Unicast Multi-Ring Paxos

Implementation & Evaluation

Master's Thesis submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Master of Science in Informatics
Distributed Systems

presented by
Samuel Benz

under the supervision of
Prof. Fernando Pedone
co-supervised by
Daniele Sciascia

June 2013

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Samuel Benz
Lugano, 7. June 2013

The Paxos algorithm,
when presented in plain English,
is very simple.

Leslie Lamport

Abstract

The rise of internet-scale services demands large distributed systems. Such huge infrastructures have to be coordinated and, to provide fault-tolerance, replicated. The underlying algorithms that implement the required properties are far from trivial. Usually, developers rely on existing implementations of coordination frameworks such as atomic broadcast. But in order to allow the systems to grow, these frameworks must scale.

One approach to scaling such an atomic broadcast protocol is Multi-Ring Paxos. It uses multiple efficient rings of a Paxos implementation to provide scalable high throughput of totally ordered traffic. This thesis presents an implementation of Multi-Ring Paxos. Different from previous implementations, it contains all components needed to tolerate adverse conditions, such as message losses and process crashes. The completeness of the implemented algorithm parts makes the code generic for different usage scenarios.

The thesis also contains a detailed performance analysis conducted in different hardware environments.

Acknowledgements

I would like to thank Fernando Pedone, Daniele Sciascia and Parisa Jalili Marandi for all the interesting discussions about Multi-Ring Paxos and related topics. Further, Ricardo Padilha, for his insightful view of making Java fast. Simon Leinen and his team at SWITCH for giving me access to their cluster, and my family Nina and Mona for their patience.

Finally, I'd like to thank Georgios Kontoleon, the panter AG and Theodor Müller. Without them, my work at the university would not have been possible.

Contents

Contents **ix**

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**

2 Background **3**
2.1 Consensus and Atomic Broadcast 3
2.2 Paxos 5
2.3 Ring Paxos 6
2.4 Multi-Ring Paxos 7
2.5 Related work 8

3 Implementation **11**
3.1 Algorithm 11
3.1.1 Proposer 11
3.1.2 Coordinator 12
3.1.3 Acceptors 13
3.1.4 Learners 14
3.2 Ring Management 14
3.2.1 Abstract Role 15
3.2.2 RingManager 15
3.2.3 Failures and recovery 15
3.3 Network communication 17
3.3.1 Transport 17
3.3.2 Serialization 18
3.3.3 NetworkManager 20
3.4 Stable storage 21

4 Performance evaluation **23**
4.1 Experiments 23
4.2 Setup 24
4.2.1 Test client and data acquisition 24
4.2.2 Configuration 25
4.2.3 Infrastructure 26
4.3 Results 27
4.3.1 Ring Paxos performance 27
4.3.2 Multi-Ring Paxos performance 34

5 Conclusions **43**
5.1 Discussion 43
5.2 Future work 44

Bibliography **47**

Figures

2.1 Paxos Algorithm 5
2.2 Unicast Ring Paxos Algorithm 7
2.3 Multi-Ring Paxos Algorithm 9

3.1 Class diagram of the ring management 16
3.2 Class diagram of *Message* 18
3.3 Cumulative garbage collection time in seconds for protobuf (red) and direct serialization (blue) 19
3.4 Object creation rate in mbyte/s for protobuf (red) and direct serialization (blue) 20
3.5 Class diagram of the network management 21
3.6 Performance comparison of different *StableStorage* implementations. . . 22

4.1 Impact of the TCP buffer size (32k values) 28
4.2 Impact of the application value (command) size (2M TCP buffer) 30
4.3 Single ring performance USI 32
4.4 Single ring performance Switch 33
4.5 Impact of the ring size 35
4.6 Efficiency of Skip messages under different ring loads ($\lambda = 600$ / $\Delta_t = 100ms$) 37
4.7 Multiple rings at one learner performance (rings on different 16 nodes) 38
4.8 Multiple rings per node performance (all rings on 4 nodes) 39
4.9 Amazon EC2 Global Ring 40
4.10 Independent local ring performance compared to local rings plus an additional global ring which synchronizes all three regions (day 1). 41
4.11 Independent local ring performance compared to local rings plus an additional global ring which synchronizes all three regions (day 2). 42

Tables

4.1 Ring Paxos default configuration (per ring). 25
4.2 Multi-Ring Paxos default configuration (system-wide). 25

Chapter 1

Introduction

The rise of internet-scale services demands large distributed systems. To handle multiple millions of users, it is common to operate thousands of servers. This infrastructure has to be coordinated and also replicated. Replication is important to provide a fault-tolerant system or to hide latency in the user experience by distributing the service all over the world.

A lot of these large infrastructures are built nowadays on top of some coordination frameworks. Such frameworks usually solve distributed consensus by implementing an atomic broadcast protocol. Chubby [Bur06], for example, is a distributed locking service that is the basis for the Google File System (GFS) and the BigTable infrastructure. The implemented algorithm is Paxos. Ceph [WBM+06], a distributed file system, uses Paxos to provide a consistent cluster map to all participants. Zookeeper [HKJR10], a well-known service, turns a Paxos-like atomic broadcast protocol into an easy-to-use interface. It enables developers to implement distributed coordination, like leader election or mutual exclusion, with only a few lines of code.[1]

To enable systems to grow, the underlying coordination frameworks must scale. Zookeeper, for example, provides strong guaranties, but it would be too slow to implement an efficient replicated state-machine. State-machine replication, also known as active replication, is an approach to rendering data accessible on different machines consistent [Sch90]. It relies on atomic broadcast. The core idea is that not the results of an operation are replicated, but the execution is done by all replicas. Atomic broadcast guaranties total order and if all replicas start from the same state, applying deterministic commands in order, they will end up in the same state.

All of the introduced coordination services are based on Paxos. But implementing Paxos efficiently is challenging itself. One approach to making it fast is Ring Paxos. Ring

[1] The implementation of Multi-Ring Paxos in this thesis uses Zookeeper for the ring management.

Paxos is a high performance atomic broadcast protocol that combines several practical observations to improve the throughput of standard Paxos [MPSP10]. The current implementation can order traffic almost at network connection speed of 1 Gbit/s. This is remarkable, but Ring Paxos does not scale, that is, the ability to increase this throughput by adding additional hardware.

Multi-Ring Paxos turns Ring Paxos into a scalable protocol [MPP12]. It uses different Ring Paxos instances and aggregates their output in an ordered way. The existing implementation is based on IP multicast, which is unfortunately rarely used outside of a single data center. This restricts the operational use to specific environments. While Ring Paxos was already implemented using unicast TCP connections, nobody ever combined the scalable multi-ring approach with commonly used unicast connections.

This master thesis is about the implementation and evaluation of a unicast Multi-Ring Paxos algorithm. The contributions of this work are as follows. The code written within this thesis is a rather complete implementation of Paxos, Ring Paxos and Multi-Ring Paxos. It uses Zookeeper for leader election, automatic ring deployment and configuration management. Further it provides different storage back-ends to make the system fault-tolerant. The code base, which is written in Java, is simple to understand and extensible. A ready-to-use cross-language RPC interface is also provided.[2]

The completeness of the algorithm parts makes the implementation generic for different usage scenarios. Not only the existing experiments could be repeated, new untested behaviors could be evaluated. For example the impact of high-latency rings on fast local rings in a cloud deployment on Amazon Elastic Compute Cloud (EC2).[3] This allows a better understanding of the algorithm itself.

The remainder of this thesis is structured as follows. Chapter 2 provides some theoretical background on topics related to the thesis and introduces the used algorithms. Chapter 3 focuses on the actual protocol implementation. Chapter 4 evaluates the work. Chapter 5 concludes this thesis and points to future work.

[2] `http://thrift.apache.org/`

[3] `http://aws.amazon.com/ec2/`

Chapter 2

Background

This chapter will furnish some theoretical background information on topics related to the thesis and will introduce the used algorithms. Finally it will present some related work.

2.1 Consensus and Atomic Broadcast

A fundamental problem in distributed systems is reaching consensus among multiple processes [Fis83]. As a simple example, consider two processes which should conclude to the same value. In a crash-stop failure model consensus is defined as follows [CT96]:

Termination: Every correct process eventually decides some value.

Agreement: No two correct processes decide differently.

Uniform integrity: Every process decides at most once.

Uniform validity: If a process decides v, then v was proposed by some process.

In a crash-stop model, a process is either correct or faulty. Further, it fails by halting. Once it halts, the process remains in that state. The fact that a process has failed may not be detectable by other processes [Fis83].

The consensus problem is notoriously difficult to solve in the presence of process failures and message losses. How can process a be sure that process b has decided on the same value? In a synchronous system, in which we have the notion of time, the first process can wait on a response or a timeout and proceed based on whatever happens first. A crash of a process can be detected in a synchronous system. But in an asynchronous system there is no notion of time. A famous result in the distributed systems theory states that no deterministic algorithm can solve consensus in an asynchronous system despite a single crash [FLP85].

Tolerating crashes while using asynchronous systems is exactly what we want in practice. One reason to build distributed systems is that we can tolerate failures. Since the synchronous model only shifts the problem and failures in the asynchronous model are not acceptable to reach consensus, we have either to weaken the problem or strengthen the model assumptions. By weakening the problem, we could for example tolerate at most k different values (*k-agreement*). Another solution to prevent processes in the asynchronous system from not making progress is to use failure detectors.

[CT96] propose a class of algorithms which use failure detectors to solve consensus. Further, they implement atomic broadcast. Atomic broadcast has similar properties to consensus:

Validity: If a correct process AB-broadcasts a message m, then it eventually AB-delivers m.

Agreement: If a correct process AB-delivers a message m, then all correct processes eventually AB-deliver m.

Uniform integrity: For any message m, every process AB-delivers m at most once, and only if m was previously AB-broadcast by sender(m).

Total order: If two correct processes p and q deliver two messages m and m', then p delivers m before m' if and only if q delivers m before m'.

In fact, it turns out that atomic broadcast can be reduced to consensus and vice versa. To achieve consensus in a distributed system, we can simply atomic broadcast a value. Since we have total order, all clients can decide on the first received value. On the other side, we can run a consensus protocol to decide on multiple independent instances of consensus. This sequence of consensus instances can be used to implement atomic broadcast. One consequence of the reduction of atomic broadcast to consensus is that atomic broadcast is not solvable in an asynchronous system in the presence of process crashes.

Despite the difficulties to build consensus or atomic broadcast protocols, they are very important in practice, since the communication paradigm they provide is very powerful. For example, a lot of protocols require a leader, a master process which coordinates the protocol. Once we have leader election, the protocol implementations are trivial. But leader election is only a simple consensus between all processes, which results in one atomic broadcast round. In general, atomic broadcast can be used for many kinds of distributed coordination of services, like mutual exclusion.

Another example where atomic broadcast protocols are required is state-machine replication [Sch90]. In this form of replication, the commands are sent through atomic broadcast to a set of replicas. Every replica executes the deterministic command in the same order. This results in the same state at every replica.

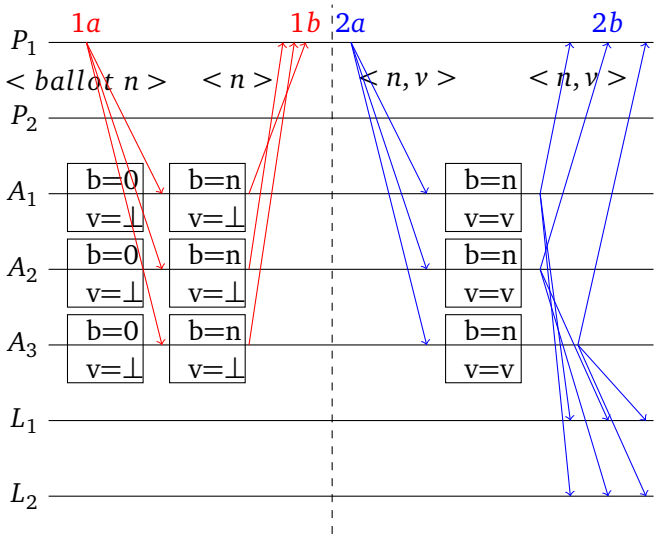

Figure 2.1. Paxos Algorithm

2.2 Paxos

Paxos is a distributed and failure-tolerant consensus protocol. It was proposed by Lamport [Lam98],[Lam01] and combines many properties which are required in practice. While Paxos operates in an asynchronous model and over unreliable channels, it can tolerate crash failures. By using stable storage, it can even recover from failures. To guarantee progress, Paxos assumes a leader-election oracle.

The protocol distinguishes three roles: proposers, acceptors and learners. The algorithm works as follows (Figure 2.1): In phase $1a$ a proposer sends a message with a unique number to all acceptors. If the acceptors never saw a higher number for this consensus instance, they confirm the reservation of the ballot by sending back a phase $1b$ message. If the proposer receives at least a quorum $\lceil (n+1)/2 \rceil$ of acceptor answers, it can start with phase $2a$. To get a quorum, a majority of acceptors must be alive. This means that Paxos requires $2f + 1$ acceptor nodes to tolerate up to f failures.

Phase $2a$ starts with a message, including the value to be proposed and the ballot number, from the proposer to all acceptors. If the ballot in the message corresponds to what the acceptors in phase 1 promised to accept, they will store the value. All acceptors will propagate their decision with a phase $2b$ message.

This is of course the most trivial case, in which no acceptor crashes and multiple proposers do not try to reserve the same consensus instance. Both of these scenarios will not affect the safety of the protocol. The later, however, could cause liveness problems. A liveness problem can prevent the algorithm from making progress, which would violate the termination property of the consensus definition. Such a scenario

can happen when P_1 receives a phase $1b$ message but before the acceptors receive its phase $2a$ message, a second proposer P_2 already increased the ballot with another $1a$ message. P_1 will re-try after a timeout while waiting for a $2b$ message and again send a message $1a$ with increased ballot. With unlucky timing, this can go on forever. Paxos solves the problem by assuming a leader election oracle, which selects one proposer only to execute phase 1.

The above Paxos algorithm solves only consensus for one instance. To use it as an atomic broadcast protocol, Paxos must be extended to run consensus for different incrementing instances.

2.3 Ring Paxos

While Paxos brings already a lot of interesting features in its original form, it is not very efficient. Ring Paxos [MPSP10] is a derivation of Paxos. It relies on the same safety and liveness properties as Paxos, but optimizes throughput and latency. The new algorithm is based on a few observations in practice.

1. The throughput of IP multicast scales constantly with the number of receivers, while IP unicast decreases proportional to the number of receivers.
2. Minimizing packet loss by limiting the number of IP multicast senders to one.
3. Limiting the number of incoming connections per host to one is more efficient than having many.

Concluding all of these observations, Ring Paxos has one coordinator which is also an acceptor and the multicast sender. Proposers send the values to this coordinator. Optimized phase 1 and phase 2 are executed in a ring of the acceptors. The decisions are multicast to all nodes.

While Ring Paxos can reach almost nominal bandwidth of 1 Gbit/s with a good average latency of 5ms [MPSP10], it depends on IP multicast. In some environments (e.g., wide-area networks), however, IP multicast is not available. To overcome this shortcoming, multicast can be replaced by pipelined unicast connections. Unicast pipelining almost achieves the same throughput as multicast, and may introduce delays, a price which has to be payed to port Ring Paxos to WAN links.

The Ring Paxos algorithm implemented in this thesis is based on unicast connections only. In this case, all nodes form a ring, not only the acceptors (Figure 2.2). Hereafter we call this protocol U-Ring Paxos.

While Paxos uses two different messages per phase (a/b), U-Ring Paxos uses several, for ring usage optimized, messages. Phase 1 has only one message with a vote count.

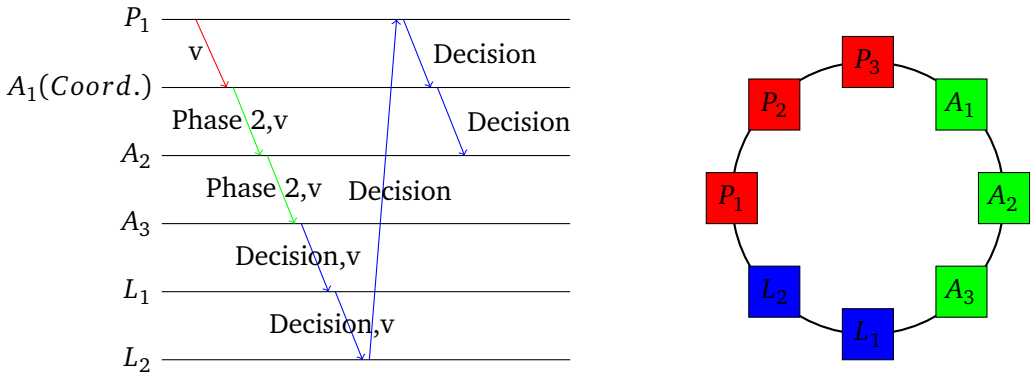

Figure 2.2. Unicast Ring Paxos Algorithm

Phase 2 has a message with a vote count and additionally a separate decision message. A proposer starts sending a message v to its ring successor. This node will store the value and forward the message until it reaches the coordinator. Where the roles are placed in the ring is not important for correctness, but it has an impact on the overall latency.

Once the coordinator receives a value, since it is also an acceptor, it starts learning the value with a phase 2 message. Phase 1 is not shown in the figure since this can be done before and for multiple instances.

When an acceptor receives a phase 2 message, it will increase the vote count in the message and store the value. At this point, the value is not yet decided. The decision message is issued by the last acceptor in the ring, if the vote count is larger than or equal to the quorum. The decision messages are forwarded in the ring until they reach the predecessor of the last acceptor. Now, the acceptor can persist the previously stored value on stable storage.

Phase 2 and the decision message do not always include the full value. The algorithm ensures that every value is only transmitted once in the ring. This is possible because the value contains a unique identifier and an actual value. The later can be removed when not needed before forwarding in the ring.

2.4 Multi-Ring Paxos

Ring Paxos solves the shortcoming of Paxos by making it fast in terms of throughput and latency. However, the resulting protocol is not scalable. Scalability is the ability to increase the overall performance by adding more resources. Ring Paxos is not scalable since all traffic must be submitted to all acceptors. Thus adding more acceptors, doesn't allow more messages to be ordered.

Multi-Ring Paxos [MPP12] however, is designed for scalability. The core idea behind

it is simple. By using multiple rings, the single-ring performance can be summed up. To still guarantee total order, Multi-Ring Paxos uses a deterministic merge function to combine the output of multiple rings.

The merge function can be a simple round-robin procedure (Figure 2.3). First take m values from the first ring, than m values from the second ring and so on. This assumes that all rings make progress at the same speed. If this is not the case, then sooner or later, some of the learners in the faster rings will wait until the slower rings deliver enough values. To overcome this problem, the coordinator of every ring keeps track of its ring throughput. The maximum throughput of the fastest ring in the system is a configuration parameter (λ). Each coordinator compares its actual throughput with λ and issues enough skip messages every time interval Δt to match λ.

A skip message is a special *null* value, which means that the multi-ring learner can skip one value since there are not enough data to propose in this ring. Several skip messages are batched and learned in one single Paxos instance. This keeps the overhead of skip messages minimal.

With this approach, it is also possible to combine rings with different throughput. While one goal could be to scale local disk writes, equally distributed ring throughput is expected; another goal could be to combine different WAN links. In the case of WAN links, fast local rings could be connected to slower but globally connecting rings.

2.5 Related work

Paxos is not the only protocol that can be used to implement atomic broadcast. Since its implementation is not trivial, there might be simpler ones. Paxos has some tricky cases which should be addressed during implementation [CGR07].

A good overview of other total order broadcast and multicast algorithms is provided in [DSU04]. This survey identifies five categories of protocols. The fixed sequencers, where one process is elected for ordering, and the moving sequencer, which balances the work by transferring the sequencer role across different processes. Further there are privilege-based protocols where the senders can only propose values when they hold a token. The category of communication history-based protocols are like the privilege-based protocols coordinated by the senders. In the case of communication history, all processes can send in parallel. The ordering is achieved by using timestamps, like vector clocks [Lam78]. An example of this category is LCR [GLPQ10]. LCR can achieve throughput similar to Ring Paxos, but requires a perfect failure detector. The last category is destinations agreement. As the name suggests, the ordering is achieved at the destinations. In case of Paxos, at the acceptors.

Multi-Ring Paxos is more than only gluing different atomic broadcast streams together. In fact, it defines a form of atomic multicast. However, this definition is not equal to

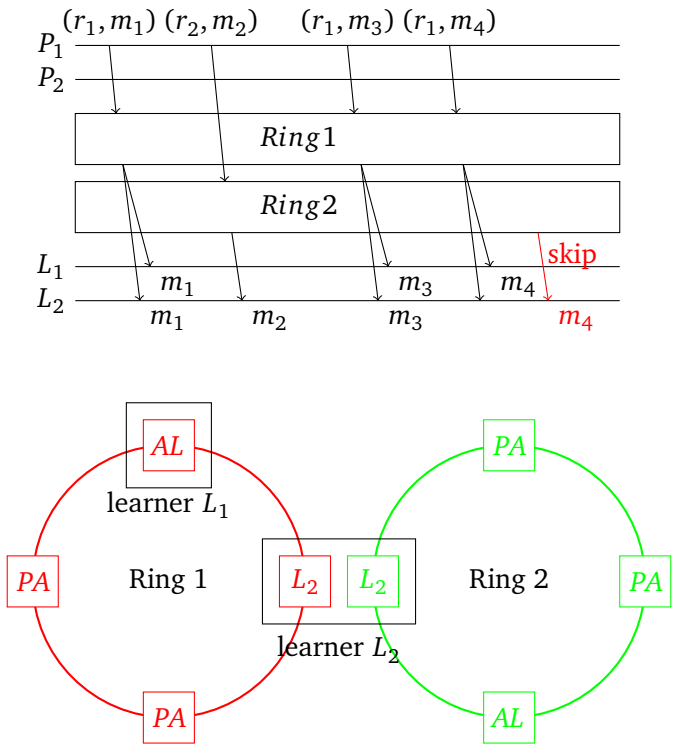

Figure 2.3. Multi-Ring Paxos Algorithm

the normal one of atomic multicast.

In normal atomic multicast, a process belongs to one group, while messages can be sent to different groups. Multi-Ring Paxos uses groups differently. A group can be seen as a ring and learners can subscribe to multiple rings. Proposers can only propose a message to one group.

Multi-Ring Paxos is a non-genuine atomic multicast protocol since processes not addressed by a message m, may need to interact in order for m to be delivered (e.g., by sending skip messages). An example of a genuine atomic multicast protocol is Skeen's [BJ87] algorithm, but Skeen's algorithm does not tolerate failures.

Multi-Ring Paxos combines several approaches also seen in other protocols. For example, Mencius [MJM08] similarly to Multi-Ring Paxos, uses skip instances to allow a mixed operation of fast and slow connections. Mencius, however, does not have the idea of groups. Multi-Ring Paxos deterministic merge function is similar to [AS00].

Chapter 3

Implementation

The complete algorithm presented in this work is written in Java. Some performance critical parts, however, are implemented in C, using the Java native interface (JNI). The decision to use Java instead of implementing everything in C was mainly due to the better code readability and maintainability of Java. While C is fast, the benefits that Java brings, for example, in the collections and concurrency frameworks, outweigh the small performance penalty we have to pay.

The following sections describe our prototype. The source code is available on Github[1] and has approximately 5030 (92.36%) Java, 296 (5.44%) Xml, 68 (1.25%) Shell and 52 (0.95%) lines C code.

3.1 Algorithm

In the context of this thesis, two Paxos algorithms were implemented: one based on Paxos [Lam01] using IP multicast and another based on Ring Paxos [MPSP10] using unicast transport. The multicast implementation was mainly targeted to understand the protocol. In the following, I will start describing Ring Paxos.

3.1.1 Proposer

A proposer is one of the simplest roles in the Paxos algorithm. In its original form, it can propose a command (value) by executing phase 1 followed by phase 2. As soon as we have more than one proposer in the system, this could cause liveness problems in the algorithm. An optimized proposer will send the commands to a coordinator (i.e., leader).

While in Paxos the values can be sent through multicast to the coordinator, in Ring

[1] `https://github.com/sambenz/URingPaxos`

Paxos, the commands are forwarded along the ring until they reach the leader (Figure 2.2). Ring Paxos introduces a new message type for this purpose.

Proposers are usually also learners. This is required to detect if an actual command has really got learned or in the case of a test client, to throttle down the generated test load. While in Ring Paxos all decisions are forwarded to all participants, proposers have to subscribe to the multicast group of the learners to receive phase 2b messages.

In this Ring Paxos implementation a proposer has two operation modes. An internal one which reads commands from standard input and an external one which can be used to directly embed the proposer in an application. Embedded values can be proposed using the following *propose()* method:

```
public FutureDecision propose(byte[] b);
```

The argument is a byte array. This command or value is wrapped in a *Value* object. A *Value* object has the actual byte array and a unique identifier (ID), which is generated in the proposer. This ID enables indirect consensus [ES06] and opens the possibility to remove the byte array from the message. For performance reasons, this ID is not a real UUID [LMS05] but it is a combination of time in nano seconds and the ring position of the proposer.

The return type of *propose()* is a *FutureDecision*. A future decision will contain eventually a *Decision*, once the byte array is learned. It contains also a *CountDownLatch* on which the calling thread can wait until the decision is set. By waiting on this lock, the caller can implement a blocking call, while the overall nature of propose is still non-blocking. This feature is used to measure the latency of a proposed value.

3.1.2 Coordinator

The coordinator is the process that starts phase 1. It is elected out of the group of acceptors. While the algorithm can tolerate multiple coordinators and still guarantee safety, it is more efficient when a single leader exists. Coordinator election is done with Zookeeper. The first acceptor in the ring acts as the coordinator.

After a coordinator is elected, it starts a new thread which is responsible for phase 1. While Paxos has a dedicated phase 1a/b message, Ring Paxos uses only one message with an additional vote count (Figure 3.2). Whenever a coordinator successfully reserves a ballot for an instance, it generates a *Promise* object which contains an instance number and a corresponding ballot and stores it in a *BlockingQueue*. While the instance numbers are continuously increasing, the ballot numbers are composed out of a counter and the last digit of the coordinator ID. This guarantees that ballot numbers are always unique in the whole system. If the proposed ballot for an instance is smaller than what the acceptors already promised to accept, then they will answer with a nack message or by sending nothing (timeout at coordinator). If the instance

was already decided, the acceptor will respond with *Value*. In this case, the coordinator must re-propose the *Value* with a higher ballot by starting phase 2.

The promise queue is not bounded to a fixed value of instances. A thread ensures that at any time the *Promise*s in the queue are more than half of the $p1\_preexecution\_number$. The reservation of multiple instances can be done in one message. This implementation, however, is conservative: Only when the smallest instance number in such a range message is higher than the highest instance the acceptors have ever seen, the range message is accepted. Otherwise, Ring Paxos falls back to a standard phase 1 messages for every instance.

When a coordinator receives a *Value* to propose, it starts executing phase 2. First it takes the next *Promise* from the queue and generates a *Proposal*. A *Proposal* is mainly a *Value* and a timestamp. The later is used to detect timeouts while proposing. Phase 2 is started by composing the *Promise* and the *Proposal* into a phase 2 message. Further, the coordinator keeps track of what is decided to remove the *Proposal* from an internal map or re-propose the same *Value* with a new *Promise*.

In the case of Multi-Ring Paxos, the coordinator has also the additional task to measure the ring throughput. The throughput per time interval is compared to a global maximum of λ messages. Whenever the actual throughput is smaller than λ, additional skip messages will be sent. Skip messages are special values, but normal Paxos instances. They are proposed and learned like every other value in Paxos, but the content is evaluated in the *MultiRingLeaner*s.

3.1.3 Acceptors

The most complex part of Paxos is implemented in the acceptors. Every acceptor has to keep track of which ballot it has promised to accept. Further, once a value is learned, it creates a *Decision* object that contains the instance, the ballot and the *Value*. This *Decision* is persisted over the *StableStorage* interface (Section 3.4). *Decision*s can be updated, but only the ballot. Once a *Value* is learned, it can't be changed anymore.

A Ring Paxos acceptor uses a hash map to keep track of what ballot it has promised to accept in which instance. Further, it uses another hash map to cache the byte arrays from the *Value*s and the *StableStorage* interface for all *Decision*s. Instead of sending back a 2b message to the proposer, a Ring Paxos acceptor increases the vote count in the message. The last acceptor in the ring checks if the vote count is larger than or equal to a predefined quorum. If so, it will generate a decision message which is forwarded along the ring up to the predecessor of the last acceptor.

Another optimization in Ring Paxos is that every byte array of a *Value* only gets transmitted once over the network. Depending on which *Role* a *Node* in the ring has and at which position it is located, a phase 2 or decision message may not contain the content (byte array). The *Value* object with the ID, however, is always present and can be used to store and look-up the content from a map.

3.1.4 Learners

Learners in the Ring Paxos implementation are interested in the decision messages. Like the acceptors, also the learners have to cache the content of the *Value*s, since the decision messages not always contain them.

While a simplistic learner delivers the values at the receipt, we usually want the learner to deliver the values in the order of the increasing instances instead. But this could block a learner in the case of an outstanding or missing instance. For that reason, a learner must also implement a proposer part to "ask" for such missing instances. In this implementation, a learner proposes a *null* value for an outstanding instance by starting phase 2 with a very high ballot number. This approach, described by [Lam01], will decide the instance to *null* if it was undecided or return the previously learned value.

Since no check-pointing is implemented, a learner that starts after instances have been decided, for example after recovery from a crash, will always start learning from instance 1.

Like the proposers, the learners can also run in a "service mode". A learner service will not write the decisions to the logging mechanism but rather store them in a *BlockingLinkedList* for further use. One application of this mechanism is a *MultiRingLeaner*.

A *MultiRingLeaner* is started if the *Node* is a learner in multiple rings. It is a wrapper around several independent learners and delivers *m* messages from every ring in a deterministic round-robin procedure. If one ring has no value to deliver, the multi-ring learner blocks on the *take()* method until some data are available in this ring. To guarantee the progress even in the absence of traffic in one ring, the Multi-Ring Paxos algorithm introduces the concept of skip messages.

Skip messages are issued by the ring coordinator. As already mentioned, they are proposed and learned in normal Paxos instances. The values that these instances include, however, have the static ID of "Skip!". Whenever a multi-ring learner gets such a skip message from a queue, it will not deliver this instance, but interpret the content of the value to figure out how many values it must skip. As described in Figure 2.3, these skip messages are not delivered but used to deliver values from all rings in a constant speed.

3.2 Ring Management

The heart of Ring Paxos is the ring management. It is responsible for creating the network connections, looking up configurations and providing a dynamic view of the current ring. The main entity is a *Node*. A *Node* is the class that implements *main()*. It is started with several command-line arguments to define a *List* of *RingDescription*s.

3.2.1 Abstract Role

For every *RingDescription*, the *Node* initializes a *RingManager*, which holds a Zookeeper connection. Further, it starts the threads for all Paxos roles described in the previous section, and registers them in the ring, using the *RingManager*. Every role has a concrete implementation which extends the abstract class *Role*:

```
public abstract class Role implements Runnable {
  public void deliver(RingManager fromRing, Message m);
}
```

The *RingManager* is passed to the constructor of every *Role*. This allows them to look up the responsible *NetworkManager* and register themselves for message delivery. The network will invoke the *deliver()* callback for the messages targeted to the specific role.

3.2.2 RingManager

The *RingManager* is the central component of the system. It spawns the *NetworkManager* to automatically open and close the required network connections to the ring successor, and is always informed about all changes on the ring. Moreover, it is available to all *Role*s and it is responsible for starting and stopping the coordinator process. Figure 3.1 shows the class diagram with these dependencies.

During the initialization, the *RingManger* registers the node ID in the given ring on Zookeeper. The *RingManager* is the only component in the system that interacts with Zookeeper. It will also publish its IP address and its randomly chosen TCP port as additional data with the ID. Further, it looks up the configurations map, also stored in the Zookeeper directory. After the initialization, the *RingManager* will have learned the ring topology and it will automatically open a network connection to the ring successor. The ring is monitored by a Zookeeper *Watcher*. Whenever the successor changes, the old network connection is closed and the new one will be opened. If a node in the ring crashes unexpectedly, Zookeeper will detect this and inform its manager after a timeout of some seconds.

Coordinator election is done in the same *Watcher*. The coordinator is the acceptor that has the lowest ID. The required thread with the corresponding role is also started by the *RingManager*.

Every node in the ring can look up the IP address and port of every other node with Zookeeper. The nodes get the addresses by parsing the local network interface configurations. For the Amazon EC2 deployment the IP is taken from an environment variable.

3.2.3 Failures and recovery

Nodes that fail by crashing are detected by the Zookeeper instance and removed from the ring. The changes are propagated to all other nodes. If required, the network

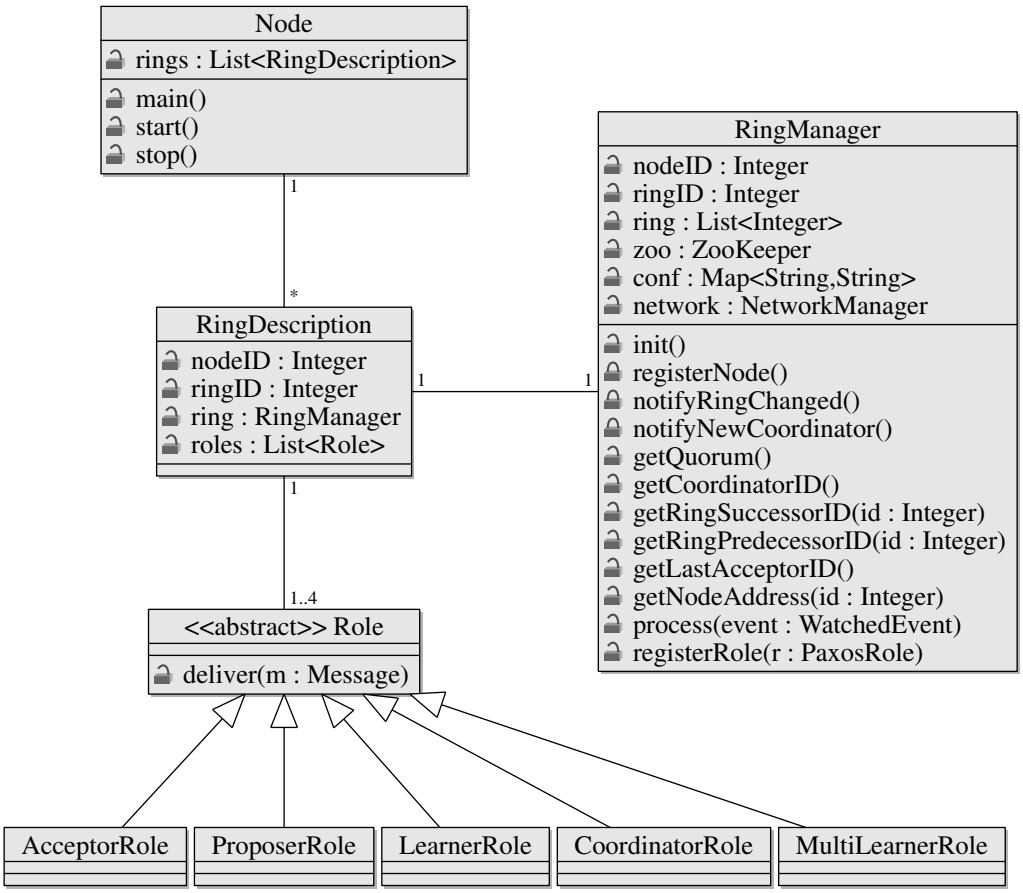

Figure 3.1. Class diagram of the ring management

connections are rebuilt. If the failing node was the coordinator, a new coordinator will be elected and started.

The same is true for process recovery. Whenever a new node joins the ring, its ID defines the position in the ring. The new node will automatically be included by the existing nodes.

While the *RingManager* is only responsible for the ring management, the roles must implement the required recovery methods. Acceptors are able to recover correctly if they use stable storage (Section 3.4). Learners can also be configured to recover. If so, they will start learning beginning from instance 1.

3.3 Network communication

Networking is a very important component in the implementation of this project since the goal of Ring Paxos and Multi-Ring Paxos is to achieve maximum throughput and low latency. Such requirements are hard to achieve by using existing frameworks (e.g., RPC stacks). They usually over-abstract the important features to make them simple. If we want to get the maximum out of the underlying hardware, careful tweaking of all possible parameters is required.

3.3.1 Transport

One of the goals of this thesis is the implementation of a unicast Multi-Ring Paxos. With this version, which doesn't use IP multicast, it should be possible to experiment on WAN links. For the transport layer we want to have a reliable message oriented protocol such as the Stream Control Transmission Protocol (SCTP) [Ste07]. This relatively new protocol combines the message-oriented principles from UDP with the reliability and congestion control of TCP. Since this protocol seems to be perfect for this purpose, the first implementation was built on top of it. The throughput test on the cluster has shown that SCTP is significantly slower than TCP. For this reason the implementation switched entirely to TCP.[2] The reason why SCTP is slow is not clear, one reason could be that it has not been as much optimized as TCP. TCP however is being optimized for more than twenty years.

TCP is stream oriented but Paxos is entirely message based. The use of TCP implies implementing our own message framing. In the current version, a length-prefix framing is used. This prepends the length in bytes of the following message (frame). The receiver will read the first two bytes interpreted as length, following the length of the message itself. While this approach is very simple and works well in practice, it has the drawback that once a receive buffer is out of synchronization, it will not find the frame borders anymore. Such a problem could happen when an unhandled exception

[2] SCTP Code in branch: "sctp"

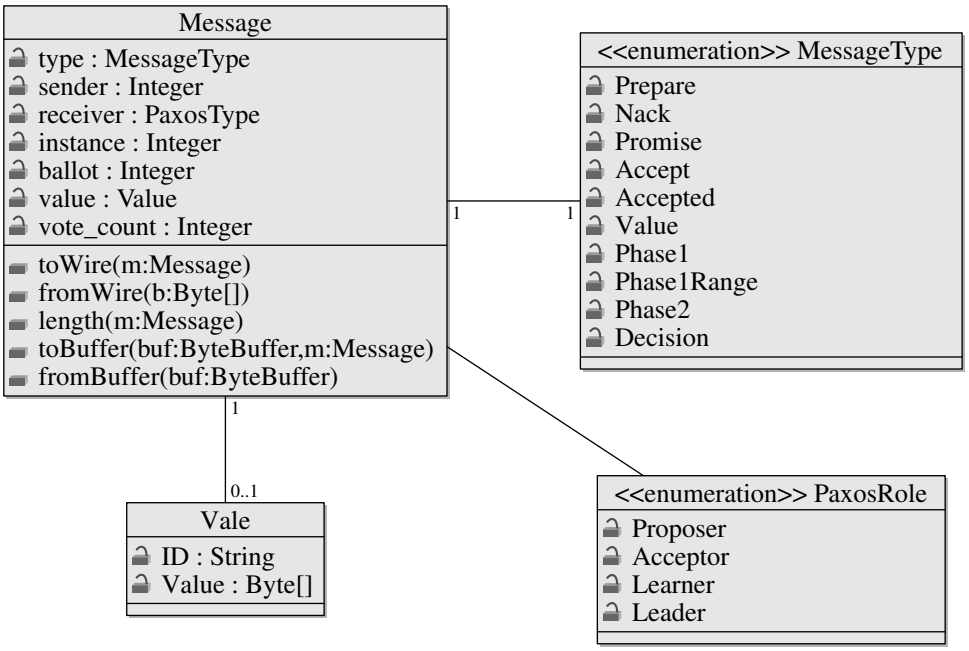

Figure 3.2. Class diagram of *Message*

is thrown in the server. (Observed rarely during experiments, while testing automatically ring joining/leaving under full load). A future release should also include a magic number in front of a package to allow a receive buffer to re-synchronize.

The current transport layer lets itself configure the TCP nodelay option (Nagle's algorithm) and the TCP send and receive windows. The impact of them are evaluated in Chapter 4.

While the first implementation of TCP uses standard (blocking) Java IO, the current implementation uses the non-blocking new Java IO (NIO). The reason to switch was not the demand for a scalable non-blocking API. But the NIO directly exposes DMA mapped *ByteBuffer*s instead of simple byte arrays, like in standard IO. This reduces the number of copies involved in transferring data, which improves speed.

3.3.2 Serialization

A critical piece to the overall protocol performance is the efficiency of serialization. Serialization is the conversion of a high-level abstraction of a message to its byte representation on the network line interface. While in C, de-serializing bytes received over the network is a simply cast to the corresponding message struct, in Java we have to re-create all required objects. It was an early design decision that the Java code will always operate on well-defined objects and never directly with byte arrays.

The classes which are responsible for messaging are defined in Figure 3.2. The

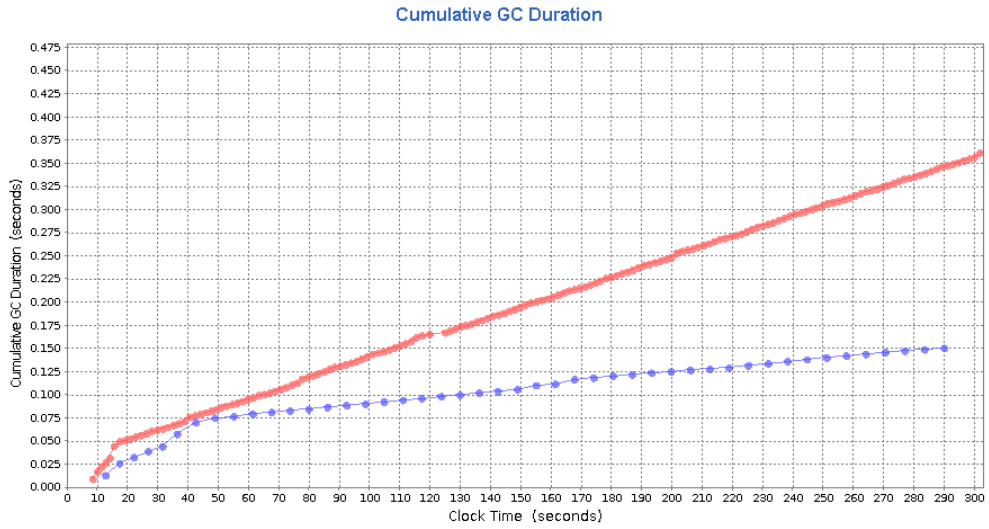

Figure 3.3. Cumulative garbage collection time in seconds for protobuf (red) and direct serialization (blue)

actual serialization part are the static methods *fromWire()* and *toWire()* in the *Message* class. The implementation of these two methods in version 1 was done with Java object serialization (*ObjectInputStream*). This proved too slow for a high speed implementation.

The next, very modular and interoperable approach, was done using Google's protobuf[3] protocol buffers. Protobuf is a framework for message serialization. It contains an interface-definition language and a compiler. The compiler, which is available for different programming languages, will generate the required bindings.

A long test run showed that the protocol was not as fast as expected, even after several modifications everywhere in the code were conduced. Finally, I decided to remove protobuf and use my own object structure for the messages, which does not copy the objects before serialization. This re-copying of the data caused before a very high object creation rate which results in a increased cumulative garbage collection time which is shown in Figure 3.3. Figure 3.4 shows the protobuf implementation in red, which allocates almost 400 MByte/s objects. Since the line speed is 1 Gbit/s, an optimal allocation rate should be around 125 MByte/s. This was achieved with a direct serialization (blue line). Direct serialization is implemented by copying every field of the *Message* object as byte to a *ByteBuffer*. This approach, together with the directly mapped network *ByteBuffer*s from Java NIO, resulted in an reasonable object creation rate.

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

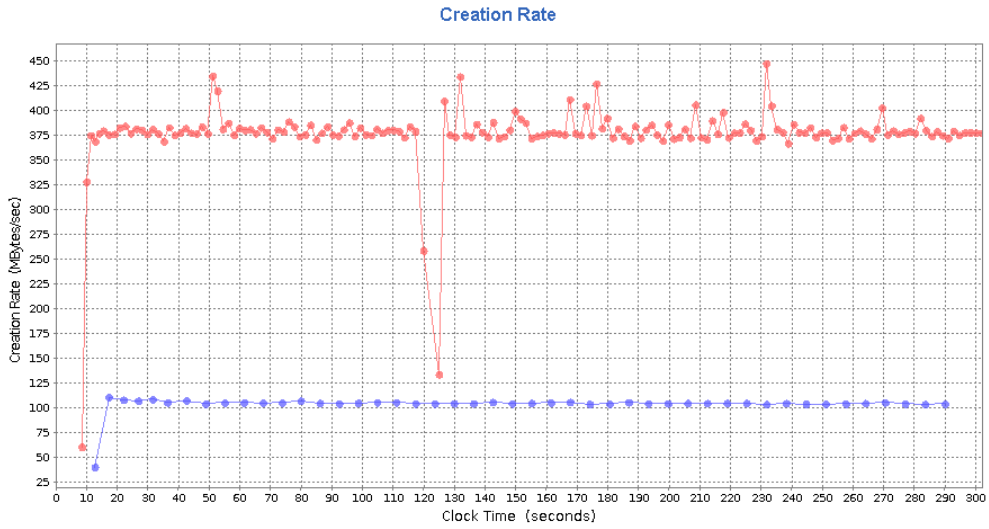

Figure 3.4. Object creation rate in mbyte/s for protobuf (red) and direct serialization (blue)

3.3.3 NetworkManager

All the incoming and outgoing TCP connections are handled by the *NetworkManager*. The *NetworkManager* is provisioned by the *RingManager*. Once the server is running, the incoming messages are dispatched to the different roles which had subscribed for delivery. Local messages, for example from a proposer to a coordinator running on the same node, are also sent through the *NetworkManager* manager. The manager takes care that such messages are never sent through the network.

Further, the *receive()* method inside the *NetworkManager*, called from a *TCPListener*, will first try to forward the message in the ring and then deliver the message locally. This saves some latency for messages that are not directly targeted to this node. Before forwarding, unneeded content will be removed from *Values*. This guaranties that the content of a *Value* is only sent once to every node.

The structure of *NetworkManager* is shown in Figure 3.5. *TCPSender* and *TCPListener* both run as threads. Messages are sent out through a *TransferQueue* which combines the *send()* method in the *NetworkManager* with the *socket.write()* in the *TCPSender*. The *TCPListener* creates a *SessionHandler* for every incoming connection and waits on a *Selector* for interruptions from the hardware. While in a NIO implementation this part would be typically done in a thread pool, this implementation is single threaded. A thread pool is not required since we will always have only one open incoming connection.

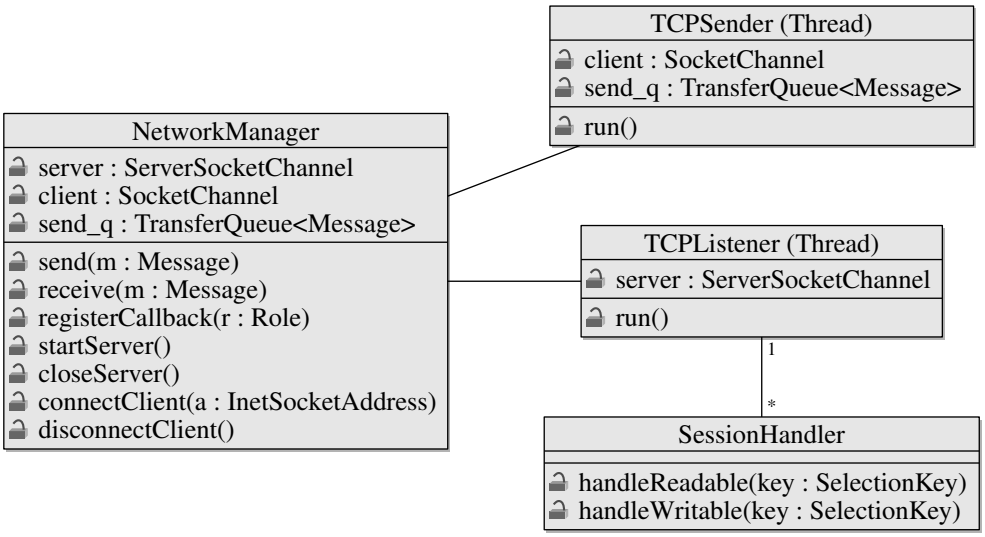

Figure 3.5. Class diagram of the network management

3.4 Stable storage

Depending on the actual implementation of the *StableStorage* interface, an acceptor may or may not recover after a crash. Everything that implements this interface can be configured in the Zookeeper cluster as back-end. The class is loaded with *Class.forName(name).newInstance()* and must follow the definition below:

```
public interface StableStorage {
  public void put(Integer instance, Decision decision);
  public Decision get(Integer instance);
  public boolean contains(Integer instance);
  public void close();
}
```

The current implementation includes the following storage back-ends:

NoMemory: This implementation is for protocol benchmarking only. Without storage, acceptors are not able to answer a single missing value. It is provided to measure the raw throughput without impact of garbage collection or disk writes.

InMemory: The *InMemory* implementation uses a *LinkedHashMap* to keep up to 15'000 *Decision*s in memory. Unfortunately, the overall throughput is very poor (550 Mbit/s). Even with the new garbage collector G1 the throughput can only reach 800 Mbit/s. This is more than 200 Mbit/s slower than the *NoMemory* implementation.

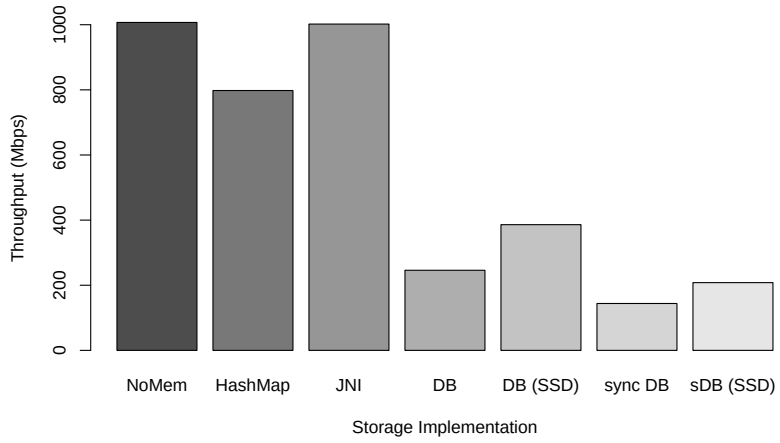

Figure 3.6. Performance comparison of different *StableStorage* implementations.

CyclicArray: *CyclicArray* is a array implementation in C. It uses JNI to provide a storage for up to 15'000 *Decisions*, not on the Java heap. This implementation has the same speed as *NoMemory* and is used in all experiments, unless stated otherwise. Since the objects are stored outside the heap, no garbage collection is required while overwriting old entries.

BerkeleyStorage: *BerkeleyStorage* is a key-value store used to provide stable storage. The database uses deferred writes, which means that we can tolerate the crash of a Paxos process but not of the whole machine.

SyncBerkeleyStorage: *SyncBerkeleyStorage* is a wrapper of *BerkeleyStorage* which enables synchronous disk writes. This is the safest configuration, since it can tolerate the crash of a whole machine.

A comparison of the different implementations deployed in a local-area network is provided in the Figure 3.6. Details about the experiment setup can be found in Chapter 4.

Chapter 4

Performance evaluation

One of the primary goals of this work is not only the protocol implementation, but also the evaluation of its performance. The overall performance depends on the implementation and on the protocol itself. While different aspects of the different protocols have been previously evaluated, there are many aspects left untested. [MPSP10], for example, evaluate the performance of Ring Paxos both in the presence and in the absence of IP multicast. Moreover, [MPP12] evaluate Multi-Ring Paxos with IP multicast. The combination of unicast Ring Paxos and Multi-Ring Paxos has never been evaluated before.

This new implementation makes it possible to evaluate Multi-Ring Paxos in new scenarios. Since it does not depend on IP multicast, it can be used outside of a single data center. While IP multicast would in principle work on WAN links, it is rarely available. This typically happens because UDP doesn't care about the existing network load and can flood the network with a huge number of packets. TCP however, which uses congestion control, would adopt this existing UDP traffic and slowdown itself to nearly zero throughput.

Nevertheless, Multi-Ring Paxos could be very interesting to operate on WAN links. While the existing evaluation tests the scalability on local machines, mainly to scale slow disk writes, it could also be used to scale a global installation. In this scenario, slow globally synchronized rings with high latency could be combined with fast local rings.

4.1 Experiments

This chapter first compares the performance of these new implementation of unicast Ring Paxos with the results provided from a previously evaluated implementation (Section 4.3.1). This should give a clear overview of the theoretical aspects of this protocol and validate the core ideas behind the protocol on a completely different implementation.

The combination of this Ring Paxos implementation together with the concept of multiple rings to scale a single atomic broadcast instance can be used to evaluate scalability. Section 4.3.2 will first focus on the scalability of Multi-Ring Paxos in different local computer clusters. The main goal of these experiments is to see how efficient local resources like CPU, network and disk can be saturated.

The last experiments will focus on Multi-Ring Paxos on WAN link. Amazons worldwide distributed computing cloud opens the ability to experiment on global installations.

4.2 Setup

The experiments are done with the following configurations on the described infrastructures. Every data point in the evaluation is an average of multiple runs (2-3) of the same experiment. In each run, multiple thousands of values are proposed (usually > 1 Mio.), which results in a experiment duration between three to five minutes each. The duration of every run should be long enough to see the impact of the implementation on garbage collection and other sorts of buffers in the system.

4.2.1 Test client and data acquisition

Values are proposed using a test client. Every proposer listens on the standard input. All input is proposed as content in a value. How many values are sent and how big they are can be configured within Zookeeper.

In a test mode, a proposer will first send the number of *concurrent_values*. Whenever one of these values is learned, it will send another one, until a maximum of the defined *value_count* is reached. The latency of every proposed value is added to a list. After all values are proposed, the latency distribution is written to the logging framework.

The logging framework (*log4j*) controls all output of a process. There is a category for statistics, which outputs every five seconds different data like network or learner throughput, or a value category, which shows the delivery of a value at a learner. The last category is a general one, which provides runtime information like ring management or exception handling. The verbosity of all categories can be controlled by the keywords “error”, “info” or “debug”. The output can be redirected to a file and/or standard output.

Additionally, every running Java process writes a garbage collection log file which can be analyzed for example with HPjmeter. The CPU usage is monitored with a Perl script which correlates the *top* output with the *jstack* information. This allows to see the CPU usage of every thread and not only of the whole Java process.

Table 4.1. Ring Paxos default configuration (per ring).

/ringpaxos/ring{ID}/config/*	
value_size	32768
value_count	500000
value_resend_time	20000
concurrent_values	10
quorum_size	2
stable_storage	ch.usi.da.paxos.storage.CyclicArray
learner_recovery	0
p1_preexecution_number	5000
p1_resend_time	20000
buffer_size	2097152
tcp_nodelay	0

Table 4.2. Multi-Ring Paxos default configuration (system-wide).

/ringpaxos/config/*	
multi_ring_lambda	600
multi_ring_delta_t	100
multi_ring_m	1

4.2.2 Configuration

If not stated otherwise, software release *R*3.3 of the implementation is used in all experiments. The implementation tested on Amazon EC2, however, uses software release *R*3.4. This version uses some minor modifications which are required to publish the external IP address on Zookeeper and did not change the protocol behavior itself. Further, the Zookeeper default configuration from Table 4.1 and Table 4.2 is used.

All Java processes run on OpenJDK 7 and are started with the following options:

```
JVM_PATH="-Djava.library.path=$PRGDIR/lib -cp $PRGDIR/lib:$CLASSPATH"

# GC implementation
#JVM_OPTS="-XX:+UseG1GC"
#JVM_OPTS="-XX:+UseParallelGC -XX:NewRatio=1"
JVM_OPTS="-XX:+UseParallelGC"

# GC logging options
JVM_OPTS="$JVM_OPTS -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -Xloggc:$HOME/$HOSTNAME-$$.vgc"

# start the program
$JAVA $JVM_PATH -Xms2G -Xmx2G $JVM_OPTS ch.usi.da.paxos.ring.Node $@
```

4.2.3 Infrastructure

The implementation is evaluated on three different infrastructures. The USI cluster consists of hardware nodes which are interconnected through 1 Gbit/s network links. The Switch environment is based on the open cloud infrastructure OpenStack with the benefit that every host machine has two 10 Gbit/s connections. The last infrastructure is the Amazon elastic cloud computing (EC2) platform, which can be used in a globally distributed environment.

USI

Cluster based on HP SE1102 servers. Connected through a HP ProCurve 2910 switch with 1 Gbit/s interfaces.

Nodes 1-17: 8 GB RAM, 2x Intel Xeon L5420 2.5GHz (Quad-core), 7.2k RPM, 500 GB, 16 MB buffer

Nodes 17-20: 8 GB RAM, 2x Intel Xeon L5420 2.5GHz (Quad-core), Samsung 840pro SSD

Nodes 31-33: 8 GB RAM, 2x Intel Xeon L5420 2.5GHz (Quad-core), OCZ-VERTEX3 SSD (VTX3-25SAT3-60G)

Switch

Switch uses kvm virtualization. Nodes 1-4 run on different physical machines, where two of them are Quanta S210-X12RS (2x 6-core Intel Xeon E5-2630 2.3GHz) and two Lynx Calleo 2250 (2x 16-core AMD Opteron 6272 2.1GHz). All systems have 128 GB RAM. The system-volumes are stored on local disks (Samsung 840pro SSD 256GB). Every host uses two 10 Gbit/s interfaces with several virtual lan connections (IEEE 802.1ad). Ring Paxos uses one vlan which is accessed via standard Linux bridging.

Zookeeper: m1.tiny, 4 GB RAM, 1 VCPU, 0 disk

Nodes 1-5: x1.large, 16 GB RAM, 4 VCPU, 10GB disk

Amazon EC2

Amazon elastic computing cloud.

Zookeeper: US East (N. Virginia), t1.micro, 613 MB RAM, up to 2 ECUs (1 core)

Nodes 1-3: N. Virginia different AZs, m1.medium, 3.7 GB RAM, 2 ECUs (1 core)

Nodes 4-6: EU (Ireland), m1.medium, 3.7 GB RAM, 2 ECUs (1 core)

Nodes 7-9: US West (Oregon), m1.medium, 3.7 GB RAM, 2 ECUs (1 core)

4.3 Results

4.3.1 Ring Paxos performance

Impact of the buffer size

Before the actual Ring Paxos performance can be evaluated in detail, several protocol parameters have to be optimized. The first of them is the impact of the TCP receive window size. This buffer size is very important in TCP, since the product of this window size and the packet round trip time defines the maximum throughput. This relation is mainly because TCP has to send an acknowledge back to the sender before it will send new data. The faster this acknowledge can be sent back (small latency) or the bigger the window of sent data is, the greater the throughput that can be achieved.

$$Throughput\ (bits/s) = Window\ (bits)/RTT\ (s) \tag{4.1}$$

In the USI cluster, the RTT of an ICMP packet is about $0.1ms$. With such a good latency, the impact of the buffer size in this cluster is expected to be small. To send 1 Gbit/s of data with 0.1 *ms* latency, a buffer of 12.5 kByte is required.

Setup: This experiment is run at USI on the nodes 1-3. Every node is a proposer, acceptor and learner (PAL) and every proposer sends several thousands of 32 kbytes values.

The impact of the buffer size is shown in Figure 4.1. By varying the receive window size from 128 kbytes up to 16 Mbytes the throughput and latency are constant. Only with the smallest buffer and in the case we increase the concurrent load of the proposers, a decreased throughput is measured. In general, increasing the TCP window size will also increase the latency until the packets get delivered to the application. Most important seems to be the influence of concurrent values of the proposers on the application latency. Concluding, for further tests a receive window size of 2 Mbytes and 10 concurrent values per proposer will be used. Also the impact of the TCP nodelay option (Nagle's algorithm) is evaluated. Since there were no significant differences in the measures, the results are not shown. Ring Paxos has an almost 50% to 50% relation between small and big packets. This applies because all big values are only transmitted once in the system. This implies that every proposed message requires a big value or phase 2 message, followed by a small decision message.

Impact of the packet size

The impact of the proposed value size is evaluated. The value size (command size if the algorithm is used for state machine replication) has an impact on the throughput, the latency and the number of messages that are sent per second. This experiment

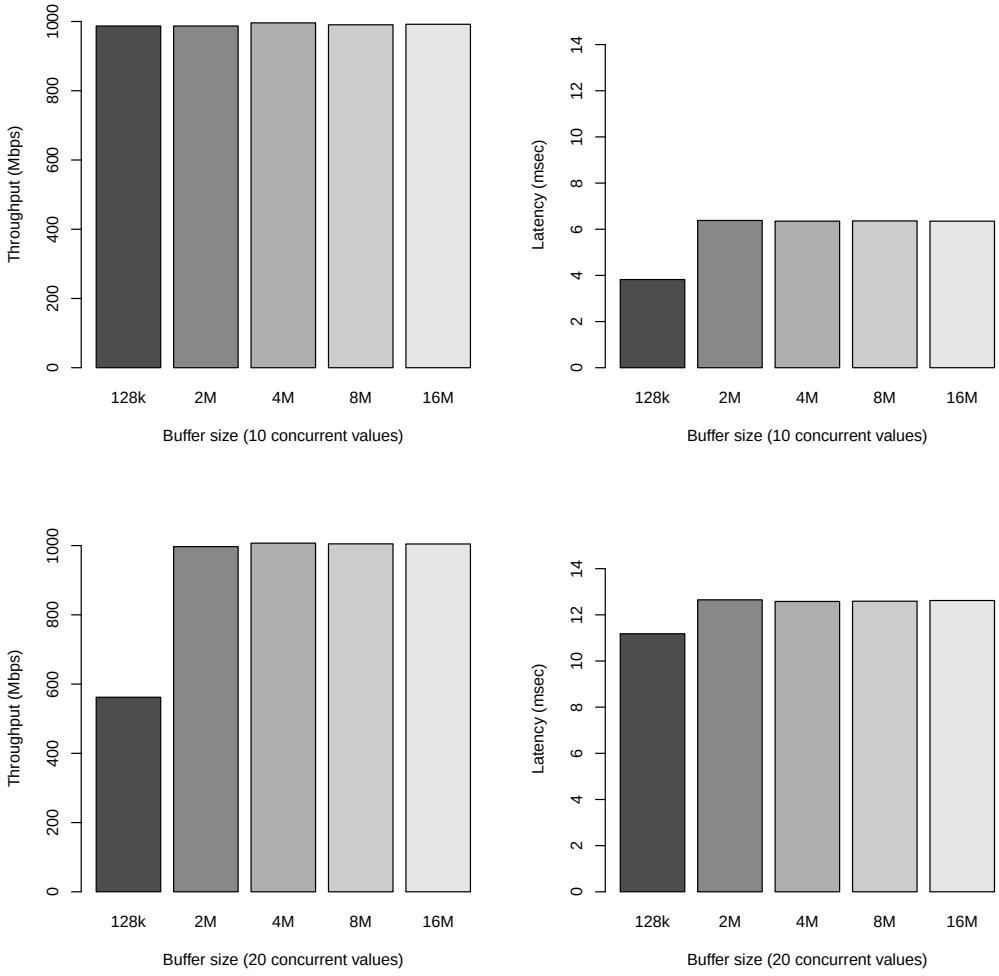

Figure 4.1. Impact of the TCP buffer size (32k values)

will show their relationship, and allow to choose the most appropriate values for the detailed performance analysis.

Setup: This experiment is run on the nodes 1-3 on the USI cluster. Every node is PAL. For this experiment every proposer send 20 concurrent values. The TCP window is set to 2 Mbytes.

The results are shown in Figure 4.2. As expected, the throughput increases with the proposed value size. The messages per seconds decrease almost linearly. The latency behaves, after a minimum of some required milliseconds, also linearly. Doubling the value size also doubles the latency. While a very good throughput of 900 Mbit/s and a latency below 4 ms can be achieved with 8 kbyte values, the maximum of 1 Gbit/s can only be reached with 32 kbyte values, this with the "price" of 12 ms latency.

Detailed performance analysis

Since we now have evaluated the optimal parameters (Table 4.1), we can start evaluating the performance in detail. The goal is, while varying the value size, to see the impact on throughput, latency and CPU usage for different storage back end implementations. This allows us to get a clear picture of what a single ring is able to achieve. This results are important when we will scale the algorithm by using multiple rings.

Setup: The experiment is run at the nodes 1-3 for standard disks and the nodes 31-33 for SSD disks on the USI cluster. Further, the experiment is repeated on the Switch environment on the nodes 1-3. Every node runs all roles (PAL) and every proposer proposes values.

Figures 4.3 and 4.4 summarize the results of the experiment. One notable fact of unicast Ring Paxos is that the maximum throughput, in small rings, is larger than the network line speed. This phenomenon takes place because every node is also proposing values to itself. These values are not transmitted over the network. The theoretical maximum throughput of unicast Ring Paxos on a 1 Gbit network (3 nodes) would therefore be 1.5 Gbit/s [GLPQ10]. Such a good throughput is not achieved with this implementation.

Despite the fact that the implementation uses several threads to benefit from multiple cores, the main algorithm is more or less single threaded. This is obvious since we implement a total order protocol and the degree of freedom to let the threads work independently is small. Some parts of the code, for example serialization can be done unordered, multi threaded. This explains, why the overall CPU usage at the coordinator is more than 100%. The limiting thread in the system is the *TCPListener*. This because it calls all the callback methods of all roles. Once this thread has reached the

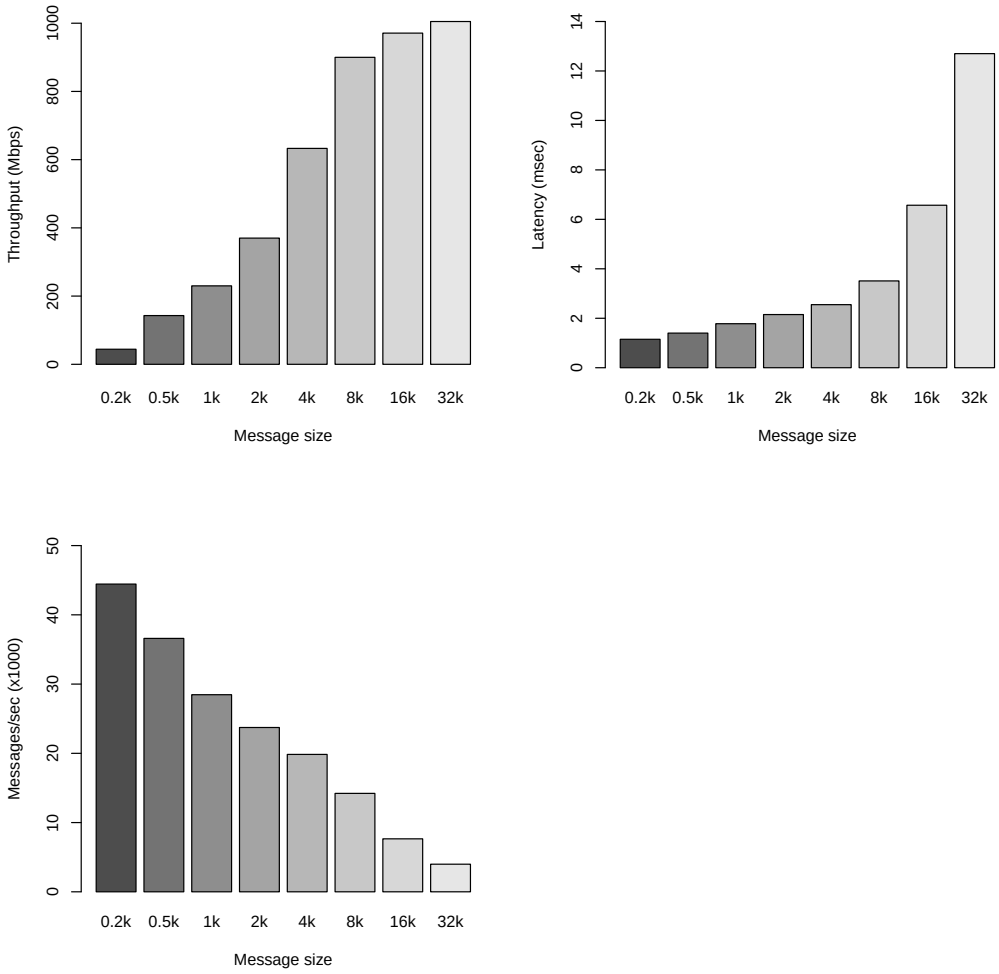

Figure 4.2. Impact of the application value (command) size (2M TCP buffer)

speed of a single core, the implementation reaches its maximum speed. In an early versions of this implementation, this component was built to be more loosely coupled, but because of the nature of the protocol, in which a message must be sent from a proposer to a coordinator, followed by all acceptor, it did not increase the performance.

The previous observation also explains why the implementation can not benefit from the faster network links on the Switch cluster. The measured throughput is higher than on the USI cluster, but not significantly. This is mainly due to the single core performance from the Switch CPUs, which is slightly better than the one at USI.

Interesting are the results of the different storage back ends. Writing to disk is slow, compared to the in-memory version. But using an asynchronous database on SSD disks, the performance is still quite good with almost 400 Mbit/s respectively 600 Mbit/s. However, to tolerate real crashes of the entire system, synchronous disk writes are required. With SSDs, synchronous writes of almost 200 Mbit/s are possible. The latency, while using any kind of disk writes, is very high. While the SSDs can benefit from large write buffers to improve the throughput compared to standard disks, the CDF of the latencies shows that there is no real difference in the latency distribution. The main latency impact comes from synchronous vs. asynchronous writes. The best performance (throughput vs. latency) is achieved with 8 kbyte values and not with the maximum of 32 kbytes.

Single ring scalability

The previous experiments were conducted with rings of only three nodes. While this is a reasonable assumption for such a protocol in a real production environment, it is still interesting to understand the behavior with more ring participants. Since $2 * f + 1$ nodes are required to tolerate f failures, three nodes are perfectly fine in practice.

Setup: This experiment used the nodes 1-20 on the USI cluster. Every node was a PAL and every proposer was proposing values.

The results are shown in Figure 4.5. The latency curve shows the clearly linear behavior. So adding new nodes increases the latency introduced by the new node itself and for an additional network connection. The throughput initially decreases rapidly. This happens because of the known phenomena that proposers sending values to themselves, gets smaller by adding more nodes [GLPQ10]. After about 10 nodes join the ring, the throughput stays almost constant which a small decrease.

Interesting are the results shown in the CDF. Exactly 50% of all proposed values are much faster than the other 50%. This becomes more and more visible by adding additional nodes to the ring. While this 50% gap is also present with only four nodes in the ring, the width which the gap spans increases as more nodes are added. This is due to the implementation. There are two threads, the *ProposerRole* and the *TCPListener* which contend on a lock and wait on the *TransferQueue* from the *TCPSender*

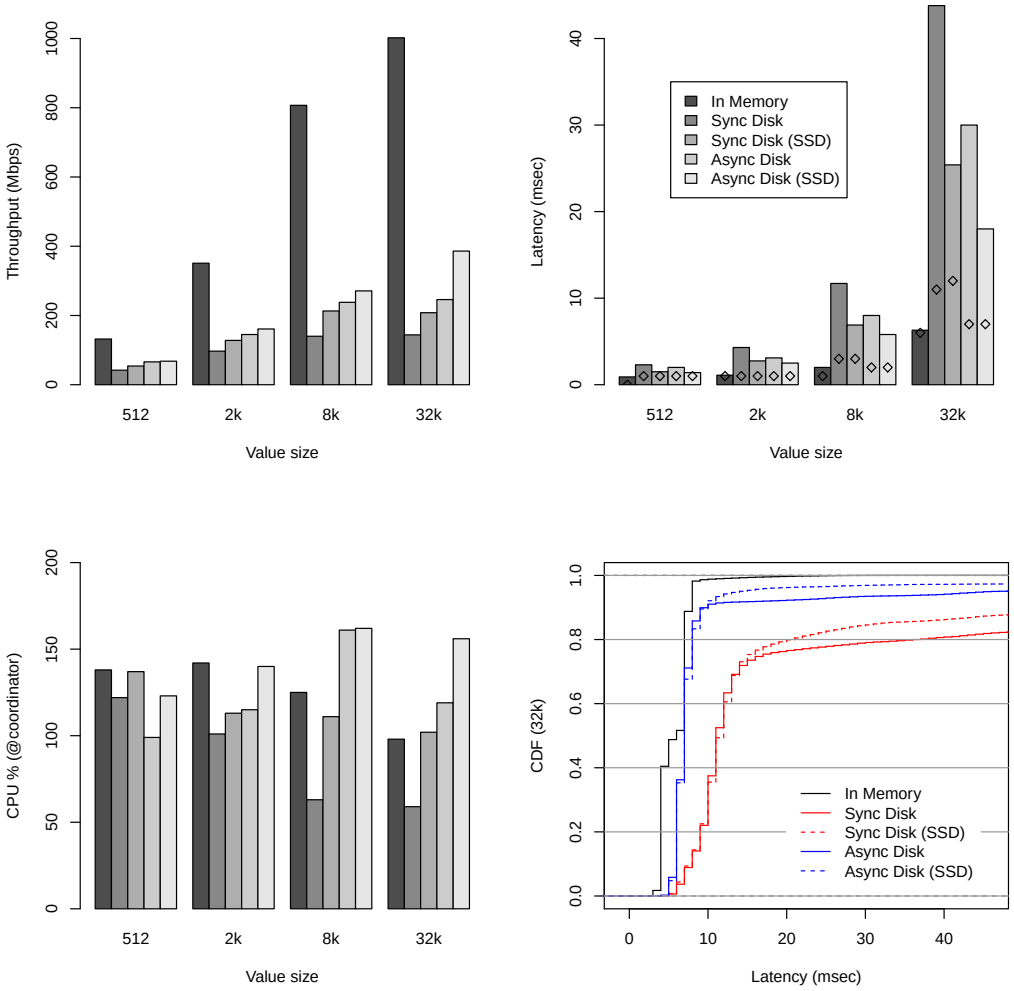

Figure 4.3. Single ring performance USI

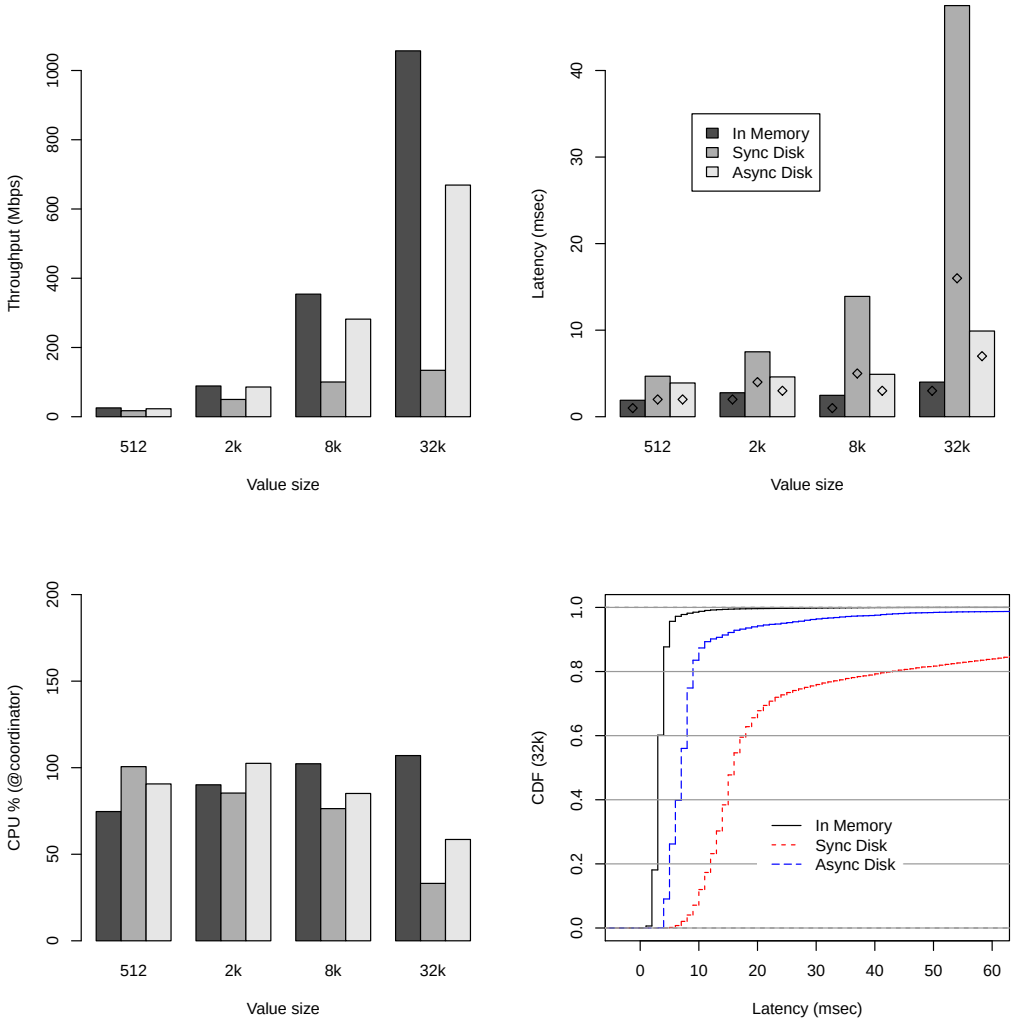

Figure 4.4. Single ring performance Switch

thread to get their messages sent out. By adding more nodes to the ring, while every proposer is trying to send out the same amount of concurrent values, the scheduling pressure between receiving more values and proposing more values increases the overall latency. A CDF without the pressure of having multiple concurrent threads is shown in Figure 4.10, while in the same environment, but with three proposers the CDF in Figure 4.11 shows the 50% gap.

4.3.2 Multi-Ring Paxos performance

Overhead and efficiency of skip messages

So far, the results of the experiments consider only the performance of a single ring. To scale this results, many single rings can be combined together. A simple approach, still guaranteeing a total order, is to deliver m messages from one ring, followed by m messages from the next ring in a round-robin manner. This implies that all ring send at the same speed. If this is not the case, the coordinator of each ring has to issue skip messages to allow the progress of the faster rings.

In this experiment, these skip messages are evaluated against their overhead and efficiency. Two rings in three runs are evaluated, where (1) there is no application traffic in the rings, (2) one ring sends at maximum speed and the other is quiet and (3) both send at maximum speed.

During this experiment, the best matching default configuration which is summarized in Table 4.2, was also evaluated. This default is used in the following Multi-Ring Paxos evaluation.

Setup: The test was run on the USI cluster using the nodes 1-7. Nodes 1-3 are part of ring 1 (3 * PAL), while nodes 4-6 are part of ring 2 (3 * PAL). Node 7 is a *MultiRingLearner* which is part of both rings. For the experiments, in which data are sent, only the first proposer per ring was proposing values.

Figure 4.6 shows the results of the three runs. The x-axis corresponds to time and the y-axis shows the number of values buffered at the *MultiRingLearner* before delivering. Without skip messages, these buffers, one for every ring, would consume all the memory of a learner until they would crash. On the other hand, sending too many skip messages would only generates a lot of traffic with unused Paxos instances and costs CPU at the learners to apply them in round-robin. One skip message is 40 bytes and can skip an arbitrary number of instances.

Experimentally, the most efficient configuration was observed when λ equals 1.5 times the maximum expected value rate multiplied by Δ_t. Since we expect a maximum of 4000 values/s, $\lambda = (4000 * 1.5) * 100ms = 600$. $Delta_t$ was chosen with 100 ms. This because smaller values were hard to schedule accurately and generated a lot of traffic, while larger values were too slow to act on fast ring speed changes. Sending

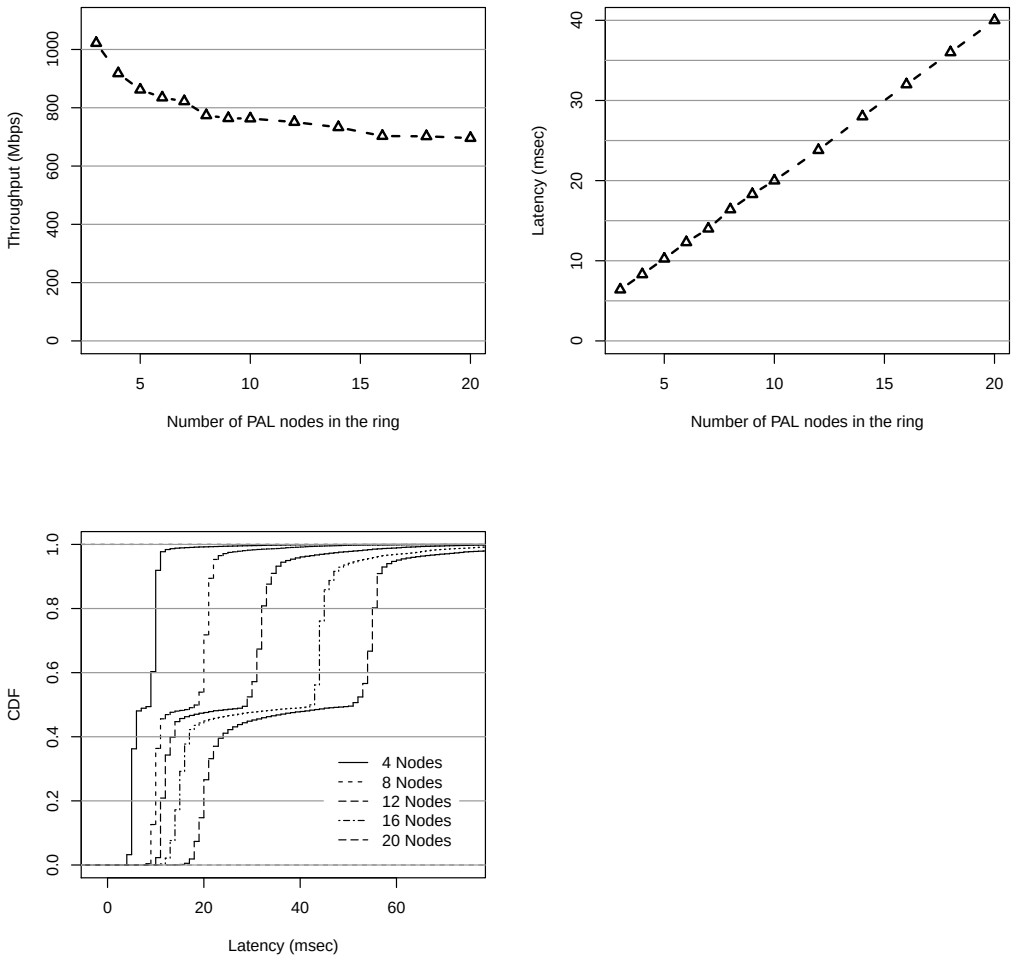

Figure 4.5. Impact of the ring size

every 100 ms a skip message resulted in a skip traffic of only 3.2 kbit/s.

The first chart shows the waiting rate at the learners by not sending any traffic. Only the skip messages are in the system, which may also be cached. At some sample points, there was one instance with a skip message waiting. This is fine, since the measurement samples at any arbitrary time and is not precisely synchronized with the *MultiRingLearner*. As long as the wait queues do not grow, the skip system works.

The second extreme run shows the behavior of only one sending ring. In this case there were continuous waiting values for delivery in ring 1. But since the message rate was below 4000 msg/s and the wait queue was always smaller than 1500 messages this can be seen as a good result. The introduced latency of a ring with 680 Mbit/s and one with 0 Mbit/s is only in an order of milliseconds.

The last run shows the results where both rings send at maximum speed. The first ring caches up to 2500 values in the beginning. This might be due to the fact that processes were not started synchronized. However, the initially queued 2000 values in ring 1 could be emptied during runtime. After 150s the ring speed even changed and the ring 2 cached some values.

Since those three cases are extreme ones, the results have shown that the skip messages technique guarantees the progress on a fast ring. Furthermore, it was able to span the time which one ring had to wait for disk write or garbage collection.

Multiple rings on different hosts

Since the protocol parameters were now evaluated, we can start using Multi-Ring Paxos to scale slow Ring Paxos instances. We achieved the slowest performance with synchronous disk writes to standard disks. This experiment should target the scalability of multiple slow rings, which are running on different machines, to achieve a higher aggregated throughput.

Setup: The experiment uses nodes 1-16 on the USI cluster. Nodes 1-3 form a 3 * PAL ring 1, nodes 4-6 ring 2 and so on until nodes 13-15 ring 5. Node 16 is a *MultiRingLearner* which subscribes to all rings.

The left chart in Figure 4.7 shows multiple rings with in-memory storage. The throughput is limited by the network interface of the *MultiRingLearner* on node 16. More interesting is the right chart. In this case, multiple slow acceptors with synchronous disk writes were aggregated to an overall throughput of 600 Mbit/s. This approach can be used to scale slow disks to build up a high throughput installation. Moreover, the scaling is more or less linear with the number of added rings. It can grow up until the network interface is limiting or the single core performance.

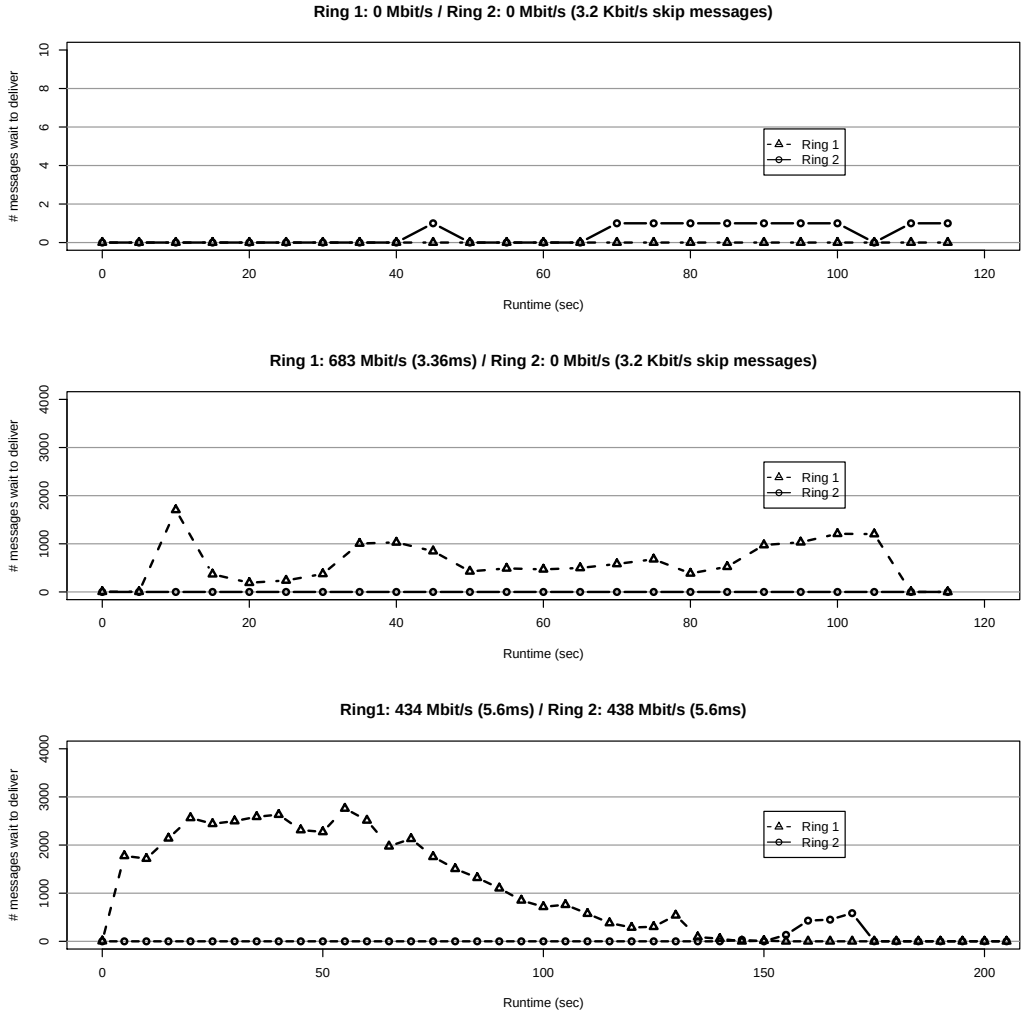

Figure 4.6. Efficiency of Skip messages under different ring loads ($\lambda = 600$ / $\Delta_t = 100ms$)

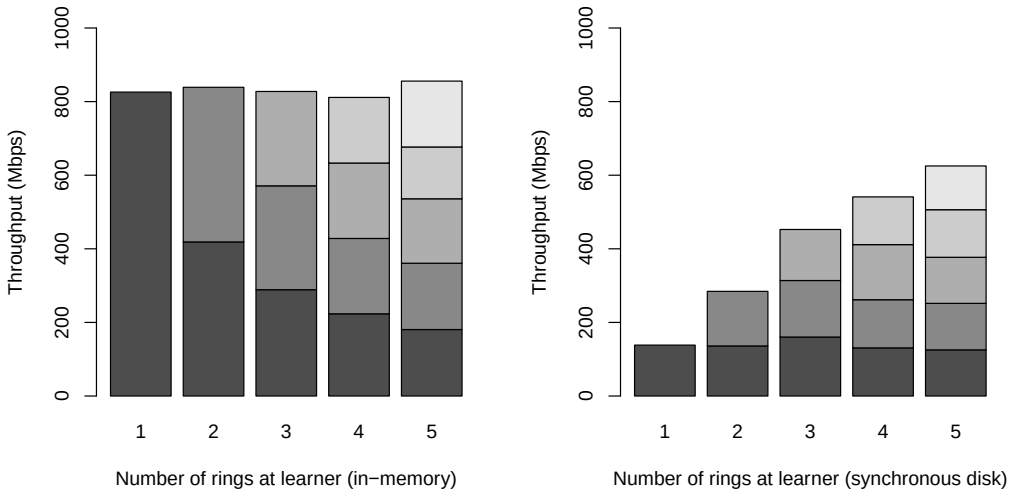

Figure 4.7. Multiple rings at one learner performance (rings on different 16 nodes)

Multiple rings on same hosts

While the previous experiment has shown how Multi-Ring Paxos can be used to scale the overall throughput using many slower machines, this experiment demonstrates the ability to scale on one machine. Spawning multiple rings on a single host could be used to better saturate the existing hardware. In the case of the Switch cluster, where we have 10 Gbit/s interfaces, it would be nice to get an aggregated throughput in that dimensions. This should be evaluated.

Setup: This experiment uses the nodes 1-4 on the Switch and on the USI cluster. In all cases the first three nodes are 3 * PAL and every proposer is proposing values. The last node 4 is a *MultiRingLearner* which subscribes to all rings.

The results are show in Figure 4.8. While the results on the USI cluster were limited by the expected limit of the network interface, the results on the Switch cluster were surprising. They do not show the expected results. In fact, the maximum throughput first increases with the number of rings per node. But after three rings, the overall throughput gets worse.

Analyzing the CPU log files might explain why. The virtual machines on the cluster provides four virtual CPUs. This CPUs however are emulated as a maximum of four concurrent threads and they might be even run on the same physical CPU on the host hardware. Since the implementation is best performing with a bit more than 100% CPU, it is clear that the optimum must be three rings. Adding more rings would only

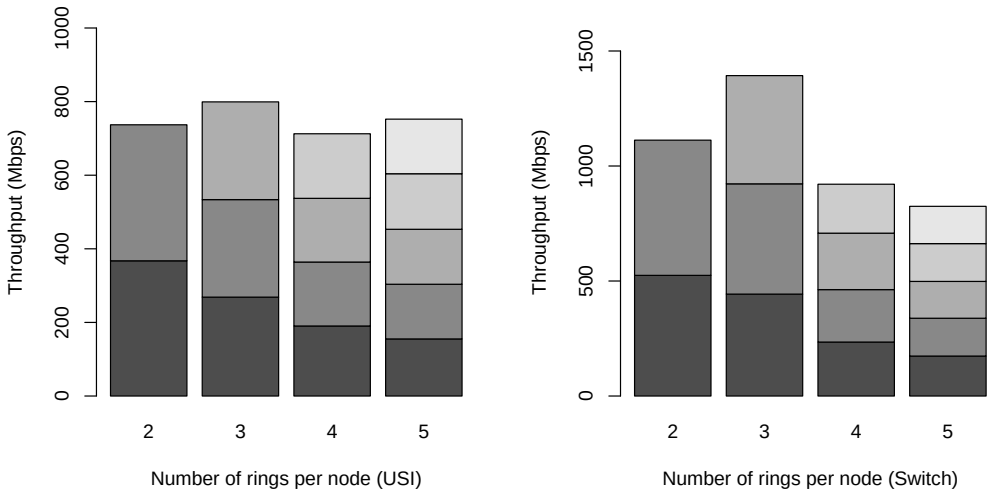

Figure 4.8. Multiple rings per node performance (all rings on 4 nodes)

increase the pressure of the scheduler and the context switching.

The experiment has shown that in fact local resources can be used more efficiently. But this approach can also be limited, in this case by the amount of available virtualized CPUs.

Multiple geographically distributed rings

The last experiment demonstrates the effect of globally synchronized rings. While in WAN connections the latency is usually high, Multi-Ring Paxos should be used to combine such slow rings with faster local rings. In practice, this approach can be used if, for example, a few global writes have to be synchronized with a lot of local traffic.

Setup: The experiment is run on the Amazon EC2 infrastructure. There are three local rings with 2 * PAL and the last node per ring is a learner only; in US west, US east and EU west. The nodes at the US east coast are further deployed over different availability zones (different data centers). A fourth global ring is deployed over all three regions (Figure 4.9). The last learner in the local rings is also a PAL in the global ring.

The performance in the Amazon data centers varies from day to day. Two measurement done on two different days are shown in Figures 4.10 and 4.11. In general, both demonstrate that it is possible to combine rings running in different speeds with Multi-Ring Paxos. Not only fast and slow local rings can be combined together, the global ring also has no impact on the local throughput or latency.

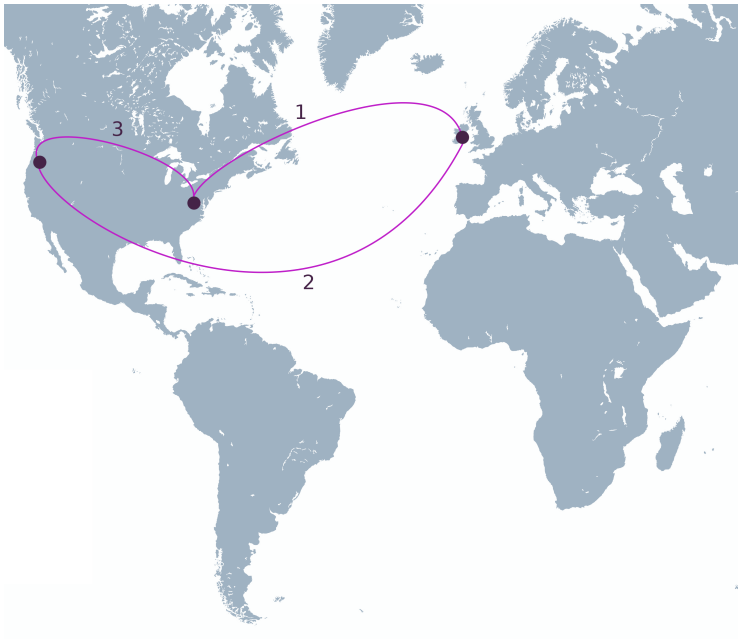

Figure 4.9. Amazon EC2 Global Ring

The maximum local throughput of 700 Mbit/s is a good result for shared hardware. The performance of the US east coast is less because the servers are located in different availability zones. This ring could even tolerate a local disaster in which a whole data center would be offline.

While on one day the overall performance was slower and without grave outliers in the proposing latency, on the second day it was faster in all tests. Interestingly, also the average latency was much higher on the second day. The median of all the latencies would hide this outliers a bit, but the difference in the day performance would remain. Even two consecutive runs produce very varying results. This might explain why in Figure 4.11 there seems to be an impact on the global ring compared to the local ring performance.

The performance of the global ring is about 10% of the maximum local throughput (5.8 Mbit/s and 7.5 Mbit/s). In practice, this means that only 10% of all write requests should be globally synchronized.

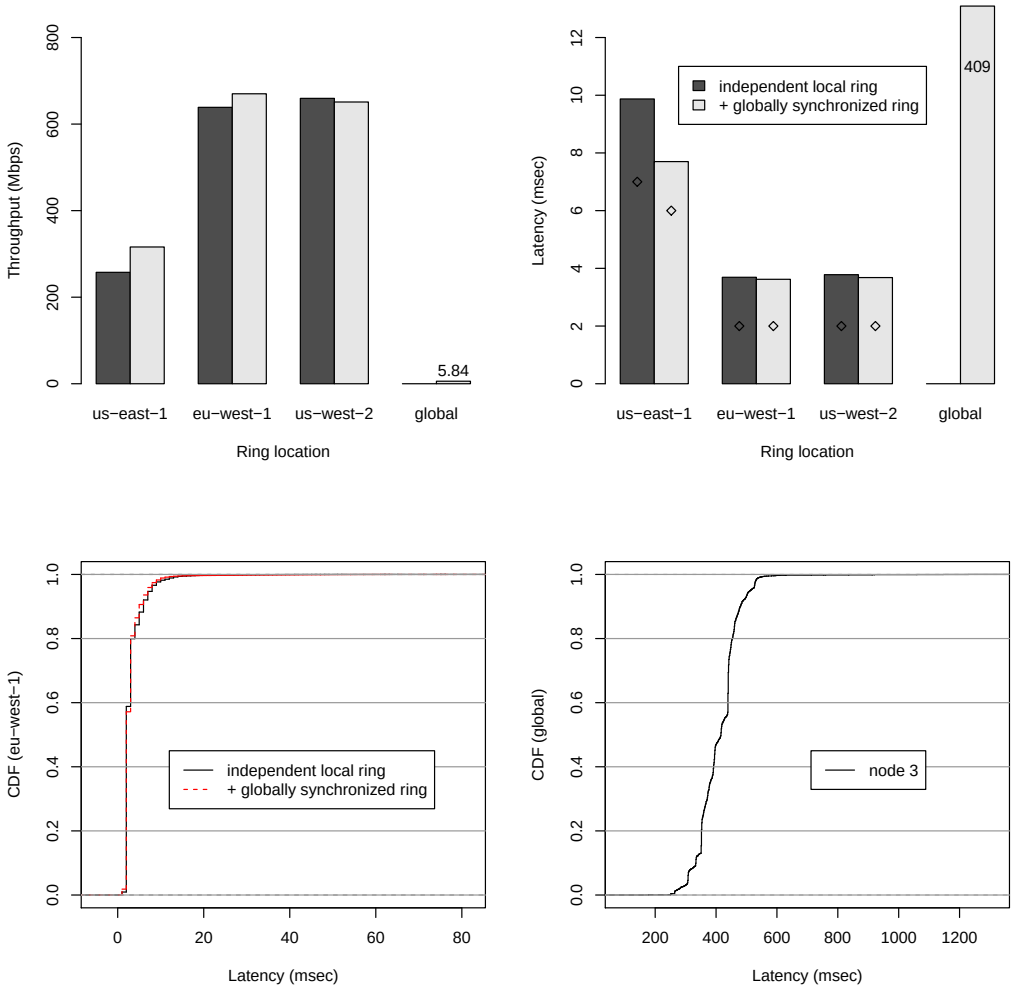

Figure 4.10. Independent local ring performance compared to local rings plus an additional global ring which synchronizes all three regions (day 1).

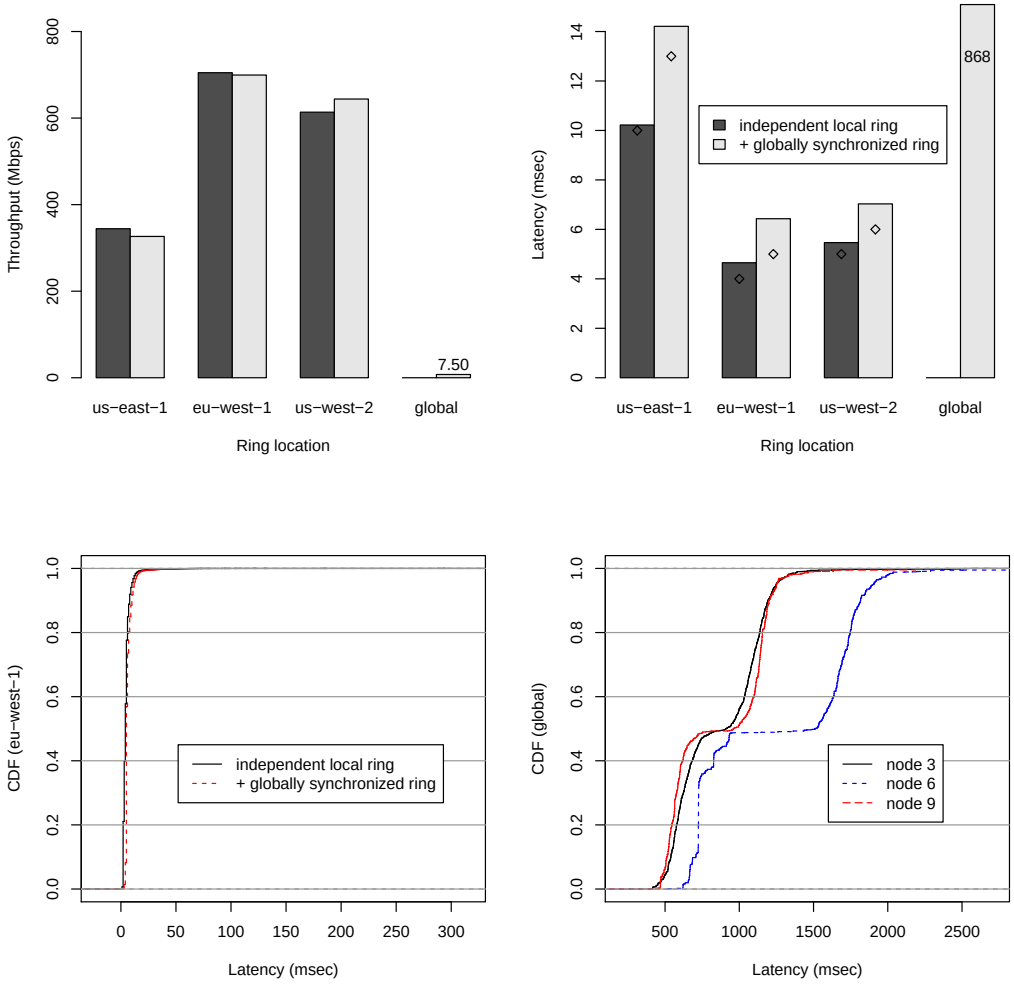

Figure 4.11. Independent local ring performance compared to local rings plus an additional global ring which synchronizes all three regions (day 2).

Chapter 5

Conclusions

5.1 Discussion

This thesis starts with an introduction (Chapter 1) about the problems which nowadays arise in distributed systems. The fundamental background of distributed, fault tolerant consensus is discussed in Chapter 2. The demand for high performance atomic broadcast protocols is targeted in the research of [MPSP10] and [MPP12]. The implementation of unicast Multi-Ring Paxos, which is provided in this work, is based on this previous effort.

Chapter 3 explains the overall structure of the way this protocol is implemented. Further, it highlights the specific ring management component, which is crucial to achieve high performance. This implementation tries to be as complete as possible, allowing its deployment in many different ways and scenarios. It provides several stable storage implementations to evaluate throughput. Acceptor and learner recovery is implemented and an external coordination service makes the configuration management and automatic ring deployment easy. Moreover it provides a generic and ready-to-use interface based on a cross-language RPC stack.

Chapter 4 starts with the evaluation of Ring Paxos. Similar experiments were already run previously. This allows to compare this specific and new implementation with those results. While Ring Paxos is focused on high throughput, Multi-Ring Paxos is built for scalability; both properties have been evaluated in this work. The evaluation concludes with two experiments which have never been done before. The first one tests Multi-Ring Paxos on virtualized machines with 10 Gbit/s interfaces. Unfortunately the virtual CPUs were not able to provide the required performance to scale up to this line speed. The last experiment demonstrates the ability to use Multi-Ring Paxos in a globally distributed ring. It combines several fast local rings with a slow global ring in order to synchronize all traffic.

The outcome of this work has demonstrated that the initially gathered insight of Ring Paxos and Multi-Ring Paxos could be validated with this completely new and

different implementation. The extension of Multi-Ring Paxos with unicast transport opens further opportunities to use it in other projects.

5.2 Future work

Two aspects can be improved in the current implementation.

RingManager: Currently, the *RingManager* sorts the ring according to the ID of the nodes. It does not take into account the role of a node. While the implementation works and is always correct with this approach, it might not be latency optimal. A better ring order would be to sort all nodes in the ring according to their role. Beginning with all the proposers followed by the acceptors and ending with all the learners.

NetworkManager: As already mentioned in the implementation part, the TCP framing should be improved by an additional magic-frame identifier number. This would allow the receive buffer to re-synchronize after an unhandled exception.

Additionally, the implementation could be improved by abstracting the *MultiRingLearner* in a way, that would allow to replace different atomic broadcast protocols for different rings. Such a modification could be used to change for example the implementation used on global ring with a latency optimized version and still use the throughput optimized local rings.

Lastly, more research is needed to improve the storage behavior of the acceptors. To enable the acceptors to recover after a crash, real stable storage is required. The data which accumulates at each acceptor can be high if the throughput is at 1 Gbit/s. If the acceptors would understand the transmitted data, like the commands in a state machine replication, a sort of full state acceptor could be implemented. In such a case, the acceptors could apply the commands to their own replica to compress the amount of required storage.

Bibliography

[AS00] Marcos K. Aguilera and Robert E. Strom. Efficient atomic broadcast using deterministic merge. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, pages 209–218, New York, NY, USA, 2000. ACM.

[BJ87] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems (TOCS)*, 5(1):47–76, 1987.

[Bur06] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350. USENIX Association, 2006.

[CGR07] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM.

[CT96] Tushar D. Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[DSU04] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, December 2004.

[ES06] Richard Ekwall and André Schiper. Solving atomic broadcast with indirect consensus. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 156–165. IEEE, 2006.

[Fis83] Michael J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In *Foundations of Computation Theory*, pages 127–140. Springer, 1983.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[GLPQ10] Rachid Guerraoui, Ron R. Levy, Bastian Pochon, and Vivien Quéma. Throughput optimal total order broadcast for cluster environments. *ACM Transactions on Computer Systems (TOCS)*, 28(2):5, 2010.

[HKJR10] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, volume 8, pages 11–11, 2010.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[Lam01] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[LMS05] Paul J. Leach, Michael Mealling, and Rich Salz. A universally unique identifier (uuid) urn namespace. RFC 4122 (Proposed Standard), July 2005.

[MJM08] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 369–384, Berkeley, CA, USA, 2008. USENIX Association.

[MPP12] Parisa J. Marandi, Marco Primi, and Fernando Pedone. Multi-ring paxos. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2012.

[MPSP10] Parisa J. Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. Ring paxos: A high-throughput atomic broadcast protocol. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 527–536. IEEE, 2010.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[Ste07] Randall R. Stewart. Stream control transmission protocol. RFC 4960 (Proposed Standard), September 2007. Updated by RFCs 6096, 6335.

[WBM+06] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D.E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.