
   return route.getSource();
  }
  return base.getSource();
 }
 @Override
 public Adr getDestination() {
  if (route != null && route.hasNext()) {
   return route.getDestination();
  }
  return base.getDestination();
 }
 ...
}
```

Listing 5. A multi-hop routing Header.

application designers to pick implementations that suit their requirements without having to extend existing classes and rely on runtime type-casts. For example, if someone wanted to implement messages that can be forwarded through multiple intermediary hosts, but finally replied to directly, they might add an `origin` or `replyTo` field to the their `Header` implementation. Or an `Address` implementations might provide an addtional `id` field which disambiguates hosts with multiple network interfaces in use.
When a `NettyNetwork` component is initiated, it is provided with the protocols and ports to listen on. Supported protocols at this point are UDP, TCP, and UDT. A single instance of the component only allows one port to listen on per protocol, but if more are required another instance with a different configuration can simply be started. Every instance manages its own Netty handlers and channels.

B. Semantics

It is important to note that the semantics of network messages differ from those of Kompics channels. While Kompics channels provide FIFO order with exactly-once delivery, network messages provide only at-most-once semantics. Even over TCP and UDT a sudden channel drop may lead to the loss of messages, and replicating message acknowledgements on the middleware layer adds unnecessary complexity and is, in general, not desirable. If message delivery is a concern for an application, it may implement resending and acknowledgements itself. Additionally, while TCP and UDT maintain Kompics' FIFO semantics, they are not guaranteed when using UDP. Adding these semantics would defeat the point of having a lightweight protocol like UDP available in the first place.
The stark difference in semantics is balanced by the fact that KompicsMessaging does *not* provide location transparency and it is always clear to the developer whether or not messages might go over the network.

However, messages that *might* go over the network do not always actually get serialised and sent over a link. Sometimes it is desirable to use "addressable" components in a way similar to Akka's actors. We provide support for this with a package for *virtual networks*, where, in addition to the network interface's IP address and port, an identifier is assigned to certain subtrees of the Kompics component hierarchy. Those subtrees are referred to as virtual nodes or vnodes. While communication across vnodes happens almost exclusively via the network port, the messages they send to each other within a single host are actually never serialised and deserialised. Instead the `NettyNetwork` component detects such occurences using the `sameHostAs` method specified in the `Address` interface (cf. listing 4) and "reflects" the messages back up through the network port. In this scenario a special `VirtualNetworkChannel` implementation takes care that messages are only delivered to the destination vnode.
As a result of this behaviour a programmer should never expect to receive copies of network messages, and instead stick to the default Kompics philosophy of *immutable messages/events*.

C. Transport Protocol Selection

While the `Header` interface does not require every implementation to have a setter for the `Transport` field, implementations are free to provide one. This leaves the system designer with the decision whether or not the transport protocol should be hardcoded for a specific message type, or injected by a configuration at creation time, or even replaced on the fly by an interceptor component between the message sending components and the `NettyNetwork` component. Such an adaptive transport system could measure network variables or have access to some deployment descriptor and would then decide the best protocol to use for a specific message type at runtime. See section IV for details on our implementation of such a system.
Regardless of how the protocol selection for a message is made, the `NettyNetwork` component ensures that the required channels, if any, are available. If necessary, new channels are created and messages delayed until the requested channels are available. When channels are no longer in use, they might eventually be dropped to reclaim resources. However, our implementation is very conservative with this, since channel establishment might be expensive, as is, for example, the case with hole punching to circumvent NATs. Thus generally channels will be kept open as long as possible.

IV. Adaptive Transport Selection

Since it can be very difficult or even impossible to statically select the network transport protocol to be used for transferring data, we have implemented an interceptor component that tries to learn the best ratio between TCP and UDT dynamically.

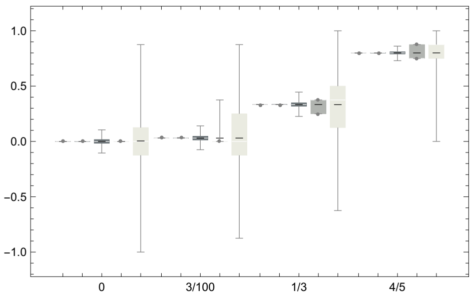

Figure 1. Distribution of observed selection ratios (100% TCP ~ −1.0, 100% UDT ~ 1.0) of the probabilistic (*Random*) and the *Pattern* selector compared to the *Target* ratio. *Episodes* contain around 1600 messages and there are 16 messages on the *Wire* concurrently. Every dataset has around 160000 entries. It is very clear that the probabilistic selection policy can be quite far off target even for full episodes, which would impede the learner significantly.

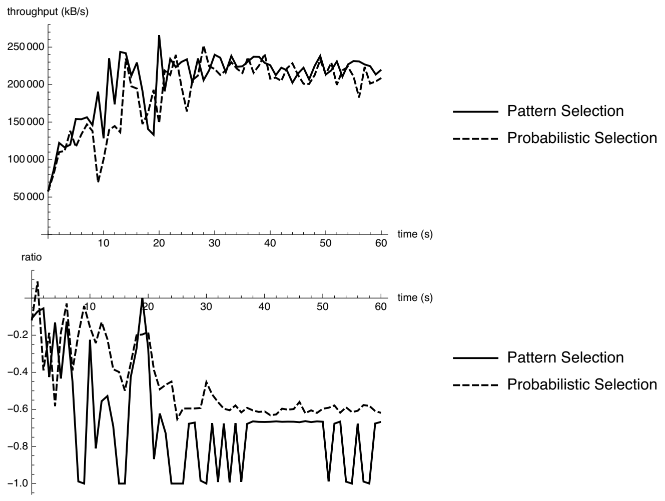

Figure 2. Impact of the protocol selection policy on throughput and true protocol ratio (100% TCP ~ −1.0, 100% UDT ~ 1.0). Probabilistic ratio selection is less accurate (smoother) than pattern selection, leading to slightly slower convergence in the throughput.

These two protocols were chosen for their practically identical properties, but very different behavioural characteristics. The ideas presented in this section could, of course, be extended to other protocols with little difficulty.

A. Integration & Structure

The `data-network-interceptor` component introduces a pseudo-protocol `Transport.DATA`, which it replaces from implementations of a specific `DataHeader` transparently at runtime with either `Transport.TCP` or `Transport.UDT`. The component is placed between the `NettyNetwork` component and any consumer, using `ChannelSelectors` to route non-data messages past it, directly to the network component. The `DataNetwork` component is provided to wrap the interceptor and the network component, in order to simplify setup.

Internally this data interceptor component controls the flow of a data stream to a specific destination node by queuing outgoing messages, and then releasing them to the network layer at an adaptive rate, inserting the transport protocol chosen by the current *protocol selection policy* (PSP). A PSP selects the transport protocol for a specific message based on internal state and the target protocol ratio r, as prescribed by a *protocol ratio policy* (PRP). Both PSP and PRP can be configured at system start, and custom policies can be provided by implementing the `ProtocolSelectionPolicy` and `ProtocolRatioPolicy` interfaces respectively. We investigated and provide a number of policies as described in the next section.

B. Protocol Selection Policies

The goal of a protocol selection policy is to assign a transport protocol to an individual message, before it gets passed off to the networking component. The policies we are interested in try to achieve a certain target ratio r between messages sent via TCP and those sent via UDT. A “good” policy in our use case is one that does not stray far from the target ratio even if we consider only relatively short message sequences out of the total stream emitted.

We consider r in different representations in the following sections, which are convenient for different circumstances. The most convenient form for analysis and visualisation of data, is the form $r \in [-1,1] \subset \mathbb{R}$ where -1 indicates 100% TCP, 0 indicates a 50-50 mix between TCP and UDT, and 1 indicates 100% UDT. For talking about probabilities it is convenient to express $r \in [0,1] \subset \mathbb{R}$ where r is the probability of picking UDT, and thus 0 indicates 100% TCP, $\frac{1}{2}$ indicates a 50-50 mix between TCP and UDT, and 1 indicates 100% UDT. Lastly when talking about patterns it is easiest to simply state how many TCP packets should be emitted for every UDT packet or vice versa. In this form $r \in [0,1] \subset \mathbb{Q}$ such that $r = \frac{p}{q}$ indicates p Ps for every q Qs and the mapping between TCP, UDT and P,Q is defined by the sign of the $r \in [-1,1] \subset \mathbb{R}$ representation. We will use these forms mostly interchangeably as they are typically clear from context and can easily be converted into each other.

1) Probabilistic Selection: This baseline algorithm, implemented by `RandomSelection`, uses a random variable with Bernoulli distribution to select the protocol, such that the protocol ratio r is the probability of choosing UDT instead of TCP. While the law of large numbers dictates that eventually the ratio between the protocols will approach r, there is no notion of short-term protocol balance in this selection policy, leading to concerns over the distortion of rewards for the PRP learner, as there could be significant skew within one update interval.

2) Issues with Probabilistic Selection: In order to investigate the severity of the skew issue in probabilistic selection, consider the following experiment: On a 100MB/s link with

10ms delay we send messages of 65kB each. One learning *episode* (s. section IV-C2) is 1s long. That means during one episode approximately 1600 messages are sent, and there should be 16 messages concurrently on the wire at any time. If we run the probabilistic selector with these values over a few different target ratios, we can see in figure 1 that while the mean (black horizontal line in a box) is always fairly close to the target value, the max/min can incur around 0.1 skew even for a full episode of 1600 messages. For short sequences of 16 messages, the max/min show a 0.5 skew with rather larger 75th percentile boxes. That means that for 50-50 almost any combination of protocols can be on the wire concurrently.

As figure 2 indicates, this behaviour leads to somewhat slower convergence of the learner in practice, but, in the worst case with very small ϵ (s. section IV-C2), it could prevent convergence altogether, as the learner would literally be learning the wrong values sometimes, and might not be able to recover from it before there is too little exploration to ever revisit those values. However, at least in figure 2 both implementations eventually achieve the same performance, so the probabilistic policy is not completely infeasible in practice. It can also be noted that the probabilistic learner displays a much smoother, but less accurate, behaviour in the true protocol ratio, that is the ratio that is measured on the receiver side, not the one prescribed by the PRP instance.

3) Pattern Selection: Instead of a random variable, this policy employs heuristics to find an interleaving pattern between UDT and TCP messages, such that a) at any point within the pattern the deviation from the target ratio r (s. above), which must be a *rational number*, is relatively small, and b) a complete run of a pattern has no deviation from r. The first target results in consecutive runs of either protocol being as short as possible, such that the protocols 'alternate' rapidly. For example, if $r = \frac{1}{2}$ and we use u to specify that UDT is selected and p to specify that TCP is selected, then $(up)^*$ and $(pu)^*$ (using regular expression notation), would be optimal patterns. For $r = \frac{1}{3}$ the patterns $(pup)^*$, $(ppu)^*$, and $(upp)^*$ would all be equivalent in long runs (ignoring imbalance caused by partial execution at the beginning or the end of a finite string).

4) Patterns: To formalise and generalise the above intuition behind pattern selection somewhat, consider values such that $r = \frac{p}{q} \in \mathbb{Q}$ with $p \le q \in \mathbb{N}$. Instead of emitting p and u, for consistency, we now want to emit p Ps for every q Qs. Similar to the kleene star notation above (e.g. P^*), we write P^p to denote p consecutive Ps in a pattern. We have found two general patterns that perform well for our case:

p In the p-pattern, the idea is to split the Qs into a number of blocks of length $b = \lfloor \frac{q}{p} \rfloor$ and interleave the blocks with a P in an alternating manner. Since this does not always work out perfectly, there is some rest $c = q - pb$ of Qs that are simply appended at the end of the pattern. The resulting pattern has the form $(Q^b P)^p Q^c$.

$p+1$ Mostly similar to the p-pattern, the $p+1$-pattern uses one additional block of Qs between the last P and the Q-tail. For certain r this can lead to lower average skew. For the $p+1$-pattern the block length is $b\lfloor \frac{q}{p+1} \rfloor$, leading to a rest of $c = q - (p+1)b$, and the actual pattern has the form $(Q^b P)^p Q^b Q^c$.

```
1: Initialize $Q(s,a)$ arbitrarily and $e_{s,a} = 0$, for all $s, a$
2: **loop**(for each episode):
3:     Initialize s, a
4:     **repeat**(for each step of episode):
5:         Take action $a$, observe $r, s'$
6:         $a' \leftarrow \pi_Q(s')$
7:         $\delta \leftarrow r + \gamma Q(s',a') - Q(s,a)$
8:         $e_{s,a} \leftarrow 1$
9:         **for all** actions $\hat{a} \ne a$ **do**
10:            $e_{s,\hat{a}} \leftarrow 0$
11:        **end for**
12:        **for all** $s, a$ **do**
13:            $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e_{s,a}$
14:            $e_{s,a} \leftarrow \gamma\lambda e_{s,a}$
15:        **end for**
16:        $s \leftarrow s'$; $a \leftarrow a'$
17:    **until** $s$ is terminal
18: **end loop**
```

Figure 3. Sarsa(λ) algorithm adapted from Sutton and Barto [13] (figure 7.11).

In general it is best to select the pattern with the smallest value for the rest c. Of course, one could always do better by spreading the c tail Qs more evenly over the sequence. However, more complicated patterns are more difficult to keep track of computationally. A possible solution might be to generate appropriate deterministic finite automaton (DFA) transition matrices on the fly, that represent a well spread pattern. But if values of r change rapidly, the effort might be wasted.

It can be seen in figure 1 that pattern selection improves significantly on the distribution of actual ratios compared to probabilistic selection, especially for the long episodes. For shorter sequences like the 16 messages on wire, there is still a large improvement, but there are also areas where it fails to cope well. For example at $r = \frac{3}{100}$ the pattern mainly consists of a long sequences Qs with the occasional P. Since the Q sequences are longer than the 16 messages on the wire, there is some significant skew there, that can not really be avoided. This also shows that it might not be worth setting the learner (s. section IV-C2) to a very fine resolution in terms of r as it might be impossible to accurately represent those ratios at meaningful timescales.

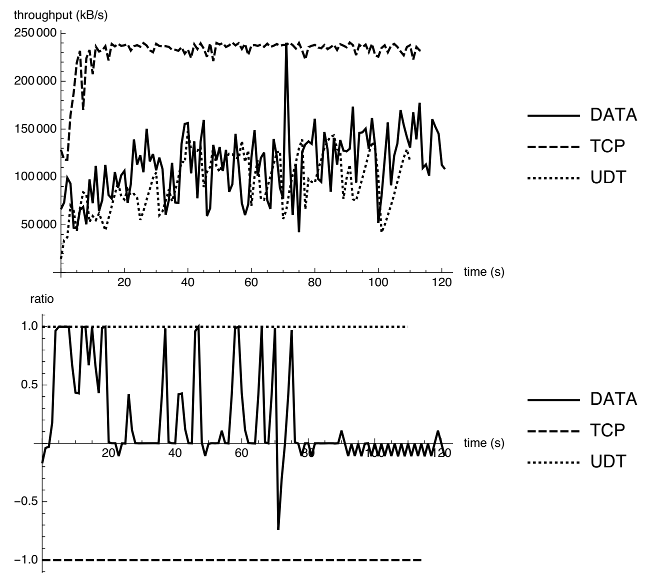

Figure 4. TD learner run with default matrix $Q(s,a)$ implementation, showing throughput and true protocol ratio (100% TCP ~ −1.0, 100% UDT ~ 1.0) with TCP and UDT as reference. For large state-action spaces the model converges too slowly to be useful.

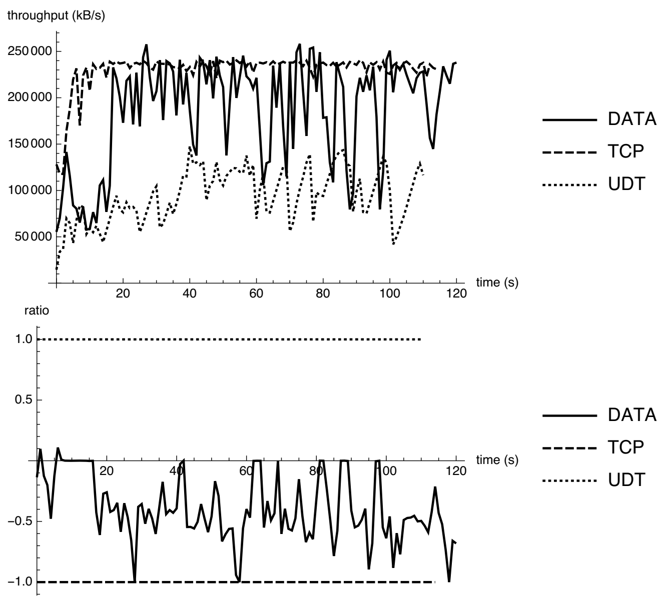

Figure 5. TD learner run with $Q(s,a)$ collapsed into a $V(s)$ and a model $M(s,a) \to s'$ implementation, showing throughput and true protocol ratio (100% TCP ~ −1.0, 100% UDT ~ 1.0) with TCP and UDT as reference. Using a model reduces the space that must be explored and improves convergence significantly.

C. Protocol Ratio Policies

1) Static: The simplest possible policy, setting the target ratio, e.g., TCP-only ($r = 0$), UDT-only ($r = 1$) or 50-50 ($r = \frac{1}{2}$), per configuration at system startup. This policy is mostly convenient for testing the PSPs and as reference for our experiments, and is implemented in `StaticRatio`.

2) Temporal Difference Learner: Implemented in `TDRatioLearner`, this TD(λ) reinforcement learning (cf. section II-C) based policy uses collected throughput and latency statistics as rewards to adjust the target ratio $r \in \mathbb{Q}$. We use an online on-policy Sarsa(λ) [13] control algorithm, depicted in figure 3, which learns an action-value function $Q(s,a)$ (for states s and actions a), and uses it to adapt an ϵ-greedy policy π_Q at every time step (or *episode*). The matrix e is called an *eligibility trace*, and describes to what degree previously visited states should benefit from the reward in the current time step. We use the so called *replacing trace* instead of the default *accumulating trace*, to avoid heavily visited state-action pairs to have unreasonably high eligibility, and thus receive an disproportional amount of reward. There is a number of parameters which have important impact on the behaviour of the algorithm:

- $\lambda \in [0.0, 1.0] \subset \mathbb{R}$ describes the decay of eligibility, where $\lambda = 0.0$ leads to complete exclusion of the eligibility trace (cf. one-step TD in [13]), and $\lambda = 1.0$ declares unbounded eligibility, and is equivalent to a Monte Carlo method.
- $\gamma \in [0.0, 1.0] \subset \mathbb{R}$ defines how far in the direction of the estimated value of the chosen state-action $Q(s',a')$ we want to shift eligible state-action entries in the estimator Q. When $\gamma = 0.0$ the expected value is ignored and only the current reward r and the previous estimate $Q(s,a)$ are taken into account. If $\gamma = 1.0$ the expected value has the same weight as the previous value.
- $\alpha \in \mathbb{R}$ is the step size for value estimate adjustments. If α is large, it can change estimates quickly, and possibly also balance out small eligibility (for example for $\alpha > 1.0$). If α is small, the learning progress can be very slow.

3) Matrix-based Value Functions for TD(λ): The default implementation without any model for $Q(s,a)$ in the discrete case is simply a large matrix with all state-action pairs. However, even for relatively small numbers of states and actions, it can take a very long time to fill out all the fields of the matrix so the policy π_Q can make greedy decisions at all (it makes a random decision if the value is uninitialised). This might lead to very long convergence times, or might prevent convergence altogether if ϵ decays too rapidly (cf. section II-C).
To make this clear, consider an example where we discretise the ratio space for r by defining a fixed step size $\kappa = \frac{1}{5}$. This is in fact what happens in our implementation as well, in order to interact well with the pattern-based PSP. To make things easier to see, we also shift the range of r from $[0.0, 1.0] \subset \mathbb{R}$ to $[-1.0, 1.0] \subset \mathbb{Q}$ such that -1 ~ 100% TCP, 1 ~ 100% UDT, and 0 ~ 50-50. With this setup we allow $2\frac{1}{\kappa} + 1$ states such that our state space is

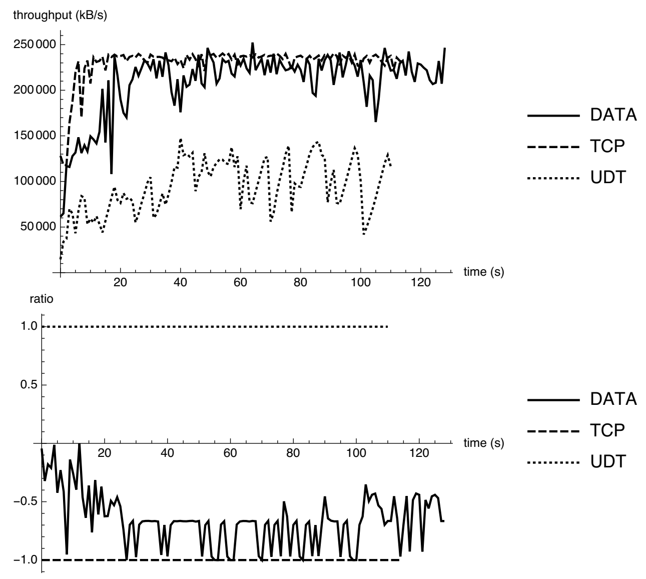

Figure 6. TD learner run with $Q(s,a)$ collapsed into a $V(s)$, which is approximated by a quadratic function, and a model $M(s,a) \rightarrow s'$ implementation, showing throughput and true protocol ratio (100% TCP ~ −1.0, 100% UDT ~ 1.0) with TCP and UDT as reference. Using the approximated values where no actual data is available, increases performance further, and avoid unnecessary backtracking by the policy.

$s \in \{-1, -\frac{4}{5}, \ldots, 0, \ldots, \frac{4}{5}, 1\}$. To make things interesting we allow not only the basic actions of taking one step or no step in either direction, but we allow two steps at once as well, giving an action space of $a \in \{-\frac{2}{5}, -\frac{1}{5}, 0, \frac{1}{5}, \frac{2}{5}\}$. Putting these spaces together gives an 11×5 matrix for $Q(s,a)$ with 55 different states. In figure 4 we can see how this implementation performs with TCP and UDT as reference, using parameters $\alpha = 0.5, \gamma = 0.5, \lambda = 0.85, \epsilon_{max} = 0.8, \epsilon_{min} = 0.1$, and ϵ-decay of $\Delta_\epsilon = 0.01$ per time step. Since TCP is very fast in this environment, the optimal behaviour would be to converge to r very close to −1. However, it can clearly be seen that this never happens in the example run, as, despite the very large initial ϵ_{max} value and relatively slow decay Δ_ϵ, after 120s the value space of $Q(s,a)$ is absolutely insufficiently explored, and with only a 10% chance for further exploration (ϵ_{min}) convergence is very unlikely at this point. While it is theoretically possible to get this model to converge with larger values of $\epsilon_{max} = 0.8$ and lower decay Δ_ϵ, waiting many minutes for the transfer to perform reasonably is simply not realistic, since many transfers might have finished by the time the algorithm converges.

4) Model-based Value Functions for TD(λ): To speed things up, we can use our domain knowledge about the problem and define a *model* $M(s,a)$, which maps a state s and an action a to the next state s' which would result from application of a to s. For most s, a this would simply be $M(s,a) = s + a$, however since we have a finite state space we have to remap the edges, such that we never leave it.

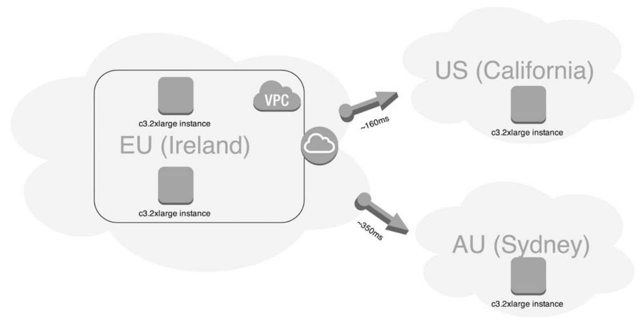

Figure 7. Amazon AWS deployment and connectivity of the experimental setup.

Thus, for example, $M(-1, -\frac{1}{5}) = -1$, and so on. Formally, for a state space S the model function M is defined as:

$$M(s,a) = \begin{cases} \min(s + a, \max(S)), & \text{for } s + a \geq 0 \\ \max(s + a, \min(S)), & \text{for } s + a < 0 \end{cases}$$

With this approach we can collapse the 11×5 matrix we used for $Q(s,a)$ into a state-value vector $V(s)$ with 11 states, and use M to calculate $Q(s,a) = V(M(s,a))$. Using this approach, we can run the same experiment as before, and, as can be seen in figure 5, it now converges to a reasonable value after around $20s$. We even used a lower initial value of $\epsilon_{max} = 0.3$, to avoid too much exploration after convergence.

5) Value Function Approximation for TD(λ): While $20s$ is significantly better than no convergence, on high bandwidth connections, e.g., 250MB/s, one could already have to transferred 5GB with the 'right' protocol provided by an *oracle*, before full performance is reached by the learner. In order to reduce convergence time further, we can make the following assumption about the reward function:
At any given time the reward function *for a given connection's protocol selection ratio has the shape of a* quadratic function *with a single maximum.*
Given this assumption, we can approximate the value estimator V using function approximation over the values we have seen, with a minimum of two, to extrapolate values in unexplored areas. In a sense, we are trying to exploit a *trend* we see in the data we have already collected. Note that we never use an approximated value if there is a learned value available. We simply fill the gaps earlier on, so we can make greedy decisions before the state space is properly explored. Using this approach, we can see in figure 6 that the learner already performs reasonably well after a few seconds, but more importantly it never backtracks significantly leading to much more consistent behaviour especially in the late states at low ϵ values.

V. Evaluation

In order to support our claims about the need for flexible transport protocols we executed a number of experiments

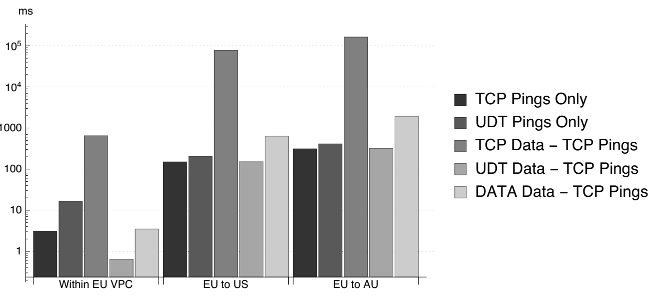

Figure 8. RTTs for simple "Ping" control messages over different distances and with and without parallel data transfer using different protocols.

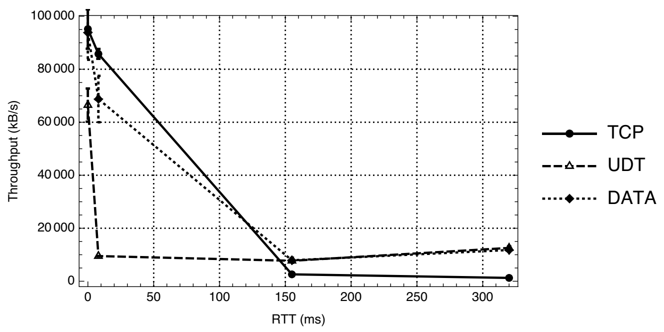

Figure 9. Data transfer throughput for different RTTs, over TCP, UDT, DATA (error bars show 95% confidence intervals). It can be seen that DATA behaves close to optimal at all RTTs.

using our KompicsMessaging implementation[2]. Specifically we wanted to answer two questions:

i) How does the link delay influence the choice of protocol for data transport.
ii) How does the choice of protocol influence the trade-off between throughput for data messages and latency for control messages.

A. Experiment Setup

On top of our KompicsMessaging implementation we developed a number of components for two purposes:

1) One set of components deals with transporting a large file from a sender to a receiver. Our implementation uses wrappers around `java.io.RandomAccessFile` to split the file into messages that would fit into our Netty channel serialisation buffers – we used 65kB buffer size in the experiments. We took great care in this to avoid duplicating data in memory or reading up any unnecessary data, as well as keeping the whole process as asynchronous as possible. Of course, writing into the file has to be synchronised. As a dataset to transfer we used a NetCDF [14] climate data file, which is around 395MB in size. A 395MB dataset provided us with a good tradeoff between getting the protocols ramped up even on fast links, while allowing us to run multiple iterations of the experiments even on the slowest links. It should be noted that, since our implementation has a *Snappy* compression handler in Netty's channel pipelines by default, the exact results might differ if the experiments are repeated with data than can easily be compressed.
2) The other set of components simulates timing sensitive control messages by sending "pings" from one host the other, with the other host replying with a "pong" and the sender measuring the round trip time (RTT).

For the experiments we also used a slightly modified Netty version 5.0.0Alpha3-SNAPSHOT, where we increased the default values for UDTs protocol buffer sizes, both send and receive buffers, from 12MB to 100MB. This was necessary as we discovered that on high BDP links the normal default values resulted in high packet loss rates on the receiver side. The overall scenario was then set up in such a way that we could pick the protocol for the data and the control messages separately and we could also decide whether to run only data transfer, only control messages or both at the same time.
As a testbed we used Amazon EC2 with pairs of c3.2xlarge instances, running Ubuntu 14.04 LTS AMIs with HVM virtualisation, in four different setups: The first setup (at 0ms RTT) copied on the same node from one SSD to the other using loopback interface, the second setup (EU-VPC) would have both instances within the same data centre, i.e. in Ireland, and even within the same Virtual Private Cloud (VPC). The third setup (EU2US) used one instance in Ireland and the other one in North California in the USA. The third setup (EU2AU) still had one instance in Ireland but the second instance was now in Sydney, Australia. These setups allowed us to test our system with RTTs between around $3ms$ and around $350ms$, as shown by the "TCP Pings Only" entries in figure 8. The first, local, setup simply measures disk throughput in the best case, or protocol or implementation buffer upper bounds in the worst case.

B. Throughput and Link Delay

We measured the throughput of our implementation by repeatedly sending the transfer dataset from the first instance to the second and measuring the disk-to-disk transfer time. For TCP, UDT, DATA (cf. section IV) transport we would do at least 10 runs, sometimes more until the relative standard error (RSE) dropped below 10% of the sample mean. The results are shown in figure 9 with the different setups represented by different RTTs. Local loopback is at $0ms$, EU-VPC would be at $3ms$, EU2US would be around $155ms$ and EU2AU would be around $320ms$. The error bars, where visible[3], show the 95% confidence interval for the sample mean.

[2]Code and Data can be found here: https://github.com/CaracalDB/ICDCS17

[3]For most data points, the errors are simply very small, as many of these tests have very consistent results.

It becomes immediately clear that transfer via TCP behaves as we had expected, with very good performance at low RTT but a sharp drop-off at higher values. In the local scenario, in fact, TCP and DATA are limited by disk performance, as memory to memory we reached even higher throughput of around 150MB/s. UDT seems to be limited by internal queue and buffer sizes. It can be noted that UDT shows consistent behaviour across all setups with real network. The reason is that Amazon artificially rate limits UDP traffic to around 10MB/s. While TCP vastly outperforms UDT within a VPC, at longer RTTs UDT is up to an order of magnitude faster. Our learner implementation shows the desired behaviour of following TCP closely, where TCP is strong and matching UDT, where it is strong, giving the best of both worlds, with the only drawbacks being somewhat higher variance and a certain ramp up time.

C. Message Latency

During this experiment series we ran both the data transfer and the control message components in parallel. We used the same statistics as in the transfer-only experiment series to determine the number of runs.
We tried four different protocol combinations: i) Both control messages and data message were sent over TCP, ii) control messages were sent over TCP and data messages were sent over UDT, and iii) control messages were sent over TCP and data messages were sent over DATA. In figure 8, which also includes results without parallel data transfer for comparison, we can see the influence of data transfer on control message latency. As expected the setup sending both message types over the same protocol (TCP) incurs a significant latency penalty – note the logarithmic scale. However, when sending data messages over UDT instead, the latency increase is barely noticeable, as the two protocols do not interfere as much. In the case where data messages use the DATA protocol, which is a mix of TCP and UDT, the result is in between the extremes. However, the RTT is still two orders of magnitude lower than sending both message type over TCP. The reason for the significantly better behaviour, are in the data transfer optimised internal queuing, the DATA protocol provides, which allows control messages to interleave better with data messages.

VI. Related Work

Protocols. The performance of different transport protocols has been well studied, especially relating to TCP [15], [16], and many alternative congestion control algorithms and protocols have been proposed. Among them are solutions that address high BDP links [17], [18], specifically satellite links [19], [20], data centre networks [21], [22], lossy links such as wireless connections [23], [24] and adaptive approaches [25]. However, changing TCP's congestion control algorithm typically requires modifcations to the operating system kernel and such access can not reasonably be expected in systems like internet-scale P2P deployments. There is also a number of alternative protocols with similar guarantees to TCP like Aspera FASP [8], the UDT [9], PCC [10], and LEDBAT [7]. And lastly it has been attempted to inverse multiplex data over multiple parallel TCP connections [26] or even multiple networks paths [27]–[29].
Middleware. There is a number of message-oriented middleware systems including Akka's remote [4], Websphere MQ [30], SIENA [31], and ZeroMQ [32]. Of those Akka is most similar to our work since it combines a message-passing progamming framework with a network messaging layer. However, while Akka does provide support for plugging in custom transport implementations, Actors within an ActorSystem are fairly inflexible when it comes to the choice of transport used for their messages. The cause of this is that Akka provides *location transparency* for its Actors, never exposing the circumstance that messages in fact may travel to another host. In our system on the other hand networking is always explicit to avoid unexpected behaviour caused by different channel semantics.
Even modern Actor-based programming models designed with specific support for multi-cloud environments, such as Microsoft Orleans [33], have not addressed the issue of mixed congestion control protocols, to our knowledge.
Automatic Protocol Selection. Wachs et al. [34] have investigated heuristic, linear programming, and reinforcement learning based solutions for selecting networking protocols automatically, with a focus on decentralised, peer-to-peer networks.

VII. Conclusions

In this paper we have presented a message-oriented middleware called *KompicsMessaging*, that provides per-message transport protocol selection, with the choice among UDP, TCP, and UDT and a meta-protocols called DATA, as well as a highly customisable API allowing for, e.g., virtual node architectures, or multi-hop forwarding systems to be built on top with minimal overhead. Additionally, we have presented an adaptive transport protocol selection mechanism employing online reinforcement learning techniques.
Our experiments clearly demonstrated the need for flexible and dynamic transport protocol selection for systems with broad application areas, and showed that our solution meets those requirements. The evaluation also showed that it is strictly desirable to separate the channels and protocols used for small, latency-sensitive control messages and large data messages which typically value high throughput over low latency, and that our DATA meta-protocol eases this separation with good results.

Acknowledgment

The authors would like to thank Professor Seif Haridi and Paris Carbone for their invaluable feedback and tireless proof reading.

References

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark : Cluster Computing with Working Sets," *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, p. 10, 2010.

[2] M. Zaharia, M. Chowdhury, T. Das, and A. Dave, "Resilient distributed datasets: A fault-tolerant abstraction for inmemory cluster computing," *Proceedings of the 9th …*, 2011. [Online]. Available: https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf

[3] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas, "Apache Flink: Unified Stream and Batch Processing in a Single Engine," *Data Engineering*, 2015.

[4] D. Wyatt, *Akka Concurrency*. USA: Artima Incorporation, 2013.

[5] C. Arad, J. Dowling, and S. Haridi, "Message-passing Concurrency for Scalable, Stateful, Reconfigurable Middleware," in *Proceedings of the 13th International Middleware Conference*, ser. Middleware '12. New York, NY, USA: Springer-Verlag New York, Inc., 2012, pp. 208–228. [Online]. Available: http://dl.acm.org/citation.cfm?id=2442626.2442640

[6] The Netty project, "http://netty.io/." [Online]. Available: http://netty.io/

[7] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind, "Low extra delay background transport (LEDBAT)," *IETF draft*, 2010.

[8] Aspera FASP, "http://asperasoft.com/technology/transport/fasp/."

[9] Y. Gu and R. L. Grossman, "UDT: UDP-based data transfer for high-speed wide area networks," *Computer Networks*, vol. 51, no. 7, pp. 1777–1799, 2007.

[10] M. Dong, Q. Li, D. Zarchy, B. Godfrey, and M. Schapira, "PCC: Re-architecting Congestion Control for Consistent High Performance," sep 2014. [Online]. Available: http://arxiv.org/abs/1409.7092

[11] C. C. I. Arad, "Programming Model and Protocols for Reconfigurable Distributed Systems," Ph.D. dissertation, KTH - Royal Institute of Technology, Stockholm, 2013. [Online]. Available: https://www.kth.se/social/upload/51b05a6af276541b8120ce4d/CosminArad-PhD-Defence.pdf

[12] J. Armstrong, "Making reliable distributed systems in the presence of software errors," no. December, p. 295, 2003. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.3.408{\&}rep=rep1{\&}type=pdf

[13] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 1998.

[14] Neale, R., "Coupled simulations from CESM1 using the Community Atmosphere Model version 5: (CAM5), see also http://www.cesm.ucar.edu/publications," 2012. [Online]. Available: http://www.cesm.ucar.edu/publications

[15] W.-c. Feng and P. Tinnakornsrisuphap, "The failure of TCP in high-performance computational grids," in *Supercomputing, ACM/IEEE 2000 Conference*. IEEE, 2000, p. 37.

[16] A. Chien, T. Faber, A. Falk, J. Bannister, R. Grossman, and J. Leigh, "Transport protocols for high performance: Whither TCP," *Communications of the ACM*, vol. 46, no. 11, pp. 42–49, 2003.

[17] S. Ha, I. Rhee, and L. Xu, "CUBIC: a new TCP-friendly high-speed TCP variant," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 64–74, 2008.

[18] D. X. Wei, C. Jin, S. H. Low, and S. Hegde, "FAST TCP: motivation, architecture, algorithms, performance," *IEEE/ACM Transactions on Networking (ToN)*, vol. 14, no. 6, pp. 1246–1259, 2006.

[19] H. Obata, K. Tamehiro, and K. Ishida, "Experimental evaluation of TCP-STAR for satellite Internet over WINDS," in *Autonomous Decentralized Systems (ISADS), 2011 10th International Symposium on*. IEEE, 2011, pp. 605–610.

[20] C. Caini and R. Firrincieli, "TCP Hybla: a TCP enhancement for heterogeneous networks," *International journal of satellite communications and networking*, vol. 22, no. 5, pp. 547–566, 2004.

[21] H. Wu, Z. Feng, C. Guo, and Y. Zhang, "ICTCP: incast congestion control for TCP in data-center networks," *IEEE/ACM Transactions on Networking (TON)*, vol. 21, no. 2, pp. 345–358, 2013.

[22] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," *ACM SIGCOMM computer communication review*, vol. 41, no. 4, pp. 63–74, 2011.

[23] S. Mascolo, C. Casetti, M. Gerla, M. Y. Sanadidi, and R. Wang, "TCP westwood: Bandwidth estimation for enhanced transport over wireless links," in *Proceedings of the 7th annual international conference on Mobile computing and networking*. ACM, 2001, pp. 287–297.

[24] S. Liu, T. Ba\csar, and R. Srikant, "TCP-Illinois: A loss-and delay-based congestion control algorithm for high-speed networks," *Performance Evaluation*, vol. 65, no. 6, pp. 417–440, 2008.

[25] K. Winstein and H. Balakrishnan, "Tcp ex machina: Computer-generated congestion control," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 123–134.

[26] A. Baldini, L. De Carli, and F. Risso, "Increasing performances of TCP data transfers through multiple parallel connections," in *Computers and Communications, 2009. ISCC 2009. IEEE Symposium on*. IEEE, 2009, pp. 630–636.

[27] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley, "Design, Implementation and Evaluation of Congestion Control for Multipath TCP." in *NSDI*, vol. 11, 2011, p. 8.

[28] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving datacenter performance and robustness with multipath tcp," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 266–277, 2011.

[29] A. Ford, C. Raiciu, M. Handley, S. Barre, J. Iyengar, and Others, “Architectural guidelines for multipath TCP development,” *IETF, Informational RFC*, vol. 6182, pp. 1721–2070, 2011.

[30] L. Gilman and R. Schreiber, *Distributed computing with IBM MQSeries*. John Wiley & Sons, Inc., 1996.

[31] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, “Achieving scalability and expressiveness in an internet-scale event notification service,” in *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*. ACM, 2000, pp. 219–227.

[32] P. Hintjens, *ZeroMQ: Messaging for Many Applications*. " O’Reilly Media, Inc.", 2013.

[33] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin, “Orleans: cloud computing for everyone,” in *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 2011, p. 16.

[34] M. Wachs, F. Oehlmann, and C. Grothoff, “Automatic transport selection and resource allocation for resilient communication in decentralised networks,” in *14-th IEEE International Conference on Peer-to-Peer Computing*. IEEE, sep 2014, pp. 1–5. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6934301