ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems Group
Department of Computer Science

Master Thesis

Flexible Replication for Personal Clouds

by
Dejan Juric

April 14, 2010

Advisors:
Dr. Oriana Riva
Prof. Timothy Roscoe

Department of Computer Science
Swiss Federal Institute of Technology (ETH), Zurich, Switzerland
ETH-Zentrum, CAB
CH-8092 Zürich

Abstract

People own an increasing number of personal devices ranging from mobile phones and laptops to tablet and desktop computers. In addition, it is more and more common to rent cloud storage resources from utility computing providers. We call this new computing environment a user's *Personal Cloud*. Managing data in such a heterogeneous environment requires a large effort on the user side. Given the storage limitations of some of these devices and the increasing amount of digital information stored on them, users demand a way to partially replicate their data across the devices in their Personal Cloud. Such a replication system must be flexible enough to fulfill the requirements of various potential applications. The heterogeneous devices may have different communication capabilities which disallow direct communication to every other device in the Personal Cloud. Furthermore, disconnections and failures of the devices must also be taken into account. In this thesis, we present a flexible replication infrastructure that is inspired by the PRACTI system and offers partial replication, arbitrary consistency, and topology independence. We have integrated our system in the Rhizoma runtime to benefit from its self-monitoring functionality and the ability to dynamically redeploy in the execution environment. We have extended PRACTI with metadata-based grouping of content and support for content-based filters. We have assessed the performance of our replication system on a realistic testbed consisting of mobile devices, desktop PCs and cloud virtual machines. Results show the feasibility of our approach and the incurred overhead.

Contents

Chapter 1

Introduction

1.1 Motivation

Computers have become ubiquitous and users increasingly own more devices of various form factors. These devices range from personal computers and laptops to tablet devices and mobile phones which offer a wide range of capabilities.
We envision an environment which facilitates the development of distributed applications that run on all of the user's devices and call this environment a *Personal Cloud*. Such Personal Clouds can consist of local machines that are owned and fully controlled by the user, as well as machines rented from Infrastructure as a Service (IaaS) providers like Amazon's Elastic Compute Cloud (Amazon EC2) [1], PlanetLab [20], or similar offerings. Figure 1.1 gives an example of a Personal Cloud.

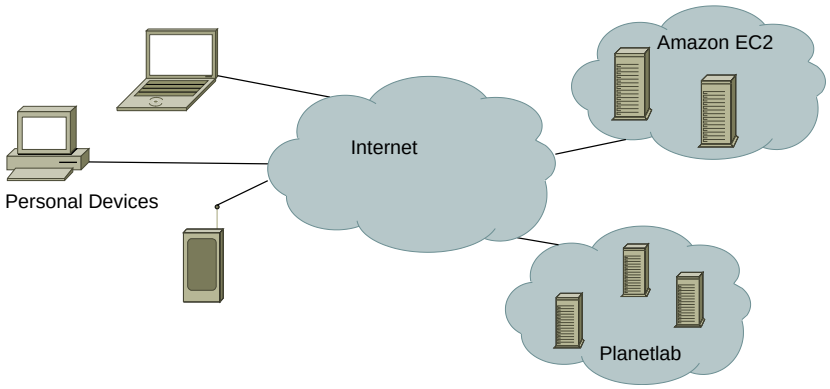


Figure 1.1: Personal Cloud comprising personal devices and cloud provider instances

An essential element of such a system is a data storage service which must address the needs of various potential applications. Devices in Personal Clouds are not constantly connected and can fail frequently. To guarantee data accessibility and a reliable service, data replication is necessary. Data replication is the task of distributing the data to different physical machines and storing it in a distributed way.

Typical systems tend to be built with very specific requirements that apply to the specific problem that is to be solved and focus either on reliability of the replication system or on its performance. Providing replicated data storage for applications that run in Personal Clouds, however, requires high flexibility in order to allow applications to benefit from a broad range of properties and guarantees. Since not every use case can be known beforehand, the goal is to provide an environment that covers a wide range of possibilities.
To offer the needed flexibility, a replicated data storage system must support the following three PRACTI [4] properties:

1. Partial Replication: not every machine in the system must have all data.
2. Arbitrary Consistency: some data items might need to be consistent across all devices in a Personal Cloud, meaning that every device always accesses the newest version. Other data items might not require this guarantee and tolerate delays until the newest data items are accessible.
3. Topology Independence: not all devices must be directly connected to each other.

Partial replication is needed because some devices in a Personal Cloud may have limited storage capacity, which makes it impossible to replicate the full data store. The reasons for arbitrary consistency stem from theoretical limitations which state that only two of the following three properties can be provided:

1. strong consistency: whenever a data item is read, the newest version is returned,
2. system availability: it is always possible to read or write, and
3. tolerance to partitions: the system continues to work even when different parts are not connected anymore.

A detailed explanation of these properties follows in Chapter 2.
Having arbitrary consistency allows balancing these three properties and, for example, reduce consistency to provide availability or to tolerate partitions. Providing arbitrary consistency raises new problems. In particular, the definition of consistency is important, together with the way of defining, interpreting, and using the offered consistency mechanisms. Easily configurable application programming interfaces (APIs) are needed, to expose full flexibility. Topology independence is required if two devices have no direct connection. E.g., a mobile phone connected via Bluetooth to a personal computer must still be able to participate as a member of the system even if another machine is present to which the mobile phone has no direct link.

1.2 The Rhizoma Data Storage

This thesis presents a replication infrastructure offering the flexibility needed by applications running in Personal Clouds. The infrastructure supports partial replication by separating metadata about writes from the actual content of the writes, and by disseminating summaries on the sets of writes to all nodes. The system guarantees eventual consistency, which means that the system always

reaches a consistent state, although with potential delays. We offer a weakened form of consistency, *causal consistency*, and *sequential consistency* through the Paxos consensus algorithm [13]. Applications can tune the required consistency level via a simple API which allows to specify bounds on the number of outstanding writes that can be tolerated.
Our system also offers simple ways to specify sets of items to be partially replicated. This is accomplished by using interest set definitions that are defined on the keys of items. Furthermore, a user can define rules that encompass various device- and content-specific properties to specify where certain items must be placed.
The replication infrastructure we have built is inspired by the PRACTI [4] system, which offers partial replication, arbitrary consistency and topology independence. We leverage PRACTI to additionally support content-based partial replication. We accomplish this by integrating the data replication system with Rhizoma [29], which is a middleware for developing applications that are capable of self-maintenance in a distributed environment by adapting to changing execution conditions.
This thesis makes the following contributions.

1. We extend PRACTI with metadata-based grouping of content, interest set subscriptions, and support for content-based filters. All these features allow the replication system to offer a richer, semantic description of the data storage.

2. The replication infrastructure is integrated in Rhizoma. It uses Rhizoma's overlay for communication and for collecting information about the nodes in the overlay. Rhizoma uses the replication infrastructure to replicate its system state.

3. The system runs on a variety of computing platforms ranging from desktop PCs, and mobile phones, to Amazon EC2 virtual machines.

1.3 Outline of the Thesis

The remainder of this thesis describes the replication infrastructure in detail. Chapter 2 gives an introduction to relevant concepts and puts this thesis in the context of related systems and technologies. The overall architecture and design of the system is introduced in Chapter 3, while implementation details are covered in Chapter 4. Chapter 5 presents results of the experimental evaluation. Finally, Chapter 6 concludes this thesis, describes limitations and open problems, and shows prospective areas for future work.

Chapter 2

Background

In order to understand the built system, some background knowledge is needed which is explained in this chapter. We start with an introduction to the problem of time in distributed systems. Afterwards, we present an overview of the different approaches to data replication. Then, the topic of consistency in the context of data replication and an overview on how Paxos works. We also give an introduction to Rhizoma, a runtime environment for distributed application. Lastly, related systems are discussed which attempt to provide similar functionality or use similar concepts.

2.1 Time in Distributed Systems

In distributed systems, when all processes are independent and do not share a global clock, the concept of time becomes ambiguous. Unlike centralized systems, where each application can ask the kernel to return the current time, such functionality is more complicated to achieve in distributed systems. Figure 2.1 shows an adapted example from [25], where two computers execute events at local times, but a divergence of the local clock can be observed, such that the event occurring at local clock value 42 on computer B has globally occurred later than computer A's event at local clock value 43.

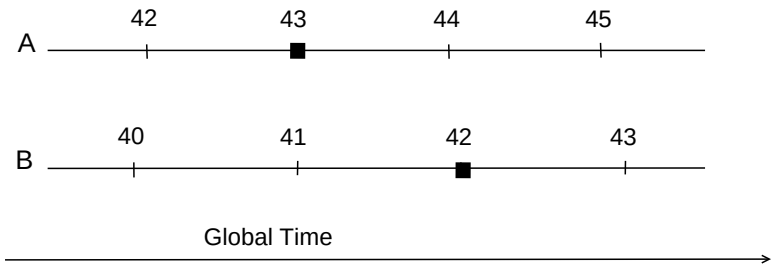


Figure 2.1: The clock values 43 and 42 of the two marked events on the machines A and B indicate that *A's* event occurred after *B's*, although the position in the global time shows the opposite. Using the clock values on these two machines to relate events will lead to wrong conclusions.

There are two possible approaches to overcome this problem:

1. synchronization of physical clocks, or
2. use of logical clocks

Synchronization of clocks can be achieved by using various synchronization algorithms. A good overview is given in [25]. Using logical clocks for the events happening in a system has the advantage of not requiring synchronization algorithms, but still providing sufficient information to relate the events that occurred on the various processors. Our system relies on logical clocks, which will be explained in the remainder of this Section.

2.1.1 Lamport's Logical Clocks

Lamport's logical clocks assume that a system consists of several processes and each process executes events. The events form a sequence that is totally ordered on a single process. Furthermore, sending and receiving a message also constitutes an event in a process. The *happened before* $\rightarrow$ relation is then defined by Lamport in [11] as the smallest relation on the set of events that satisfies the following conditions:

1. If two events x and y occur on the same process and x comes before y, then $x \rightarrow y$.
2. If y is the receiving of a message that originated from the sending event x, then $x \rightarrow y$.
3. If $x \rightarrow y$ and $y \rightarrow z$, then $x \rightarrow z$.

If $x \not\rightarrow y$ and $y \not\rightarrow x$ and $x \neq y$, then x and y are *concurrent*.
Using this *happened before* relation, clocks are defined as a function C_i for every process P_i which assigns an integer value $C_i(x)$ to an event x. Across all processes, C can be defined as the function that assigns any event y an integer value $C(y)$ where $C(y) = C_j(y)$ if y occurred on process j.
The clock condition then states, that:
For any events x, y: if $x \rightarrow y$ then $C(x) < C(y)$.
While Lamport's logical clocks assign an increasing value to every event y that happened after any event x, it is not possible to infer the causaulity relation between two events only from their clock values. This follows from the clock condition, where the inversion does not hold, i.e., $C(x) < C(y) \not\Rightarrow x \rightarrow y$ for any event x and y. The most we can deduce from Lamport's logical clocks is that if $C(x) < C(y)$, then x may have happened before or at the same time as y.

2.1.2 Version Vector Clocks

To be able to infer the causality relation between events, version vector clocks can be used. Version vector clocks have been introduced by Mattern in [16] and are an extension of Lamport's logical clocks. Instead of having only one linear logical clock value, a vector of clock values is used. Each entry in the vector represents the logical time of one process, and the logical time of the process obeys the rules of Lamport's logical clocks.

With version vector time, each process P_i has a clock C_i of length l where l equals the total number of processes in the system. When an event occurs, process P_i increments the i-th component of its clock:

$$C_i[i] := C_i + 1$$

The timestamp of an event is defined as the version vector immediately after the event occurred. This timestamp can be sent together with the message. The recieving node updates its local version vector time by combining it with the received one,

$$C_i := sup(C_i, t)$$

sup is a function which produces the maximum of each component. C_i is the locally known version vector time, while t is the timestamp of the message.
For any two virtual times specified by version vectors a and b, the following three properties hold:

1. $a \leq b$ iff $\forall i : a[i] \leq b[i]$
2. $a < b$ iff $a \leq b \wedge a \neq b$
3. $a \| b$ iff $\neg(a < b) \wedge \neg(b < a)$

Note that $\|$ means that the two events happened concurrently. From these properties, it is now possible to relate two events and either infer the potential causality, or deduce that the two events occurred concurrently. The ability to relate events using version vectors will be important in later sections, where consistency is discussed. Our system uses version vectors to determine the relations between events that happened in the system.

2.2 Data Replication

2.2.1 Replicating State Versus Operations

A fundamental question for replicated data stores is in which way the data is replicated. There are several ways to replicate data, each with its own advantages and disadvantages.

Propagation of data One possibility is to propagate the data resulting from a given operation. After a locally executed write, the written data is distributed to the other replicas. Figure 2.2 illustrates the propagation of the resulting value v for the item x from node A to node B.

Propagation of updates Instead of sending the actual data, it can be worthwhile to distribute only the operation that led to the final data value. This approach is called *active replication*, because the operation is executed on every replica. The main advantage is that the operation is often much smaller than the actual resulting data value. But this comes at the cost of having to execute the operation on every node. Several systems propagate updates instead of values, most notably Bayou [19], where writes consist of multiple update operations and a merge procedure. The disadvantages are increased complexity

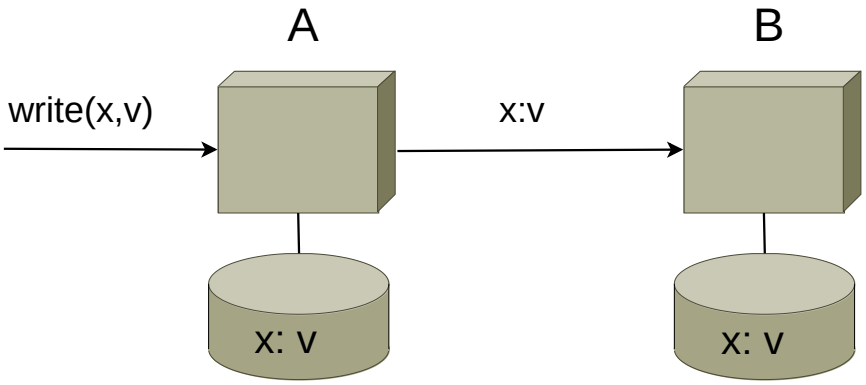


Figure 2.2: Propagation of the value v for x

and complications with respect to executing operations on stale data where a reexecution is required to ensure that each node has a consistent view of the data.
The propagation of an operation f with input y that is to be applied to item x is shown in Figure 2.3. Both node A and B have to execute f to obtain v. It is important to note that f may depend on other locally stored items. It is foreseeable that propagating operations needs either sufficiently strong consistency guarantees to prevent the operations from reading stale data, or mechanisms to detect and recover from faulty states.

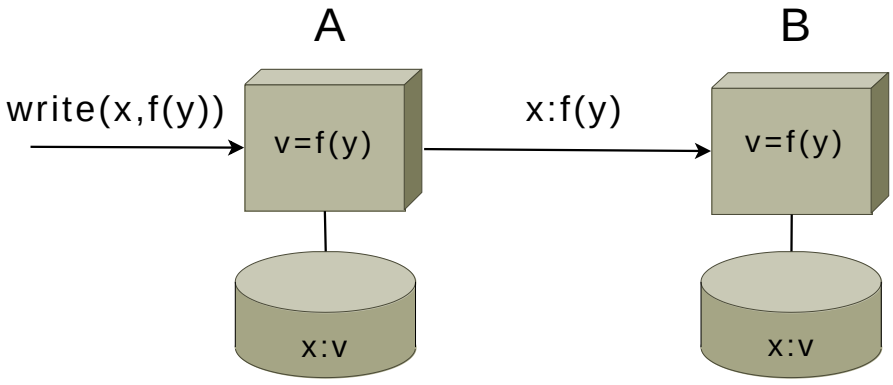


Figure 2.3: Propagation of x's update with the result from f on input y

Propagation of notifications Another possibility is to only notify other nodes about updates that occurred, instead of sending the updates. Such protocols are called *invalidation protocols* because they render the stored value on a receiver node invalid. Figure 2.4 shows one such case where v_2 has been written to x on a node A. Node B receives a notification that item x has been changed. Starting from the point of the arrival of this notification, the stored value of item x on the node B must be treated as invalid. While the notification informs node B about the change to x, which makes the stored value v_1 invalid, the new

value of x is still unknown. Systems using notifications still have to resort to propagations of the actual data or operations that led to the resulting data. The benefits, however, are that node B is now informed and can fetch the updated value upon a read request for x.

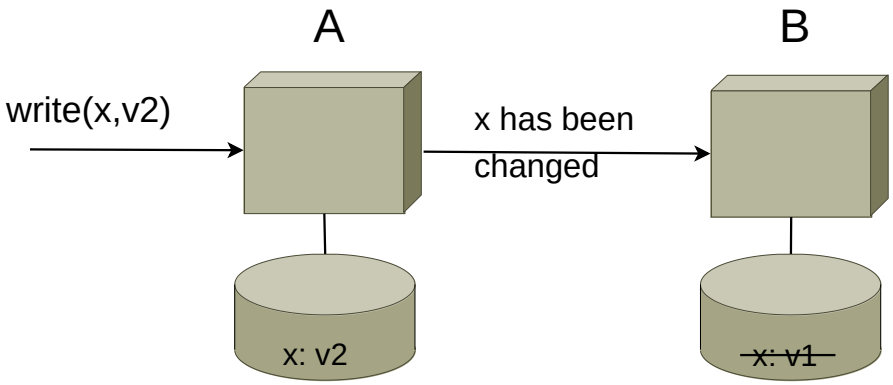


Figure 2.4: Propagation of the notification that item x has changed

2.2.2 Log-based Replication

Instead of writing the received data directly to the persistent storage, some systems use a log of occurred writes. Storing the writes in a log before processing them provides the possibility to reorder the log content and execute the writes in a different order. This approach is used by several replication systems, for example Bayou [19], PRACTI [4], and PacificA [15].

2.3 Replication Protocols

Replication protocols define the policies and needed actions for replicating data among machines. They define two fundamental properties of data replication:

1. the nodes which can execute a write, and
2. the messages exchanged between the nodes.

Additionally, by requiring that each machine participating in the replication obeys the rules of the protocol, replication protocols can offer consistency guarantees on the written data. Several approaches have been developed and will be introduced in the following sections.

2.3.1 Primary-based Protocols

In primary-based protocols, only one machine, the *primary*, receives and executes writes. By having one dedicated machine executing all writes, sequential consistency, which will be explained in detail in Section 2.4.1, can easily be guaranteed because all writes are ordered on one machine. Two possible protocol strategies in primary-based protocols exist: remote-write protocols and local-write protocols. In remote-write protocols, the primary is fixed and all

machines forward the writes to this primary. Figure 2.5 shows three nodes with primary A. While node A can execute the write w_2 locally, the writes w_1 and w_3 from nodes B and C, respectively, must be forwarded to the primary A.

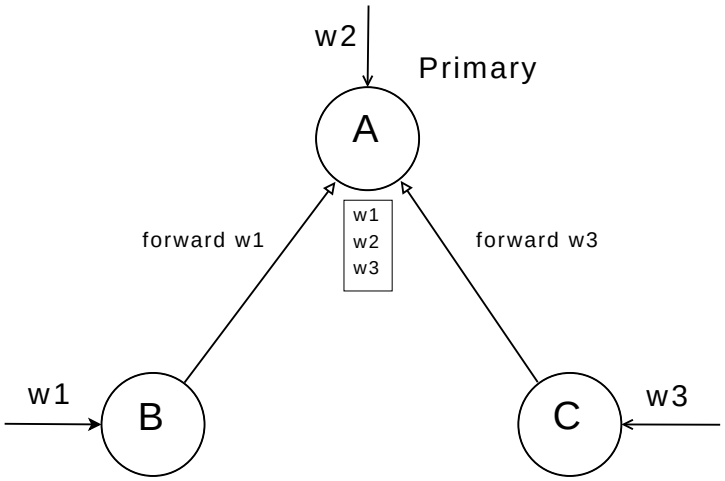
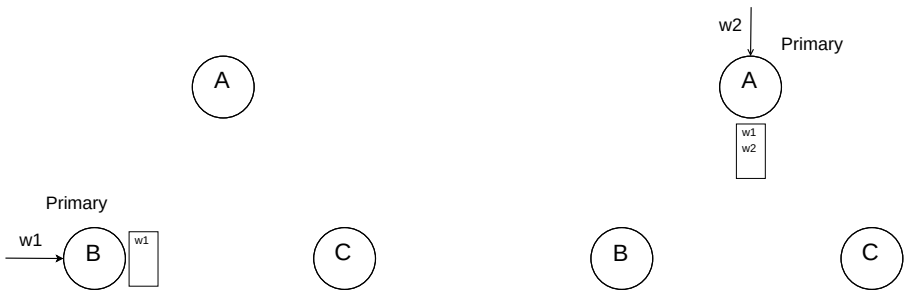


Figure 2.5: Remote-write protocol: all writes are forwarded to the primary

Local-write protocols also require that all writes are executed on the primary, but instead of forwarding the writes to the primary, they allow other machines to become the primary, such that the write can always be executed locally. The first two writes from the remote-write example in Figure 2.5 are shown in Figure 2.6 when using a local-write protocol. Nodes A and B each are the primary when they are executing their writes. While this approach eliminates the need to forward writes, it requires a mechanism to change the primary.



(a) Local write on primary A

(b) Local write on primary B

Figure 2.6: Local-write protocol: the primary changes such that writes can be executed locally

2.3.2 Replicated-write Protocols

In replicated-write protocols, a write can be executed at multiple replicas, as opposed to primary-based protocols, where this is only possible at the primary.

This approach, called *active replication*, requires each replica to have a process that carries out update operations. Updates are propagated to the other replicas, where the write that caused this update will also be carried out. The biggest challenge with active replication is that sequential consistency cannot be achieved without additional measures. Causal consistency can be reached by using Lamport clocks [11] or version vectors [16]. We will introduce consistency models in Section 2.4.1.
The previous example with three replicas A, B, and C and the writes w_1 to w_3 is shown in Figure 2.7 using active replication.

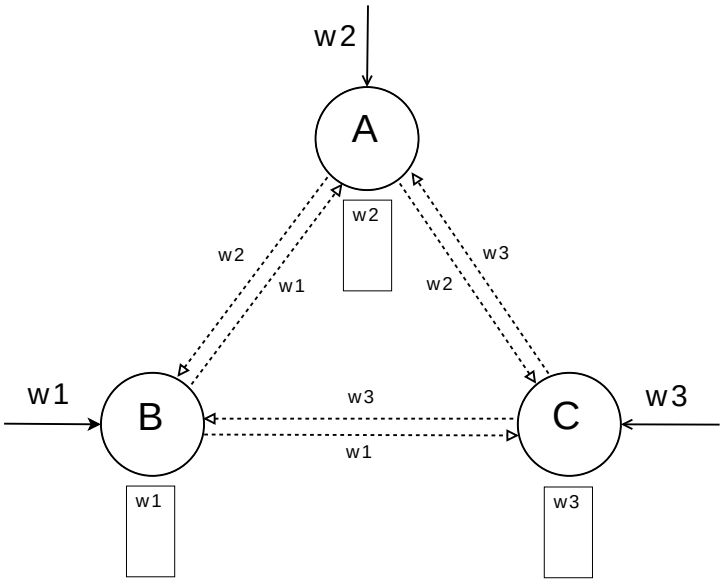


Figure 2.7: Active replication: each replica can execute the operation locally. The update is then sent to other replicas.

2.3.3 Quorum-based Protocols

Quorum-based protocols are yet another possibility to replicate writes. As opposed to primary-based protocols, quorum-based protocols attempt to get a majority of votes that will allow a certain operation to be carried out. If a majority has accepted the operation, the node can execute it. Figure 2.8 shows node A attempting to execute write w_1. A majority is found with node A and B. Node A is therefore free to execute the write.

2.3.4 Pull Versus Push Protocols

Push protocols propagate updates to other replicas without being asked to do so, while with pull protocols, the receiver must ask for the updates. Both approaches have advantages and disadvantages, and the decision to use one over the other depends on the expected usage scenarios for the application using the replication system. Figure 2.9 shows a write w_1 on replica A and how this write is propagated to replica B. In Figure 2.9(a) the update is immediately

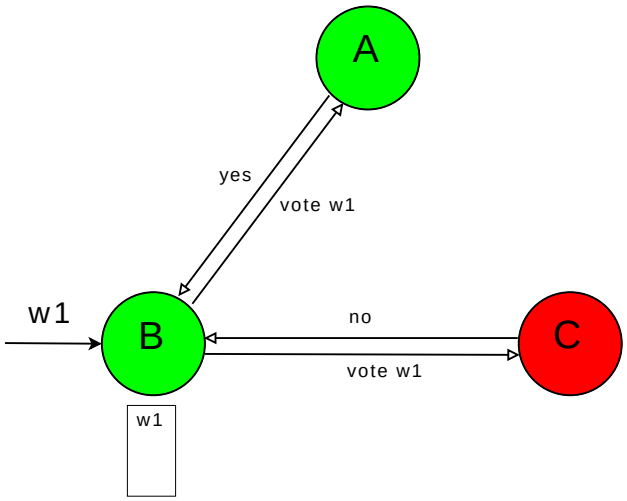
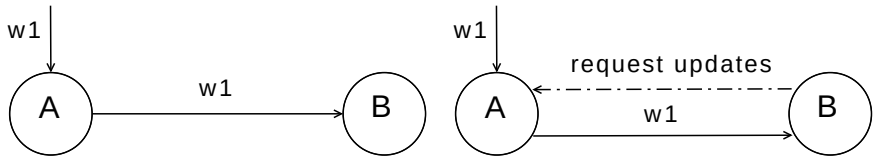


Figure 2.8: Quorum-based voting: node B can execute w_1 because a majority accepts it, no other write can interfere

pushed to node B while in Figure 2.9(b) node B must specifically ask node A to propagate the update.



(a) Push protocol (b) Pull protocol

Figure 2.9: Push versus pull protocols

2.4 Consistency

Consistency of replicated storage systems can be defined based on the ordering of read and write accesses that each node observes. We introduce the theoretic concepts the remainder of this thesis relies on.

2.4.1 Ordering of Operations

To reason about the consistency guarantees, we need the concept of order with respect to time.
Let $\preceq$: $\mathcal{O} \times \mathcal{O}$ be a binary relation on the set $\mathcal{O}$ of read and write operations. We call $\preceq$ a *total order* if the following properties hold for all $w_i, w_j, w_k \in \mathcal{O}$.

1. $w_i \preceq w_j \wedge w_j \preceq w_i \Rightarrow w_i = w_j$ (Antisymmetry)
2. $w_i \preceq w_j \wedge w_j \preceq w_k \Rightarrow w_i \preceq w_k$ (Transitivity)

3. $w_i \preceq w_j \vee w_j \preceq w_i$ (Totality)

These properties are equivalent to demanding that each write can be uniquely assigned a position on a linear timeline where no two writes can happen at the exact same moment. Figure 2.10 shows an example with the writes w_1 through w_5 on a timeline from left to right.

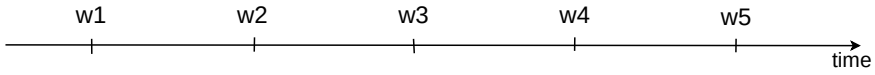


Figure 2.10: Timeline of writes with a total order

We call $\preceq$ a partial order if the following properties hold for all $w_i, w_j, w_k \in \mathcal{O}$.

1. $w_i \preceq w_i$ (Reflexivity)
2. $w_i \preceq w_j \wedge w_j \preceq w_i \Rightarrow w_i = w_j$ (Antisymmetry)
3. $w_i \preceq w_j \wedge w_j \preceq w_k \Rightarrow w_i \preceq w_k$ (Transitivity)

Figure 2.11 shows an example with the writes w_1 through w_5 where neither $w_3 \leq w_4$ nor $w_4 \leq w_3$ holds. Note that this instance cannot be shown on one linear timeline, because the relation between w_3 and w_4 is not known.

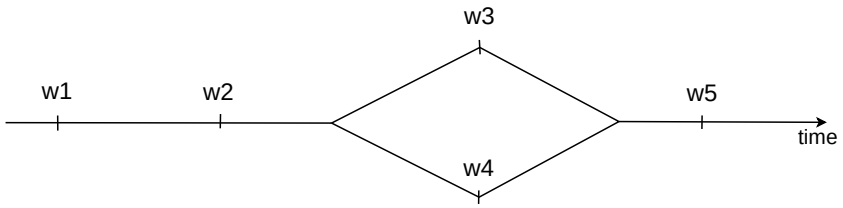


Figure 2.11: Example of a partial order: w_3 and w_4 are not related to each other

Having defined the basics of the ordering relation, we can now explore consistency models that apply to replicated data stores.

Causal consistency Causal consistency is described by Hutto and Ahamad [10]. It is a weak form of consistency which only requires that potentially causally related operations are consistently ordered, without emposing restrictions on the ordering of causally unrelated operations.
For a data store to be causally consistent, the following is a necessary requirement (according to Tanenbaum [25]):

> *Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.*

Looking at the partial order given in Figure 2.11, we can assume that writes w_3 and w_4 occurred on different processes concurrently. To fulfill the requirements of causal consistency, a process must transform this partial order to a total order. Two possible results are valid and each process is allowed to pick one of them:

1. w_1, w_2, w_3, w_4, w_5
2. w_1, w_2, w_4, w_3, w_5

If the two writes w_3 and w_4 apply to the same data item, reads on the processes with differing orderings may return different values.

Sequential consistency Sequential consistency has been first defined by Lamport [12] as consistency model for shared memory in multiprocessor systems. Tanenbaum [25] describes the condition for a data store to be sequentially consistent as follows:

> *The result of any execution is the same as if the (read and write) operation by all processes on the data store were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program.*

Causal consistency requires that potentially causally related writes are seen in the same order by all nodes in the system. Sequential consistency is stricter than causal consistency. It requires that all operations are executed in some sequential order that is consistent with respect to the order observed by each individual process [3]. We can again take the partial order from Figure 2.11. Sequential consistency then would require, that each process observes a consistent order, i.e., the processes may pick an order from the two causally consistent ones, but all of them must pick the same.

Linearizability Linearizability, which has been introduced by Herlihy and Wing in [9] and is also explained in [3] goes further by requiring that the order must also preserve the order of non-overlapping operations globally. Linearizability is what is typically meant by strong consistency [30].
Taking again the example from Figure 2.11 and assuming that the write w_3 occurred before w_4 (in global time), then the processes must choose the order w_1, w_2, w_3, w_4, w_5.

2.4.2 Strong and Weak Consistency

Less formally, strong consistency can be explained by assuming a system consisting of three machines A, B, and C. Strong consistency is provided if after an update executed by A is completed, all further accesses by any of the three nodes A, B, or C will return the updated value [28]. If this guarantee is not provided, a system is regarded to be weakly consistent.

2.4.3 Eventual Consistency

Eventual consistency is a special form of weak consistency, which assumes that in the absence of new updates, the system will eventually return the last updated value [28]. Systems offering eventual consistency define an *inconsistency window*, during which only weak forms of consistency are guaranteed. The inconsistency window's maximum size may depend on communication delays, the number of replicas and the load on the individual machines. After this inconsistency period, the system guarantees strong consistency.

Vogels [28] classifies causal consistency as a variation of eventual consistency. Additionally, more variations are identified which can be offered by eventually consistent systems.

- Read-your-writes consistency: a read on a node A after an update on that node will return the updated value.
- Session consistency: during a session, read-your-writes consistency is guaranteed. The guarantees are not provided across sessions.
- Monotonic read consistency: after a node has read a certain value, it will never read an older one.
- Monotonic write consistency: an update to an item is completed before any successive update to that item.

2.4.4 Continuous Consistency

Yu and Vahdat [30] introduce continuous consistency which tries to quantify inconsistency by looking at the probability of inconsistent accesses. The inconsistency bounds can be specified using three application-independent metrics: *numerical error*, *order error*, and *staleness*. Numerical error specifies a bound on the total weight of a write, which can be defined as numerical change in an integer value. Order error describes the number of tentative writes, i.e., writes where the final total order among all writes is not yet known. Staleness quantifies the maximum amount of time tentative writes are tolerated. Algorithms to enforce these bounds are presented, which control the time at which communication between replicas must occur. Fine-grained control on these error bounds is possible with the concept of *conits* which allows to define sets of items for which bounds can be specified, instead of specifying them for individual items only. TACT is an implementation of the continuous consistency model. PRACTI [4] uses the TACT framework to offer arbitrary consistency.

2.4.5 CAP Dilemma

During an invited talk [5], Brewer introduced the *CAP Theorem*, which states that:

> *You can have at most two of the following three properties in any shared-data system:*
>
> - *Consistency*
> - *Availability*
> - *Partition-tolerance*

Gilbert and Lynch formalize the CAP Theorem in [7] and explain consistency, availability and partition-tolerance. Consistency means linearizability which has also been explained in Section 2.4.1 and is further explained in [3]. In linearizability, any read that starts after a given write has finished must return the previously written value. In essence, every operation must behave as if it had

been executed on a single node. Availability means that in the absence of failures of the request-issuing node every request completes successfully. Partition-tolerance requires that even in the presence of a network partition with nodes in one part disconnected from another one, the system must not respond incorrectly.

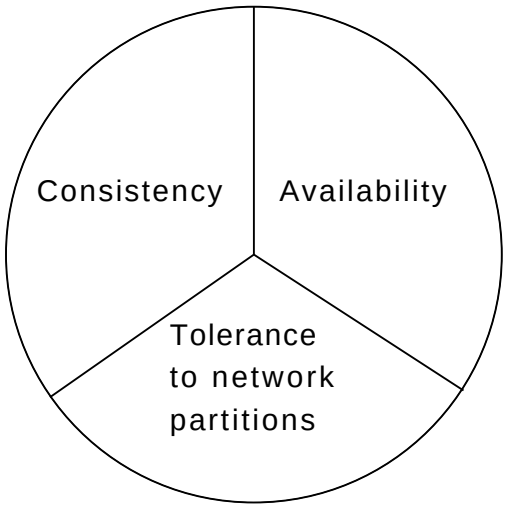


Figure 2.12: CAP Theorem: only two of the properties can be guaranteed

This theorem has far reaching consequences for distributed systems and can be observed in different systems sacrificing one of the properties or weakening the guarantees to fulfill application-specific requirements. Brewer's keynote [5] gives examples of systems for combinations of the three properties.

2.5 Paxos

Paxos, introduced by Lamport [13], is a consensus algorithm that allows a group of machines in a distributed system to agree on a value. The algorithm guarantees that, on termination, the algorithm will reach consensus even in the presence of unreliable networks or if several machines propose differing values. Maziéres [17] gives a good introduction and describes how Paxos can be implemented.
The algorithm runs in three phases:

1. A proposer choses a proposal number and sends the message *Prepare(n)* to all other nodes. A node can reject this message if it has already received such a *Prepare* message with higher proposal number. Each node responds with a *Prepare-Result(n′,v′)* message, where n' is the highest proposal number encountered by this node where $n' < n$ and v' is the corresponding value. If no prior proposal has been received, n' and v' are set to 0 and *nil*, respectively.

2. If a majority of the group does not reject the *Prepare* message, the proposer choses v to be the highest-numbered *Prepare-Result*. If all received

values are *nil*, the proposer is free to chose any value for v. This is then sent with a *Propose(n, v)* message to all other nodes. If any of the receivers has already received a higher-numbered *Prepare(n'')*, it will reject the request, otherwise the node will accept the *Propose* message.

3. The proposer decides, based on the received acceptance and rejection messages, whether the group has agreed on a value. If a majority accepts, the proposer sends the message *Decide(n,v)* which informs all machines about the successful agreement.

We show a successful execution of Paxos in Figure 2.13 where after *Decide(n,v)* all three participants know that the agreed value is v.

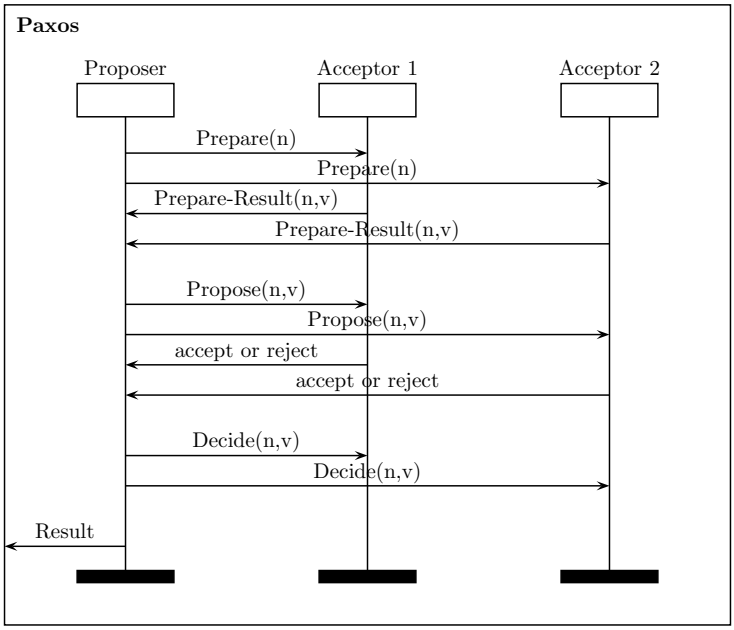


Figure 2.13: Paxos: messages exchanged between the proposer and two acceptors during the three phases

2.6 Rhizoma

Rhizoma [29] is a runtime environment for distributed applications targeting utility computing infrastructures like Amazon's EC2 [1]. It eliminates the need for a separate management machine and transforms the application into a *self-deploying* system. Constraint logic programming (CLP) is used to describe the constraints of the system. Rhizoma tries to optimize the deployment to meet the constraints and maximize the value of a given utility function.
Rhizoma's architecture is shown in Figure 2.14. Rhizoma includes an overlay network providing membership functions, leader election, failure detection, and communication. One overlay node is elected as coordinator and the others are members. Sensors periodically collect information about the environment

and report this to the reasoning engine. This collected data is stored in the coordinator's knowledge base. The reasoning engine periodically reassesses the deployment and initiates deployment changes when constraints are no longer met or a better deployment is possible. Actuators are responsible for creating and starting virtual machines. A separate actuator for each supported platform is used. An application can interact with Rhizoma through an API.
Rhizoma is built using the Python programming language [21] and relies heavily on event-based, asynchronous operation. It uses the Twisted networking engine [26]. At the time of writing, Rhizoma supports x86 and ARM-based linux machines, Amazon EC2 [1] and PlanetLab [20]. It has been successfully tested on all the mentioned utility computing providers, linux-based personal computers and Nokia's N810 Internet Tablets.

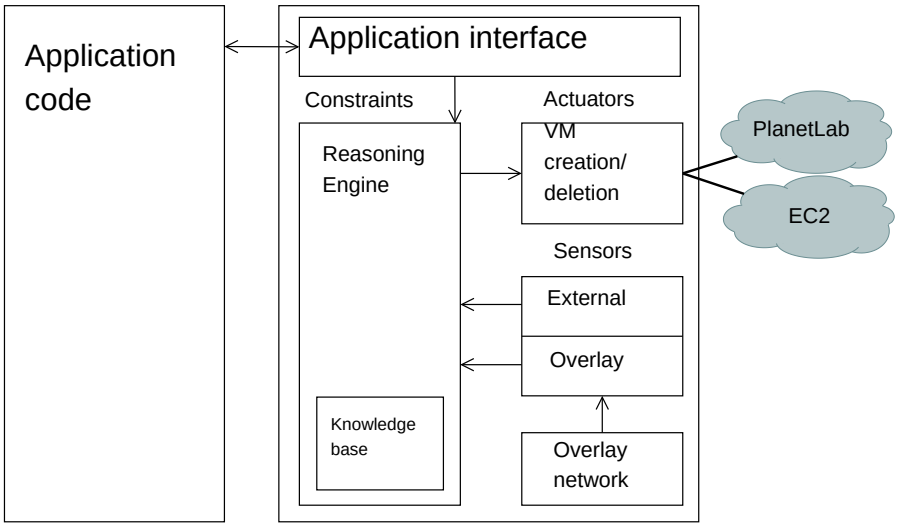


Figure 2.14: Rhizoma architecture

2.7 Related Work on Replication Systems

PRACTI [4] is a system that offers flexible data replication. It allows to balance the trade-offs between performance, consistency and availability. PRACTI offers partial replication, arbitrary consistency, and topology independence, and claims to be the only system offering all three properties. The system distinguishes between mechanism and policies and allows the exchange of the replication policy implementation without changing the replication mechanism. The units of replication are notifications about writes, the *invalidations* which are distributed to other replicas. A log is used to replicate these writes. The corresponding data is replicated separately from the invalidations, which allows different policies to choose between push- or pull-based approaches. The log exchange mechanism guarantees that the log content is always causally ordered. PRACTI offers eventual consistency: all nodes in the system will eventually agree on the order of all writes. Total ordering is achieved by using Golding's algorithm [8]. For tuneable consistency, the TACT framework [30] is used.

Bayou [19] is a weakly consistent replicated storage system in which updates can occur anywhere. These updates are propagated to all other replicas with Bayou's anti-entropy protocol for replica reconciliation. Writes are stored in a write log on each replica and the anti-entropy protocol allows two replicas to bring each other up-to-date, which enables them to agree on the set of writes stored in their logs. A write consists of a set of updates, a dependency check, and a merge procedure. Replicas receiving a write from an application, assign monotonically increasing logical timestamps, the accept-stamps, to these writes which define a total order over all writes accepted by a server and a partial order over all writes in the whole system. In order to stabilize and commit the writes, a primary-commit protocol is used, which assigns commit sequence numbers (CSN) to all committed writes. This stabilization step defines a total order between the writes, which in turn allows the propagation of these CSNs such that all replicas commit these writes in the same order.

The Google File System (GFS) [6] is a distributed file system for large files. Its goal is to provide fault-tolerance on commodity hardware under the assumption that hardware failures occur often. Files are stored in fixed-size chunks that are identified by a global chunk handle. Chunks are stored on the chunk servers as Linux files and replicated to multiple chunkservers for increased reliability. The system consists of a master and several chunkservers where the master only provides the metadata needed to find the chunkservers, and the chunkservers themselves handle all actual file operations. Clients only interact with the master for metadata operations, all other comminucation occurs directly with the chunkservers. A relaxed consistency model is provided by GFS where files can be in one of three states:

- consistent: all clients will always see the same data from any replica
- defined: if consistent and client sees what the mutation writes in its entirety
- inconsistent: if different clients may see different data at different times

GFS guarantees that after a series of mutations, the mutated file region will be defined. It achieves this by applying the mutations to a chunk in the same order on all its replicas, and using chunk version numbers to detect any stale replica. To guarantee a consistent ordering, the master will grant a chunk lease to one of the replicas, which will act as primary. This temporary primary will pick a serial order for all mutations to the chunk and propagate them to the other replicas (the secondaries). The data flow is decoupled from the control flow to increase efficiency by pushing data linearly along a chain of chunkservers.

Cimbiosys [22] is a replicated storage platform for loosely coupled systems that offers content-based partial replication among peers. Each node in the system can hold full or partial copies of one or more data collections. Each of them contains a set of items and every item contains an XML object and an optional associated file. A filter specifies which items are included in the device's replica by a selection predicate over the XML contents of an item. Local reads and updates are allowed. Each update provides a new version of an item that is propagated to other replicas. Joining the system happens by creating

an empty collection and synchronizing with some existing replicas. Cimbiosys describes no conflict resolution, but assumes that either an automatic model is available, or that manual resolution is needed. For every operation on an item a new version-id is created that uniquely identifies the item. The version-id is combined with the replica's identifier and a counter of the updates that occurred on that replica and associated to the item. Synchronization is achieved by maintaining so called *item-set knowledge* in each replica that records the set of versions that are known to that node. This describes the items that are known and version vectors with the known version counters and the replica ids. Replicas can exchange items via a one-way pull-style synchronization protocol where item-set knowledge and filters are exchanged. The system ensures that eventually the replicas will receive all items of interest.

Vertical Paxos [14] is an attempt to use the Paxos [13] consensus algorithm to replicate actual data, not only global configuration information. It achieves this by having four different roles: clients, learners, leaders and acceptors. Clients submit requests that are to be totally ordered. Learners are informed about each request and execute it, while only the leaders and acceptors participate in the execution of the protocol to reach consensus. Like the original Paxos algorithm, the concept of a quorum is used, where any two quorums of acceptors must be intersecting. The protocol runs by having leaders chosing ballots with unique numbers on which acceptors have to decide. An operation is attached to the ballot and, if a majority of the acceptors acknowledge this ballot, is chosen as the next command to be executed. Vertical Paxos generalizes the traditional Paxos by having configurations, which define sets of acceptors and the quorum structures, and allowing each ballot to be associated to a different configuration. By using configurable read and write quorums, it is effectively possible to run a primary-backup algorithm, where a node can read locally by having a read quorum of 1 and a write quorum of n (where n is the total number of acceptors).

Perspective [23] is a distributed file system offering a semantic view-based interface. It introduces views, which are descriptions of a set of files. These views allow to specify query-like rules to select files based on metadata properties and describe where these files shall be stored. The system ensures that eventually the set of files will be stored on the specified devices. Perspective's main target are home users with their devices and the system has been built according to results from user investigations on the preferred way to handle personal files. Synchronization happens by means of update propagation with an update log. Consistency guarantees are enforced by using version vectors and having timestamps for freshness guarantees about the currently stored files.

PacificA [15] is a replication framework for large-scale, log-based storage systems. It provides strong consistency and is targeted to local-area-network cluster environments. The configuration replica group management is separated from the data replication. Paxos is used for managing the configurations, while a primary-backup approach is chosen for the data replication. Failure detection is implemented in a decentralized manner and allows to trigger reconfigurations. Updates are held in a log, which is processed on the primary node. Log contents

are maintained as a prepared list which is divided into a committed list with a committed point. Each request is stored with an assigned serial number. This order, which is imposed by the primary, ensures strong consistency. Leases are used for failure detection.

Chain Replication for Supporting High Throughput and Availability [27] is a new approach for storage systems to offer availability and high throughput even in the presence of failures and configuration changes. Additionally, the chain-replication protocols offer strong consistency. Nodes that provide replication, are linearly arranged in a chain. Updates are directed to the head of the chain which processes it and forwards it to each node in the chain using reliable FIFO links. Reads are issued directly to the tail. This configuration ensures that each request, both reads and updates, are processed serially at the tail. If a server fails, it is removed and the chain is reconfigured. A master keeps track of all servers in the chain and informs them if changes to the chain have occured. This setting tolerates up to $n-1$ concurrently failing servers, if a total of n servers are used.

A Weak-Consistency Architecture for Distributed Information Services [8] describes an architecture for services provided on wide-area networks. Clients access these services by contacting any of the servers which replicate the service. Servers cooperate and communicate amongst themselves to exchange update information. Every server holds a copy of the database which is used to persist the data. Entries in the database have a unique key. The *time-stamped anti-entropy* group-communication protocol provides reliable, eventual delivery of messages. Messages are received by a process and written to a log together with a timestamp. Anti-entropy sessions between two servers allow them to exchange the log contents. This is accomplished by using summaries of the timestamps which describe what the server has already received. Different orderings can be achieved depending on the types of timestamps used. By the additional use of an acknowledgment timestamp vector, which records what messages have been received by other servers, each server can determine which log entries can be written to the database and discarded from the log. This approach is used in the PRACTI system [4] for the write commitment.

Chapter 3

Design and Architecture

We have created a flexible replication environment that addresses the requirements of applications running in Personal Clouds. This chapter gives an overview on the system's architecture and explains how data is replicated. We also explain how the problem of partial replication is addressed and introduce the system's consistency model. We conclude this Chapter with a discussion of our system's replica management and its integration into the Rhizoma runtime environment.

3.1 System Architecture

3.1.1 Overview

Like the PRACTI [4] system, our system consists of two main modules: a core and a controller. Additional modules include the communication, consensus and persistent storage. The consensus module is special in that it is built modularly such that a controller can use different consensus protocols. In our current implementation we use Paxos [13] as consensus protocol. Figure 3.1 gives a high-level overview of the parts.

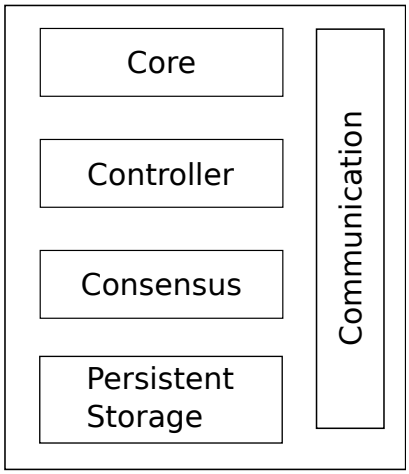


Figure 3.1: Node architecture

Core and controller fulfill two complementary roles. While the core's responsibility is to provide the replication mechanism, which allows to send and receive data and control messages, the controller ensures that replication policies are enforced by deciding when and to which nodes messages shall be sent and what the reaction to received messages is. This separation provides the advantage that the controller can be replaced with an implementation of a totally different replication policy without having to reimplement the system's code base. The following sections provide detailed explanations of all parts of the system.

3.1.2 Core

The core is responsible for the mechanism of data replication, which means that it knows *how* to deal with local reads and writes and *how* to send and receive information and data to and from other nodes. A local API is exposed that allows applications to locally create, read, write, and delete data items. Information about writes is held in the *Log*, while the actual data belonging to a write is stored in the *Store*. Figure 3.2 shows an overview of the parts contained within the core.
The Log holds information about known writes. Information about writes is stored in so-called *invalidations*, the notifications about writes. Invalidations can be stored in the Log and retrieved from it in a causally consistent way. Furthermore, the Log allows to reorder the invalidations. It also plays an important role for partial replication by allowing to efficiently retrieve only subsets of the whole content. The MemStore, on the other hand, allows to store the values for these invalidations. The values are embedded in so-called *bodies*. The details about invalidations and bodies will be explained in Section 3.2.

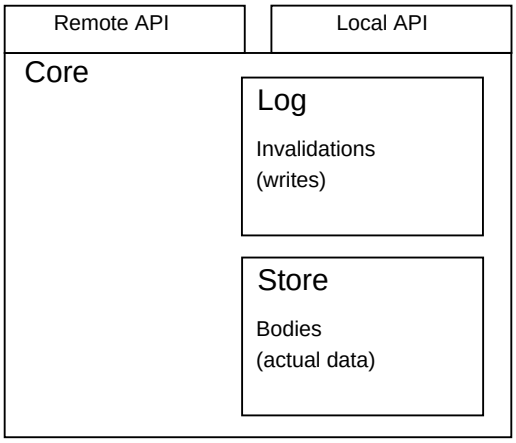


Figure 3.2: Core with the Log, Store and the Local and Remote API

Local writes When a local write is submitted at the *Local API*, the core creates an invalidation that describes the write and stores it in the Log. It also creates the corresponding body, which contains the written value, and stores it in the Store.

Remote writes A remote write is characterized by the arrival of an invalidation through the *Remote API*. When such an invalidation is received, it is stored in the Log like in the case of a local write. Additionally, the core sets the flag for the corresponding entry in the Store to invalid. As soon as the body arrives, the flag is set to *valid*, signifying that the value for an invalidation is available, and the value is stored in the Store.

Reads Reads are always executed locally through the *Local API*. When a read is submitted, the core first checks whether the requested item is contained in the Log. If an invalidation which corresponds to the requested item is found, the matching value is retrieved from the Store. An invalid entry in the Store means that the body has not yet been received. The body may be missing for two reasons: the arrival of the body is delayed, or the policy in use expects the bodies to be fetched on demand. The API offers a flag for the read, to specify whether blocking or non-blocking operation is to be used. In the non-blocking case, the read will fail if no body is available, even if a matching invalidation is contained in the Log. In the blocking case, the read will return as soon as the body has arrived, possibly starting a demand fetch.

Additionally, a **create** and a **delete** operation is offered. A create sets up the state to allow subsequent writes. A delete is a special write, which is handled in exactly the same way, except that no body is created.
Intertwined with the mentioned operations, the core notifies the controller about each event that happened. In particular, when a request is received either from the local API or from the remote API, the core calls the corresponding notification methods from the controller, to ensure that the controller always knows the state of each operation.

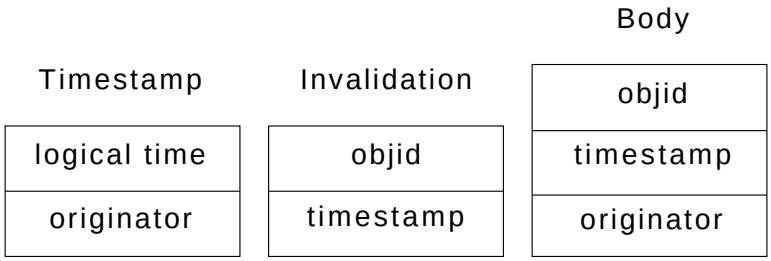
3.1.3 Controller

The controller is responsible for the enforcement of the replication policy. It decides *what* has to be done and *when* it is supposed to happen. The controller exposes a notification interface which is invoked by the core whenever an action is executed. In response to these notification calls, the controller determines which actions are needed according to the implemented replication policy. When a local write occurs, the controller determines which nodes need to receive the corresponding invalidation and initiates the sending of them. It also determines whether the bodies belonging to the invalidations must also be sent and initiates the sending accordingly. Furthermore, the controller holds the responsibility to initiate and execute any concensus protocols that might be needed to enforce the consistency guarantees. In our implementation the controller uses the Paxos consensus algorithm to agree on the order of the writes that can be committed. When consensus has been reached, the controller initiates the reordering and subsequently commits the Log contents to the persistent storage.

3.2 Data Replication

Our system replicates writes which are described with invalidations. As opposed to other systems like Bayou [19], our writes do not carry operations but comprise

the information that a write has occurred. Such a write is represented as an invalidation and contains a string identifier and a timestamp that describes the logical time of the occurrence of the write together with the node where the write originated from. The corresponding value of the write is represented differently with a body that embeds it. Figure 3.3 shows a graphical representation of the objects involved in the representation of a write.



(a) Timestamp (b) Invalidation (c) Body

Figure 3.3: Representation of a write

3.2.1 Separation of Invalidations and Bodies

The separation of information about writes (invalidations) and the written data (bodies) provides several advantages. Invalidations are a compact way to represent that a write has occurred and can quickly be disseminated to other nodes, while the actual data can potentially be very large and therefore impose a large transmission delay. The body-invalidation separation allows to specify policies where the information about a write and the actual data are transmitted through different links. Furthermore, by separating the bodies from the invalidations, we accomplish partial replication of the data. A replica can receive invalidations without the corresponding bodies with the data. This separation also provides benefits for the replication policies that can be supported.
Our system allows different replication policies to be implemented with controllers. The currently implemented controller employs both a push and a demand fetch policy, based on the specific interests and subscriptions of the individual nodes. One possible scenario, shown in Figure 3.4, would be to transmit invalidations through a low-latency link, while the bodies are transmitted through a high bandwidth link which might offer higher latency or a lower cost. A similar approach is used in the Google File System [6], where the dataflow is separated from the control flow to increase efficiency.

3.2.2 Imprecise Invalidations

We have implemented imprecise invalidations to enable partial replication of invalidations. Instead of receiving, storing and processing all invalidations describing all writes in the system, a node can receive an imprecise invalidation that summarizes the information of several *normal* or *precise* invalidations. An imprecise invalidation has a target set denoting a set of items to which the imprecise invalidation applies, and a start and end version vector. The meaning of such an imprecise invalidation is that something happened to the objects in the

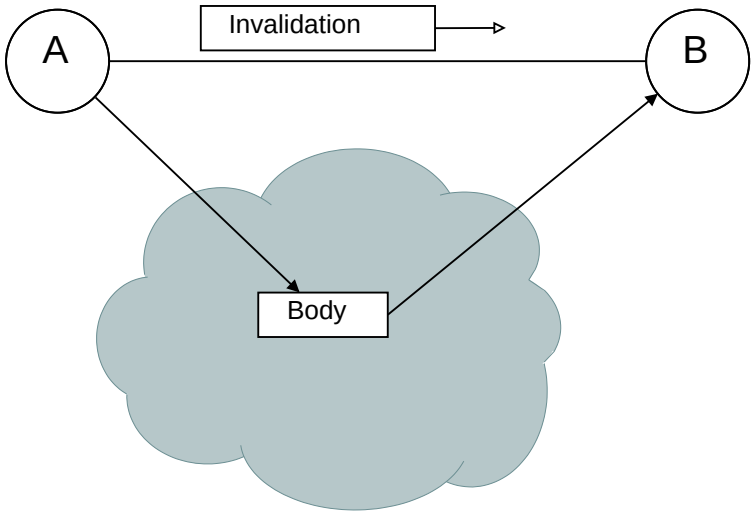
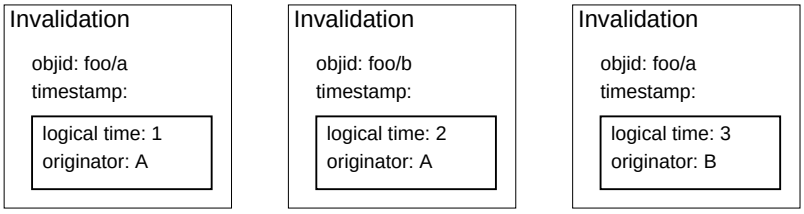


Figure 3.4: Separation of the invalidation stream and the bodies

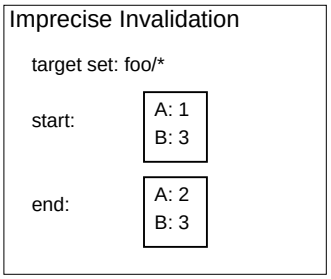
target set within the logical start and end version vector times. These imprecise invalidations are used to summarize all the writes that occurred to items in which a replica has no interest. After receiving such an imprecise invalidation, the replica knows that some writes happened, but since it is not interested it the items, no further information is needed. In particular, imprecise invalidations have no corresponding bodies.
Figure 3.5(a) shows three precise invalidations that describe writes to two different items *foo/a* and *foo/b* from two originating nodes A and B, respectively. The imprecise invalidation shown in Figure 3.5(b) shows these three invalidations in their summarized format.
Imprecise invalidations can be made precise after the arrival of every precise invalidation covering the whole set of the imprecise invalidations.

3.2.3 Invalidation Streams

To distribute the invalidations to other nodes, the system uses so-called *invalidation streams*, which guarantee ordered delivery of the invalidations. These invalidation streams distribute the invalidations in the same order as they are contained in the sending node's Log. The order is preserved and the invalidations are inserted into the Log of the receiving node. Section 3.4.1 discusses the detailed ordering guarantees that are ensured by the use of the invalidation streams.
Each invalidation stream starts with a version vector that denotes the logical version vector time just prior to the first invalidation contained in the stream. In Figure 3.6(a) a conceptual model is shown. Figures 3.6(b) and 3.6(c) depict how the previously shown invalidations are propagated with invalidation streams, both in the precise and the imprecise case. Version vectors and Lamport's logical clocks have been explained in Sections 2.1.1 and 2.1.2, respectively.
The invalidations sent through the invalidation stream satisfy the ordering constraints imposed by the system.



(a) Precise invalidations



(b) Imprecise invalidation

Figure 3.5: Representation of several precise invalidations with an imprecise one

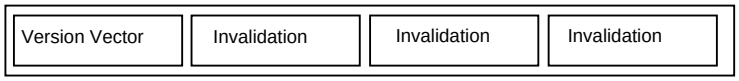
- Invalidations that originated on the sending node are streamed in a sequentially consistent way.

- Invalidations from other nodes are streamed in a causally consistents way.

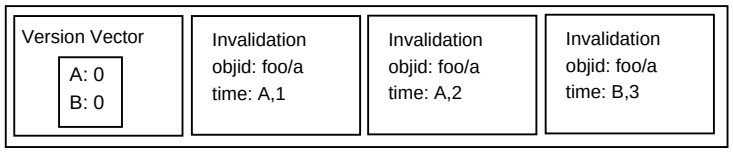
Both, the locally created and the received invalidations, are intertwined, and the resulting order of these invalidations is causally consistent.
In Figure 3.7, an example of the contents and the order in the invalidation stream is shown. A, B and C are individual processes with writes w_{xi} where x denotes the originating process and i the logical timestamp of the write. The example shows the invalidation stream sent from B to C after B has received all information about the writes from process A. In Figure 3.7(a) the setting of the example is shown. Figure 3.7(b) then shows the causal relationships between the writes in the example, and Figure 3.7(c) depicts one possible causally consistent ordering in the invalidation stream. It is important to note that the shown invalidation stream is only one possibility among several that obey the rules of causal consistency (as explained in Section 2.4).

3.3 Interest Sets, Rules and Filters

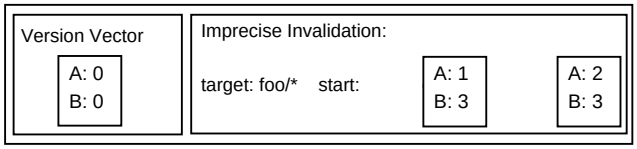
Several possible ways are offered to specify the items of interest. Each of these possibilities allows a node to declare the desire to receive full information about these items. Items that are not contained in these interest specifications are only received partially, which means a node will only receive imprecise invalidations summarizing the changes that occurred.



(a) Conceptual model of an invalidation stream



(b) Precise invalidations from previous example



(c) Imprecise invalidations from previous example

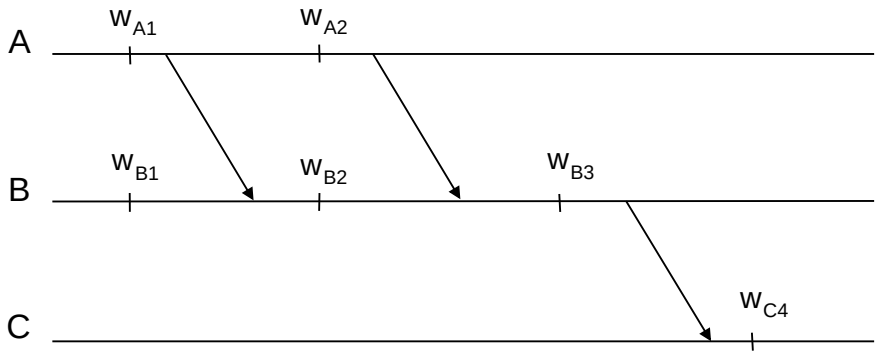
Figure 3.6: Invalidation streams

Item key	Matching interest	Replicated
music/abcd	music/abcd	yes
music/foo	None	no
policies/ab1	policies/*	yes
texts/example	None	no
photos/photo1	photos/*	yes
photos/photo2	photos/*	yes

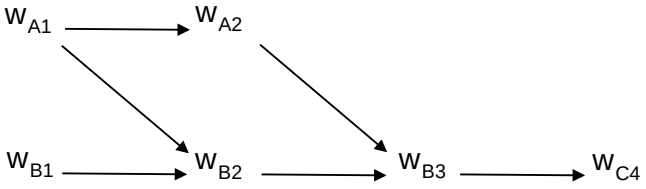
Table 3.1: Items and matching interests for IS = {*"photos/*"*, *"policies/*"*, *"music/abcd"*}

3.3.1 Interest Sets

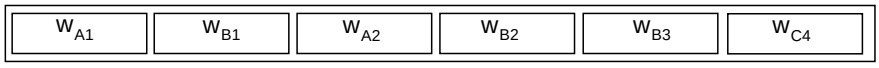
An interest set is a set of strings where each of these strings specifies a set of items by matching the item's key. This is accomplished by introducing a directory-like hierarchical key structure similar to the target set of imprecise invalidations. Interest sets are used to concisely define the subset of all items in which a replica is interested.
An example interest set might be defined as follows: {*"photos/*"*, *"policies/*"*, *"music/abcd"*} . A star denotes a wilcard, where every item with a key that has the same prefix matches the interest. Assuming the system consists of the items *music/abcd*, *music/foo*, *policies/ab1*, *texts/example*, *photos/photo1*, and *photos/photo2*, Table 3.1 shows which interest matches their item keys and whether they are replicated or not.
Every node specifies its interest set and accompanies its subscription to other replicas with its interest set definition. Each replica will maintain this interest set for each subscribed replica and propagate items accordingly. For full replication of all items, a replica can specify the interest *"*"*. Items that match any of the specified interests will be replicated with precise invalidations. All other



(a) Setting of the example



(b) Causal relations between the writes



(c) Possible invalidation stream

Figure 3.7: Invalidation stream example

writes that occurred since the last invalidations have been sent, but for which no interest exists, are summarized as an imprecise invalidation.

3.3.2 Rules and Filters

To allow for higher-level specifications of item interests, rules and filters can be specified. Rules are evaluated within the Rhizoma system and allow to take environment information into account. Example rules might be:

- replicate item *texts/thesis* on at least 3 replicas, or
- replicate item *music/foo* on one replica with a maximum access latency of 100 ms.

Such rules can be fully evaluated in the Rhizoma runtime, while the data replication system supplies the necessary information on replica distribution and offers an API that allows to initiate changes in the replica configuration.
More sophisticated rules, that involve filtering on the item metadata instead of only specifying item keys, are also possible. Our system allows to store metadata for the individual items and offers the possibility to specify rules that involve filtering on the item metadata. Example rules with filters might be:

- *do not store private data in the cloud*, or

- *replicate all music with a five star rating to my mobile phone.*

These rules could be specified with filters on the metadata, where every item with a tag *private* would match the filter and not be placed on machines that are not owned by the user and item with type *music* and a five start rating would only be replicated to the mobile phone. The Rhizoma system also allows to use device-specific properties, e.g., the current status of CPU load and memory, but also network properties like latency.

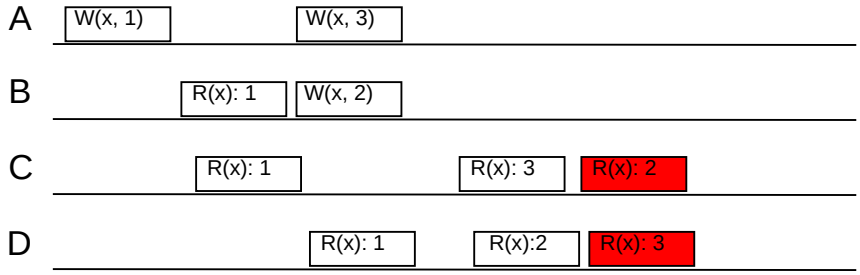
3.4 Consistency and Order Agreement

3.4.1 Consistency

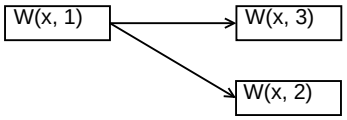
Each node maintains writes that occurred locally on that node in a sequential order and ensures a total order on the writes that originated from this node. The result of a write executed on a node can immediately be read on the same node, i.e., read-your-writes consistency is offered.
Invalidations describing the writes are distributed in the same order as they are stored in the local Log. The invalidation streams then ensure two essential properties:

1. all invalidations in an invalidation stream are received in the same order as they were sent, and
2. a node will always get a prefix of all writes that occurred in the whole system.

The first property ensures that the order of the invalidations is maintained during transmission. A receiving node can be sure that all writes from any specific originator will be received in exactly the same order as they have been sent. This is achieved by using a TCP stream which provides reliable and ordered delivery.
The prefix property is ensured by the first entry in the invalidation stream that holds a version vector of the state immediately prior to the logical vector time when the first invalidation happened. The receiving node maintains the known version vector state and can compare these two version vectors to detect gaps. If gaps are detected, the invalidation stream (with the contained invalidations) is discarded.
Following from the second property, we can deduce that a receiving node must know all writes that happened prior to the received writes.
By using timestamps that obey the rules of Lamport's clock [11] in the invalidations and by using version vectors denoting the known vector time state at the time when the invalidation has been created, the invalidations corresponding to writes that occurred on other nodes can be inserted into the local Log in a causally consistent way. This leads to a Log that is causally consistent (an introduction to causal consistency has been given in Section 2.4).
But, as already discussed in the background chapter, causal consistency can still lead to inconsistencies that are not necessarily desired. Figure 3.8(a) shows a possible execution that is correct within the rules of causal consistency. Still, the last value of the written item x is different on node C as compared to the value that would be read on node D. The causal relations between the writes of this example run are shown in Figure 3.8(b).



(a) Example execution. *W(k, v)* denotes a write to item k with value v, *R(k): v* denotes a read of item k that returns value v



(b) Causal relations between the writes

Figure 3.8: Example of a causally consistent execution where two nodes read different values

3.4.2 Order Agreement

Writes that are not causally related may be observed in different orders by different nodes. The causal consistency is still maintained, however, to ensure that each node also agrees on the same ordering of the writes that are not causally related but interfere, the Paxos [13] consensus algorithm is used. With Paxos, the replicas of this item will agree on one ordering of the causally unrelated writes to the item, thereby ensuring that after consensus has been reached, a read on any node will return the same value or a newer value, but never an older one. Figure 3.9 shows a possible execution that results from running Paxos to agree on the order of the writes that are not causally related.

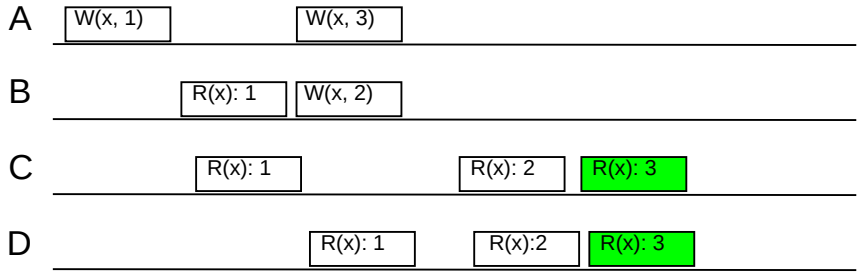


Figure 3.9: Execution after Paxos consensus on the order is reached.

By agreeing on the order of the writes for which no causal relation has been observed, we effectively transform the causally consistent ordering to an ordering obeying the rules of sequential consistency. Note that this does not imply that every read will return the value that is obtained from a sequentially consistent ordering, because we allow reads to be executed on tentative parts of the Log. A read could read a *newer* value than the one produced by the last agreed write,

because new writes could have been added.

3.4.3 Flexible Consistency and Conits

We offer flexible consistency through the concept of conit-based continuous consistency [30]. A *conit* is a unit of consistency that allows to specify and reason about the consistency of the items within the conit. Each item belongs to at least one conit, and for each conit we keep track of the current order deviation, which is the number of writes that occurred on the items within the conit without synchronizing with other nodes.
Our implementation allows conits, the units of consistency, to be defined on the whole data store, on interest sets, or on individual items. The choice of the definition is not exclusive, i.e., it is possible to define both the whole data store as one conit, and an interest set as another one. By using this definition of conits, it is possible to attach properties to consistency units and react in case the system changes and these properties are not valid anymore. We use numerical bounds that are attached to the conits. These numerical bounds express the maximum order deviation of all items within the conit which can be tolerated without resynchronization, i.e. a commit. The order deviation is defined as the number of writes that occurred on the conit set.

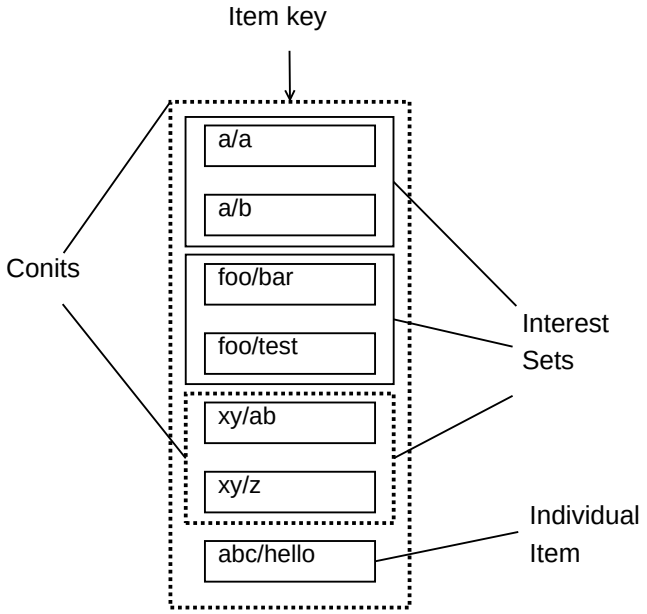


Figure 3.10: Conits

Yu and Vahdat's approach [30] also includes numerical error, which measures the error of the value to be stored, and staleness, which is the time from the last resynchronization. We have not implemented numerical error, because we replicate the actual data and not operations that change the value by a certain amount. Our expected values are not known, but it would be simple to add numerical error to the system if a measure of it could be extracted from the values. One possibility would be to bound the maximum change in the size of the

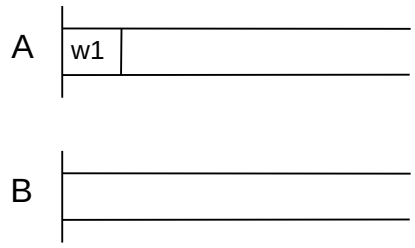
value. Staleness has also not been implemented, but it would be straightforward to add it.
In Figure 3.10 a simplified data store is shown. Two conits are presented: the conit covering the whole data store, and the one covering only the interest $xy/*$, which contains the items xy/z and xz/ab. Our system allows to specify order error bounds on these two conits which – if exceeded – trigger a commit phase where the outstanding writes will be committed in a sequentially consistent order. Additionally, we have added bounds for the sending of invalidations which can be lower than the bounds for the sequential consistency.
These two concepts allow us to tune our consistency depending on application's needs.

Send bounds allow to bound the number of outstanding writes before a dissemination of invalidations is triggered. This effectively defines how much potential causality we require. Note that causal ordering is achieved by using version vectors and the invalidation streams. Therefore, a deferred distribution of the invalidations will decrease the dependencies between the writes and lead to more concurrent writes.

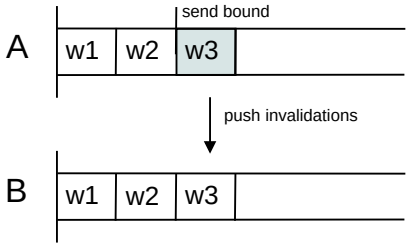
Consistency bounds limit the number of tentative writes until our agreement algorithm is used to agree on a sequential order. This bound limits the maximum amount of inconsistency that is tolerated.
Send and consistency bounds can be configured on startup, changed at runtime, or even refined when executing read or write operations, thus allowing us to balance the desired level of causal and sequential consistency. High bounds imply less communication but also that the probability of stale or inconsistent reads increases accordingly. On the other hand, low bounds increase the needed communication but leave less probability for stale or inconsistent data.
Figure 3.11 shows an example execution with two nodes A and B and their Log contents at different times. Node A executes a total of 5 writes. The send bound is set to 2, while the consistency bound is set to 4. In Figure 3.11(a), the setting after a first write *w1* is shown. The write is stored in node A's Log. Neither the send nor the consistency bounds have been exceeded, so the information about this write is not yet propagated to node B. The situation after the send bound has been exceeded by the execution of the third write on node A is shown in Figure 3.11(b). Node B has now received all writes that originated on node A. The final Figure 3.11(c) shows the situation after the consistency bound has been exceeded. In this situation, our system attempts to reach consensus on the order of the writes that have to be committed. Subsequently, the contents of the Log are processed in the agreed order and removed.

3.4.4 Replica Set Consistency

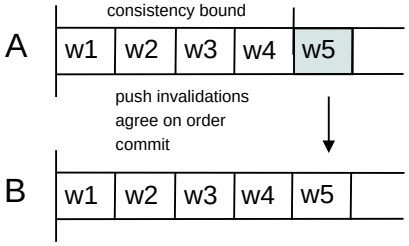
Several possibilities have been analyzed on how replica sets must be handled to allow for flexible operation but still ensure the above mentioned consistency guarantees. We have identified three different approaches that all have their advantages and disadvantages.



(a) Initial setting: write *w1* has been executed on node *A*



(b) The send bound is exceeded: the writes are pushed to node *B*



(c) The consistency bound is exceeded: the writes are pushed and a consensus run is initiated

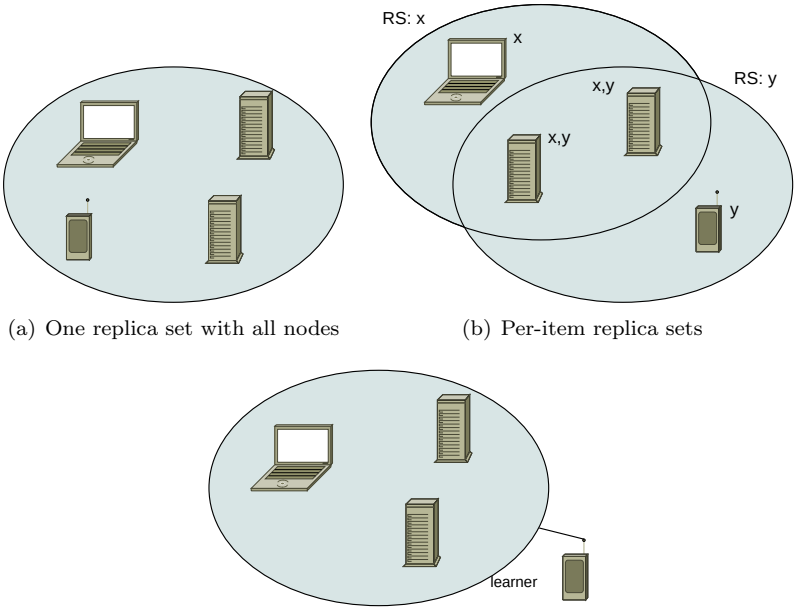
Figure 3.11: Conit bounds example: the send bound is set to 2, the consistency bound to 4

All nodes as one replica set: One possibility is to use all nodes in the system and define one replica set for all. This scenario is shown in Figure 3.12(a). This has the advantage of being the simplest solution and closely resembles the assumptions made by PRACTI [4]. It is also straightforward to apply the concept of conit-based consistency to it: these conits would be defined on one global replica set holding all writes. The main disadvantage is that all nodes have to participate in the consensus and are relatively tightly coupled.

Individual replica sets for all items: Figure 3.12(b) depicts a possible approach where the items x and y are stored in different, overlapping replica sets. This approach allows us to have partial replication of the agreement protocol instance where only the nodes that are involved with the item have to participate in the consensus protocol. The main advantage is that this approach allows for flexible coupling, allowing different nodes to run their own agreement protocols. Also, in such an approach one could potentially use different agreement protocols for different replica sets, possibly exploiting characteristics of the devices contained in each replica set. Such an approach, however, complicates the processing of writes and the enforcement of the consistency guarantees, which are already provided by the basic replication mechanisms.

One replica set with learners: In Figure 3.12(c) a scenario is shown where 3 nodes are actively participating as members of the replica set. In the case of Paxos these would be two *acceptors* and one *proposer*. The devices outside the replica set are so-called *learners*. They get to know all updates but do not

participate in the agreement process. They are eventually informed about the outcome of any agreement. The advantage of having learners is that devices with limited computing capabilities like mobile phones do not have the burden of participating in the consensus protocol but are still informed about the consensus outcome. The savings for the resource-constrained devices are potentially large, but only if the learner's interest set is defined concisely and the replication policy avoids strong consistency by specifying high consistency bounds. Such a setting would minimize the communication that informs the learner about successfully reached consensus on the ordering. An obvious disadvantage of this approach is that a learner cannot initiate runs of the consensus protocol.



(a) One replica set with all nodes

(b) Per-item replica sets

(c) One replica set with learners

Figure 3.12: Replica sets for agreement

While the decision which of the above approaches to take is a question of policy – therefore a question of how the controller shall be implemented – the implications of having only one or several replica sets that are used to establish an agreement has implications on the core. When the replicas agree on an order of the writes, the controller must inform the core and the core then adapts the Log contents accordingly. Having more than one replica set vastly complicates this Log adaptation. In case of only one replica set, the controller can ensure that agreement with respect to ordering is always reached on all invalidations in the Log. In the case of several replica sets, the agreed order could contain invalidations that are arbitrarily distributed in the Log, which would require a more complex commit operation in the core and complicate the handling of imprecise invalidations. A detailed description of the Log handling will be provided in the next Chapter in Section 4.2.4. It is important to note that the meaning of replica sets applies only to the replication policy that is implemented by the controller. The mechanism of partial replication is not affected by the number

of replica sets.

3.5 Replica Management

New replicas can be started or existing replicas can be shut down. This task, in general, is independent from the stored data. Rhizoma is responsible for ensuring that:

1. a sufficient number of replicas which together hold at least one copy of the data are running,

2. new replicas are started properly and initialized correctly. The existing replicas are informed about this system change,

3. a data snapshot of the whole persisted data store is copied to this new replica, and

4. the new replica starts operation once it is correctly set up.

Furthermore, Rhizoma's features allow to specify rules for the replicas, for example the maximum latency between two nodes which can trigger new replicas being started or existing replicas being moved, copied, or discarded completely.

3.5.1 Roles of Replicas

As discussed in Section 3.4.4, there are several possibilities to set up the replica set. In our current implementation, we have chosen to have one replica set and differentiate between the following three roles, which are inspired by Paxos' notion of roles and its use for consistency enforcement. However, the other two replica configurations could be easily supported by extending the core's commit Log operations. The three components of a replica set are:

Proposers: they lead the Paxos agreement protocol. A proposer initiates and executes the agreement and is responsible for its completion. Our system assumes that a proposer is interested in every item and thus has all needed information.

Acceptors: they participate in the Paxos execution, but take a more passive role than proposers. They follow the lead of the proposer and object to the agreement only if a different proposer has submitted a different ordering for agreement.

Learners: they are actively *informed* about the outcome of every reached order agreement. Our assumption is that they will typically participate as replicas with only partial replication. Typically they will be personal computers, mobile phones or similar devices, which might be resource-constrained or only temporarily connected.

3.5.2 Startup

Replicas can independently be started and the mechanisms to add new nodes are flexible enough to support many different usage scenarios. When replicas are started, they need to be *wired together*, which means they need to find the other replicas to subscribe to their invalidation streams. Additionally, a replica that wants to actively participate in the agreement phase and therefore assumes either the role of a proposer or acceptor, must join the replica set running the concensus algorithm. This *wiring* is initiated by Rhizoma, which is responsible for the optimal placement of replicas.
In the case of a fresh startup, this task is relatively straightforward. Each replica subscribes to other replicas of interest and the consensus algorithm establishes a replica set for the agreement. The setting is more complicated if a new replica wants to join an existing replica set. Before becoming an active new member, the following steps must be undertaken.

1. The replica must start and acquire the knowledge of the current replicas in the system.
2. It must fetch a snapshot of the current committed and persistently stored state from one of the other replicas, preferably the proposer or alternatively an acceptor.
3. After the snapshot has been fetched, the valid current version vector state must be reconstructed.
4. As soon as the persisted data and the state has been recovered, the replica can start its operation as a new replica.

3.5.3 Copy Operation

To support copying of replicas, the replicated storage module offers the functionality to transfer the full state of one replica to a new one. This state transfer includes the content of the persistent data store and the currently valid state description. Additionally, in order to receive the persisted data and state, the new replica receives sufficient information to initiate subscriptions to the other replicas in the system. The subscription ensures that all new invalidations and bodies are received. The new replica also joins Paxos' replica set, which ensures that the replica is informed about successful ordering consensus.
In addition to a full copy, the operation allows to specify only parts of the data store content to be copied to a new replica. The part to be copied is again specified by using interest sets which have been introduced in Section 3.3.1. To allow rule-based copying, the system allows to initiate fetching of individual items, which can be triggered by rule-based changes to the partial replication.

3.6 Rhizoma Integration

The integration layer (shown in Figure 3.13) used for the interaction of the storage system and Rhizoma consists of three parts:

1. the Storage Sensor which is responsible for replicating item metadata,

2. the Storage Optimizer, which is in charge of interacting the Rhizoma CLP solver, and

3. the Storage Executor which provides various actions needed to actualize the output plan of the Rhizoma CLP solver.

The Storage Sensor replicate a user's replication policies, item metadata, and item location information across the Rhizoma's overlay. Item location information means on which nodes a specific item is currently stored. All this data is fully replicated to all other nodes, typically with high consistency requirements. The StorageSensor simply gathers the data and then writes them using the core's Local API. When the coordinator receives this information it stores it in Rhizoma's knowledgebase and it is then used by the CLP solver for optimization.
The Storage Optimizer evaluates the policies and outputs an execution plan consisting of various actions. The Storage Executor is responsible to enforce these actions in the system by interacting with the storage system.
Finally, communication between the nodes is conducted using Rhizoma's overlay. Paxos messages, invalidations, bodies, etc. are sent using the overlay API and benefit from Rhizoma's optimized communication module. For instance, best paths from node to node are constantly computed (based on a number of QoS metrics), multi-hop heterogeneous paths are established, and device failures are constantly monitored.

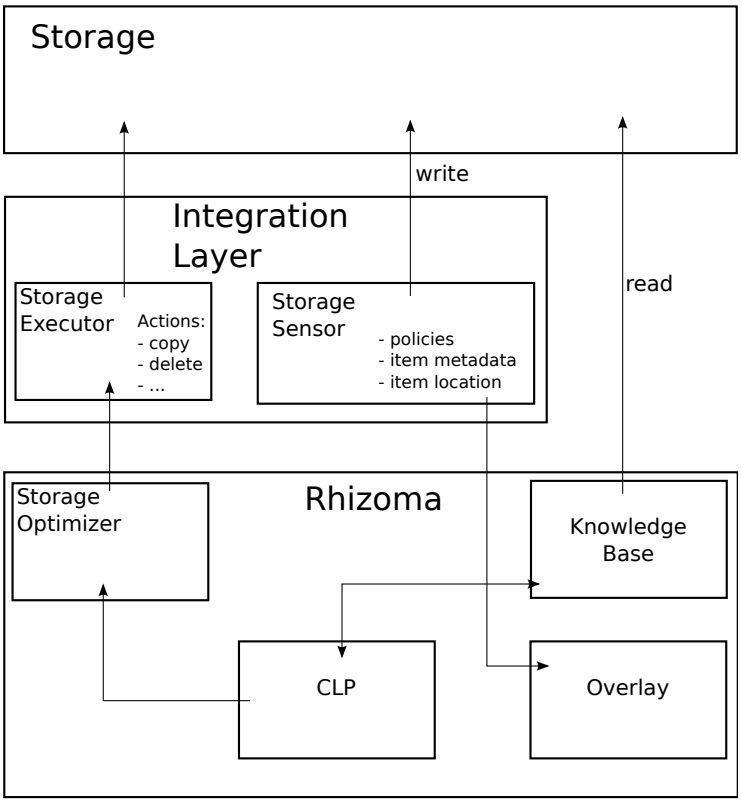


Figure 3.13: Rhizoma integration

Chapter 4

Implementation

4.1 Implementation Overview

The replication system is implemented using the Python programming language [21] and consists of around 10'000 lines of code (including tests and set up code, but excluding comments and blank lines). We are using the Twisted Networking Engine [26] for all network communication and rely on Twisted's *Deferred* to offer asynchronous operation. Modularity is offered by the use of POSGi, which is a module system for Python, similar to OSGi [18] for Java. Modules interact through interfaces which allow the implementations to be easily replaced. Interfaces between the different parts are defined and shown in the appendix in Chapter B. A detailed overview of the individual modules and their interactions is given in Figure 4.1.

4.2 Read and Write Processing

Handling local and remote writes is central for the correct functioning of the replicated datastore. To ensure the desired consistency properties, but also to ensure that partial replication does not break them, is a central objective. As explained in the previous chapter, our system replicates writes which are represented as invalidations, and bodies that contain the value of the write. The following subsection explain how our implementation processes these invalidations.

4.2.1 Local Writes

Writes arriving via the Local API are directly handled in the core's *write()* method. The method is responsible for creating the corresponding invalidations and bodies, and inserting them into the system's Log and Store. This is the only way to create precise invalidations. After the write, invalidations are immutable and therefore never altered again.
To ensure the correct insertion of a write's invalidation and body, *write()* executes the following operations.

1. It updates the global conit state, which is defined over all items. Potentially, it also updates the conit state associated with a subset of items.

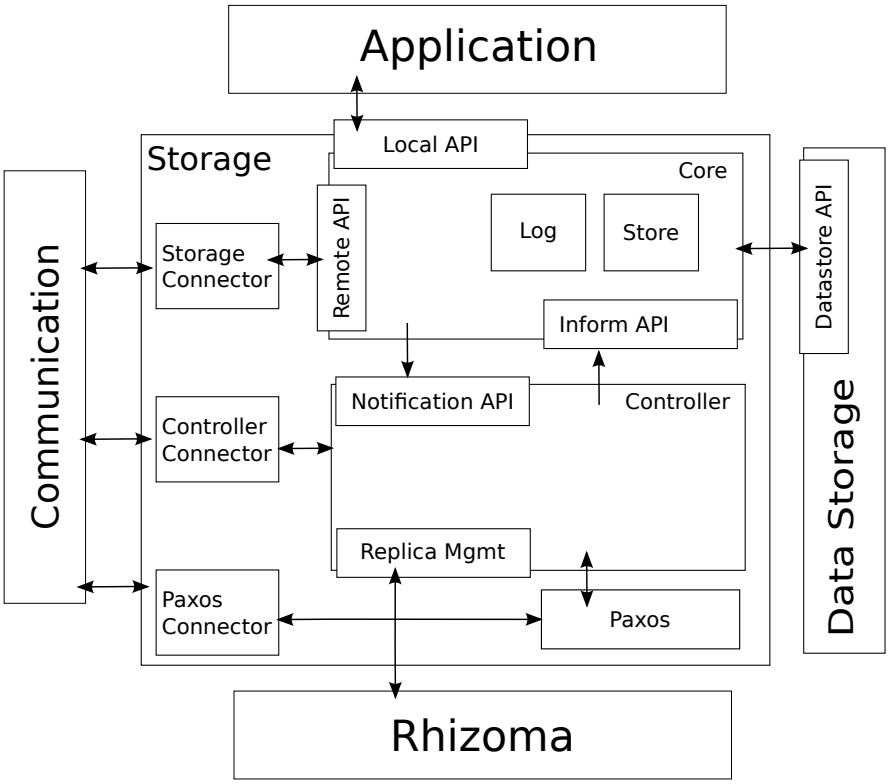


Figure 4.1: Detailed module overview

2. It determines whether send or consistency bounds have been exceeded.
3. It updates the node's local version vector state and extracts a new timestamp obeying the rules of Lamport's clocks.
4. It creates body and invalidation. The body contains the value of the write while the invalidation contains the previously extracted timestamp, an identifier of the current node, and the item key.
5. It updates information about the last precise invalidation for all currently precise interest sets.
6. It inserts the invalidation into the Log together with the version vector that was valid immediately before this write was received.
7. It inserts the body corresponding to the invalidation into the Store.
8. It informs the controller about the local write and whether any bounds have been exceeded.

The core's direct responsibility for this local write ends after the controller has been informed. Since the controller implements the replication policy, it will decide what must be done after this local write. The core guarantees that the created invalidation and body are correctly inserted into the Log and Store and that the core's local state is correctly updated.

4.2.2 Remote Writes

The handling of received invalidations is more complicated. A node has to handle invalidation streams from different replicas and must ensure that the received invalidations are correctly inserted into the core's Log.
To successfully receive and process the invalidations, the first step is to set up the invalidation stream state. The version vector, which is the first element of every invalidation stream, must be processed and the local state of this stream must be set up. An invalidation stream is only processed if the starting version vector of the stream denotes a logical version vector time that the receiving node knows, i.e., the node has already received invalidations covering all version vector times up to the received one. This is required to prevent gaps in the invalidations that are received. The invalidation stream ensures that the invalidations following the version vector have no gaps.

Invalidation stream setup When a new invalidation stream is received, the core's *handleVersionVector()* method is invoked. This method takes the version vector and an identifier of the sending replica as inputs. The following steps are then executed for the stream state to be set up.

1. The version vector is compared to the node's local current version vector time. To ensure that no gaps can occur in the processing, the received version vector must denote a logical time that precedes or is equal to the current logical version vector time. If this check fails, the handling of the stream is aborted.

2. The version vector is stored. It denotes the current logical version vector time just prior to the first invalidation in the stream.

3. In a final step, the core notifies the controller about the received version vector.

Processing the invalidations Each invalidation in the invalidation stream must be inserted into the Log and the Store which requires steps similar to the processing of local writes. The fact that both precise and imprecise invalidations can be received significantly complicates the processing.
Whenever a new invalidation is received from the invalidation stream, it is dispatched to the core's *handleInvalidation()* method. It's inputs are sender and invalidation. The method closely resembles *ProcessInvalStream()* of the PRACTI system.
The operations that are executed for every invalidation are the following.

1. The stream state is retrieved. It always holds the version vector of the logical time immediately prior to when the current invalidation was received.

2. This retrieved version vector is advanced to reflect the start time of the received invalidation.[1] It will serve as the start version vector for the next invalidation in the stream.

[1] Note that the start time and end time can differ if the invalidations are imprecise. In the precise case, the start and end times are the same.

3. The received invalidation is inserted into the Log together with the version vector of the logical time immediately prior to the invalidation. How exactly the invalidation is stored in the Log will be explained in Section 4.2.4.

4. To represent all known invalidations, the node's known version vector state is advanced to include the invalidation's end time. This known version vector state always represents the highest version vector state assembled from all invalidations that are known to the node.

5. If the received invalidation is precise, the local interest set state, to which the invalidation applies, is updated to reflect this invalidation as the last received precise invalidation. If the invalidation is imprecise, its last known precise interest set status will only be advanced to include the start time of the invalidation. The state of all other precise interest sets are advanced in both cases. An interest set is precise, if no imprecise invalidations matching it have been received, i.e., the last known precise version vector is equal to the node's known state version vector.

6. In the Store, the entry for the received invalidation is set to *invalid*. It will become *valid* as soon as the corresponding body is received.

7. To be able to handle the next invalidation, the stream's start version vector is advanced to include the just received invalidation.

8. The node's global and, if a fine-grained conit is available, also the local conit state is updated to reflect this invalidation.

9. After the invalidation has been processed, the core notifies the controller.

Processing of bodies Bodies arrive separately from the invalidations. The arrival of a body can occur before or after the corresponding invalidation has been received and processed. When a body arrives, these two scenarios require different handling.

1. If the body arrives before the invalidation, it is stored in a buffer that is checked whenever an invalidation is received.

2. In case the invalidation has already been processed, the body can safely be inserted into the Store. The entry is marked as *valid* and any subsequent retrieval of the body for the received invalidation is successful.

4.2.3 Reads

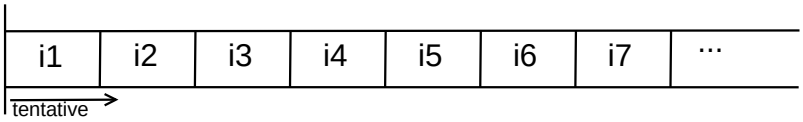
A read can be initiated through the Local API. This *read()* operation allows to specify which item to read and whether the read shall be blocking or not. In the non-blocking case, the read is guaranteed to return the most recent value that is known on this node without contacting any other replica. Blocking reads might contact other replicas and initiate resynchronization. To specify the desired consistency level, a read can be accompanied by a conit identifier for fine-grained conit consistency. If no conit identifier is specified, the default global conit over all items is used. A maximum error bound that is tolerated can be given which allows to further limit the node's error bounds. Currently,

this includes only the consistency bound. As explained in Section 3.4.3, we have not implemented numerical error or staleness.
When a read is executed, the following steps are needed to fulfill the request:

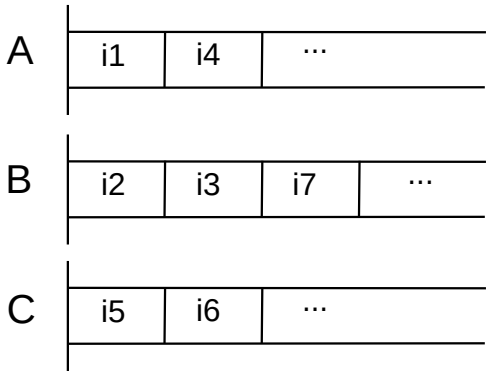
1. In a first step, the system analyzes the current conit state and determines whether the consistency bounds are exceeded.
2. Depending on the parameter choices, different actions are needed to fulfill the read request.
   (a) If a blocking read is requested and the consistency bounds are exceeded, the controller is notified. The controller is expected to arrange everything that is needed according to the implemented replication policy. If the controller returns successfully, we can assume that the requested value is committed and can directly be read from the persistent storage.
   (b) A non-blocking read will always be executed locally without interaction with other replicas. If the Log contains an invalidation for the requested item, the corresponding value will be retrieved from the Store. If no such invalidation is in the Log, the read will return the value from the persistent storage. The read will fail if the body for the invalidation has not yet been received.
   (c) In case of a blocking read where the consistency bounds are not exceeded, the read will attempt to return the last value like in the non-blocking case. If the body for the last invalidation has not yet been received, the controller is notified and initiates a demand body fetch. The controller will return as soon as the body has arrived, in which case the read will return the newly arrived value.

4.2.4 Log

Writes processed by the core and inserted into the Log come in two different formats: either as a precise invalidation, which covers one specific write and describes it in detail, or as an imprecise invalidation, which summarizes several writes. The Log resides in the core (see Section 3.1.2). Writes can come from two sources: from the local API when a write occurs on the local node, or from the remote interface, when a node learns about writes that occurred on other nodes.
The Log is only conceptually a single log. Internally, a separate log for each originator node is used. Figure 4.2 shows the Log holding invalidations $i1$ through $i7$. In Figure 4.2(a) only the conceptual view is shown, while in Figure 4.2(b) the actual structure is shown. Each invalidation is stored in a separate log for the corresponding originator. The example shows logs which contain invalidations that originated on the replicas A, B, and C.
Local writes are straightforward to handle, because they are always precise and the local node receives these writes in a definite order. When a local write occurs, the core will create a precise invalidation and the corresponding body holding the actual data of the write. The newly created invalidation will then be inserted into the Log together with the logical version vector time immediately prior to the invalidation. The Log will store it in the separate log for the respective originator.



(a) Conceptual Log

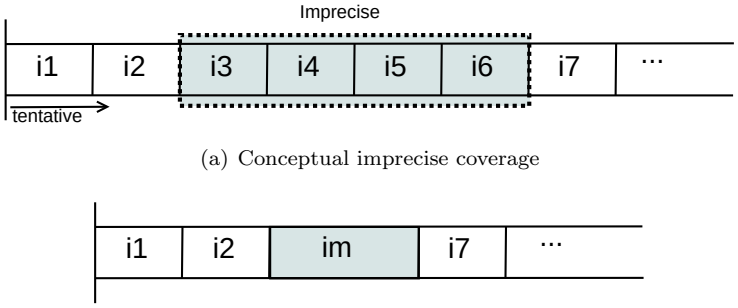


(b) Actual Log

Figure 4.2: Log holding invalidations $i1$ through $i7$

Handling imprecise invalidations is facilitated by this per-originator separation of the Log. Recalling the structure of imprecise invalidations, introduced in Section 3.2.2, we see that imprecise invalidations span an area of invalidations in the per-originator logs. An example of this coverage is shown in Figures 4.3 and 4.4, respectively. The former shows only a conceptual model with one log, while the latter accurately reflects our implementation. The imprecise invalidation summarizes the invalidations $i3$, $i4$, $i5$, and $i6$. While Figures 4.3(a) and 4.4(a) show the area that the imprecise invalidation covers, Figures 4.3(b) and 4.4(b) show the imprecise invalidation in the right places in the respective per-originator logs.
Although the coverage of imprecise invalidations is summarized as one imprecise invalidation, three separate imprecise invalidations are inserted. When an imprecise invalidation is received, the contents are cut into per-originator imprecise invalidations.

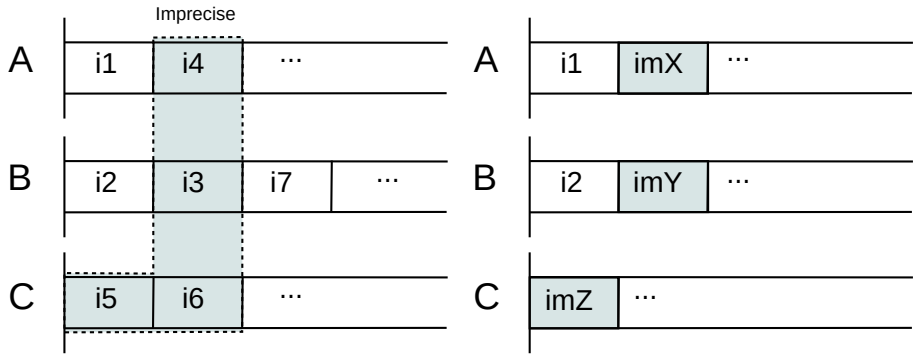
Extraction of invalidations Each subscription of a replica to receive invalidations is accompanied by a version vector which describes the known state of the subscriber. When invalidations are sent, the invalidations after this known state have to be extracted from the Log. The separated log design facilitates this use case. It is possible to extract invalidations starting from a certain logical version vector time across all originator logs. When the invalidations are extracted, their causal ordering is ensured by comparing each invalidation's timestamp and the accompanying version vector that describes the logical time just prior to when this invalidation has been received. It is furthermore possible to create imprecise invalidations for all invalidations for which no interest exists. The start and end times of these imprecise invalidations can directly be inferred from the timestamps of the precise invalidations. When committing in-



(a) Conceptual imprecise coverage

(b) Conceptual log with imprecise invalidation

Figure 4.3: Imprecise invalidations in the conceptual log



(a) Actual log with imprecise coverage

(b) Actual log with imprecise invalidation

Figure 4.4: Imprecise invalidation in the actual log implementation

validations, the Log assumes that all invalidations up to a certain logical version vector time will be processed and written to the persistent storage. This use case is also facilitated by the per-originator log design. All invalidations up to the specified logical time can be extracted and discarded after they have been written to the persistent storage.

4.2.5 Store

The Store holds the values of tentative writes. These values are contained within the bodies that are received. When a new invalidation is received, the Store is informed and sets the entry for this invalidation to *invalid* until the corresponding body is received which will change the entry to be *valid*. Received bodies are stored in a temporary persistent datastore preventing the system from keeping the received bodies in memory. Bodies are removed from the Store when the corresponding invalidation has been removed from the Log, i.e., processed by the commit.

4.3 Separation of Core and Controller

The controller is notified of all relevant events that happen in the core. Events result from interactions with the Local API, from the arrival or sending of invalidations or bodies and from internal sources in the core, for example when the node is started or when the interest set is changed.

4.3.1 Notification API: Core – Controller

Local actions are actions executed through the Local API. Creating, reading, writing, or deleting an item results in a notification to the controller. Each Local API call has a corresponding notification call. The Local API methods with the corresponding notification methods are shown in Table 4.1.

Local API Method	Controller Notification
create	notifyCreate
read	notifyRead
write	notifyWrite
delete	notifyDelete

Table 4.1: Basic operations in Local API and corresponding notification methods

Thus, whenever any of these basic operations is executed, the controller is notified and will react accordingly. The notification interface is shown in the appendix in Listing B.4.
Remote events occur whenever invalidations or bodies are received. Like with the Local API, the occurrence of remote events is reported to the controller via matching notification methods. A mapping of events to the corresponding notification methods in the controller is shown in Table 4.2.

Remote Event	Controller Notification
invalidation received	notifyInvalidationReceived
invalidation stream started	notifyInvalidationStreamStarted
invalidation stream closed	notifyInvalidationStreamClosed
body received	notifyBodyArrived
version vector received	notifyVersionVectorReceived

Table 4.2: Remote events and notifications

The notification methods corresponding to Remote API accesses are shown in Listing B.5 in the Appendix.
Internal sources are events that are not directly related to local actions or remote events. They include all events related to initialization and shutdown, but also include pre-fetching of bodies, subscriptions and added or removed nodes from the system. An (incomplete) overview is given in Table 4.3.

Internal Source	Controller Notification
node started	notifyNodeStarted
invalidations requested	notifyRequestInvalidations
body requested	notifyRequestBody
node added	notifyNodeAdded
node removed	notifyNodeRemoved
⋮	⋮

Table 4.3: Internal events and corresponding notifications

A list of the notifications is given in Section B.2 in Listing B.6. Our current controller does not react to all of these notifications. They are however provided and guaranteed to be called when the corresponding event occurs to ensure that alternative controller implementations can make use of it.

4.3.2 Inform API: Controller – Core

The controller's responsibility is to enforce the replication policy and initiate the sending and receiving of invalidations, bodies and other control messages. As explained in Section 3.1, the core is responsible for the actual mechanism of sending and receiving invalidations and bodies. The controller reacts to these events. Depending on the implemented policy, the controller informs the core about the needed actions telling it what it has to do.
Sending of invalidations and bodies is initiated by using the core's Inform API, which is shown in Listing B.2 of the appendix.
The Inform API also allows the interaction with the Log (shown in Listing B.3), giving access to the Log contents and allowing the controller to initiate a commit. Our controller implementation retrieves the Log content to submit a potential order to the consensus protocol. The commit can either take the current order of the Log, which is inferred from the known information about the invalidations, or an order that is submitted by the controller. This allows us to let all replicas commit the Log contents in the same order.

4.4 Replication Policy

The controller implements our replication policy, which defines when replication mechanism actions like the sending of invalidations and bodies or the commit have to be carried out. We assume full connectivity between the replicas which means that all replicas have a subscription to every other replica. This assumption does not contradict the requirement of topology independence because we assume that the Rhizoma runtime's overlay together with the routing infrastructure will always provide a connectivity to every other replica, even if they are not directly connected. The controller sends invalidations according to the specified send bounds. When this bound is exceeded, the controller initiates the sending of all accumulated invalidations in the local node's Log to every other replica. The resulting invalidations are precise if the subscriptions of the other replicas specify an interest in the described items. In all other cases imprecise invalidations are sent. We employ a push-based approach, where the bodies corresponding to the sent invalidations are also sent at the same time.

4.5 Consistency Enforcement

The system promises causal consistency in all cases and sequential consistency after the agreement protocol, in our case Paxos, has reached consensus on a sequentially consistent ordering. To ensure that these consistency guarantees are provided, our system uses different techniques, which have been partly explained from a conceptual point of view. Their detailed implementation is described in the following sections.

4.5.1 Sequential Consistency with Paxos

By using the already described mechanisms of invalidation streams, the processing of the invalidations and the Log functionality, the system offers a causally consistent ordering of all writes. To provide sequential consistency we use Paxos, a consensus algorithm introduced by Lamport [13]. Paxos has been explained in the background chapter, in Section 2.5. We are using an existing implementation of Paxos and have adapted it so as to work with our system.
The implemented controller assumes the same roles that are introduced by Paxos, which are *proposers*, *acceptors*, and *learners* and ensures that all replicas correctly join to Paxos' replica set. When the core notifies the controller that the consistency bounds are exceeded, the controller will retrieve the local node's Log content. This content is already causally ordered, but other nodes could have different orders for causally unrelated, i.e., concurrent, writes. This order is then submitted as a proposal for all other replicas to agree. If the replica submitting the proposal is a proposer, Paxos will start a consensus protocol run. If the replica is an acceptor, Paxos forwards the proposal to the proposer, which submits the proposal on behalf of the initiating acceptor. Every acceptor in the replica set will either accept or reject the proposal. Accepting the proposal is the normal operation, while rejecting it can occur for example if a previous proposer has crashed after submitting a proposal which was successfully received by the rejecting node.
As already discussed in Section 3.4.4, we have evaluated several possibilities for the consensus reaching. We have abandoned the idea of having several replica sets, each running its own instance of Paxos after first tests indicated a large overhead and a vastly increased complexity when handling commits. We have also investigated the use of Vertical Paxos [14], which allows the use of Paxos to directly replicate data and allows to tune the size of the read and write quorums. Although it is an interesting idea, we have chosen to follow the approach which has been explained above.

4.5.2 Commitment and Persistence

After a consensus has been reached with Paxos, the controller informs the core about the outcome and instructs it to commit the Log content in the agreed order. This happens in two phases. First, the invalidations that are to be committed are extracted from the Log and inserted into the ordered *commitlog*. The content of this *commitlog* is subsequently processed and the corresponding body values written to the datastore. This procedure is guaranteed to only process the invalidations after they have been instructed to be commited by the controller and – in our current implementation – after all corresponding bodies

have been received. This means that our current implementation typically uses the commitlog only for a relatively short amount of time, because we proactively push the bodies together with the invalidations.
If a body for an invalidation in the commitlog is not available, the processing is paused and the corresponding body fetched from another replica. The correct functionality of the data store is still guaranteed. In particular, new writes – both local and remote – and new commit instructions can still be executed.
When an invalidation is contained in the commitlog and its corresponding body is also available, the invalidation is removed from the commitlog and the data contained in the body written to the persistent data store. Additionally, the version vector denoting the currently persisted state of the datastore is updated accordingly.
By chosing to proactively push the bodies together at the same time as the invalidations, we are potentially sending bodies that are not relevant when the invalidations are committed, because newer invalidations for the same data with the corresponding bodies have been received. This is a trade-off that could easily be reversed by using the demand body fetching, which would however increase the latency and therefore increase the overall time needed to commit and process the invalidations and bodies.

4.6 Failure Handling

Every distributed system must deal with failures of machines and unreliable or unavailable connections. The communication module attempts to reconnect if a connection fails. Because the controller is responsible for initiating the actions of the core, it can register callbacks which are called in error cases. The controller implementation must take all needed actions to recover and reinstanciate a stable system state. We assume that rules which are evaluated by Rhizoma will be defined to also handle fault-tolerance scenarios. In particular, rules of the form *"store every item on at least three replicas"* will allow to decrease the risk of data loss. The provided *copy* operations allow to set up such replication but also to change the assignment of items to individual replicas in response to environment changes.

4.7 Communication

All network communication is handled by the communication module. It is intentionally designed to be as independent as possible from the actual replicated data store to allow for easy replacement with other network communication implementations. To interact with the network communication, an instance of the class CommunicationManager is used. This instance implements the interface ICommunication, which provides the essential methods to send and receive messages and data. Connectors are an abstraction, which allow different parts of the system to implement their own receiver instances which react to incoming messages. They implement the IConnector interface which provides methods for the receiving of all relevant message types. Both the ICommunication and IConnector interface are shown in Section B.3.
Any part of the system must use the provided CommunicationManager instance,

but an individual implementation of the IConnector interface must be supplied. Figure 4.5 shows the class diagram of the communication related parts of our system. The communication manager which is dependent on the used network communication interacts with the individual connectors. In our system, three different connectors are used, each with its own responsibility. The StorageConnector is responsible for receiving messages specific to the core and is present even if the controller is changed. Bodies, timestamps, version vectors but also invalidation streams are received through it. The ControllerConnector allows to send and receive replication protocol specific messages, for example control messages for synchronization. It is used for control messages to handle subscriptions and interest sets. PaxosConnector is a connector specifically implemented for Paxos, because the offered ControllerConnector does not provide enough capabilities to fulfill Paxos' requirements. It shows how easily the communication can be adapted to new requirements and how new protocols can be integrated.

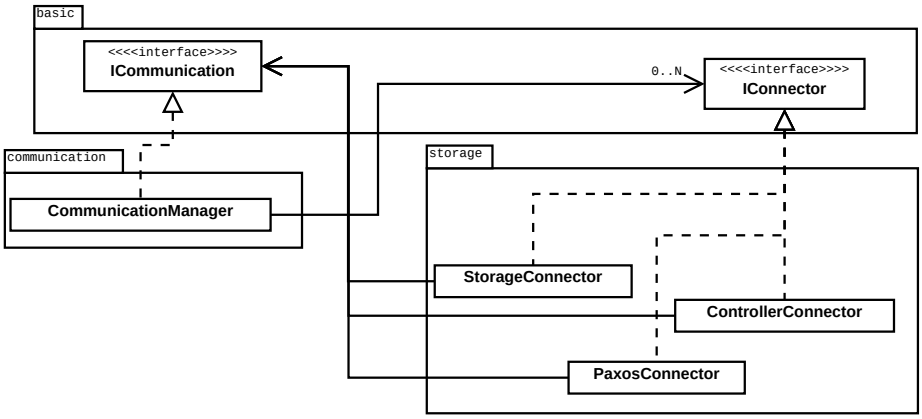


Figure 4.5: Communication class diagram

We are using the Twisted Networking Engine [26] to implement the actual network communication. The communication offers the ability to send control messages as UDP datagrams. Invalidations are sent through a TCP connection between the replicas. Additionally, the bodies are sent through their own TCP stream. Following the separation of invalidations and bodies, this separation in the networking module allows us to use different routes for the two. These three separate communication facilities are shown in Figure 4.6. An alternative implementation of the *ICommunication* interface is planned, which will use Rhizoma's overlay and routing capabilities for the network communication.

4.8 Remote Functionality

Each node exposes a remote API that allows the interaction between the replicas. This API is used for various means, in particular to manage subscriptions, to relay replication protocol specific messages, and to control the operation of the replica.

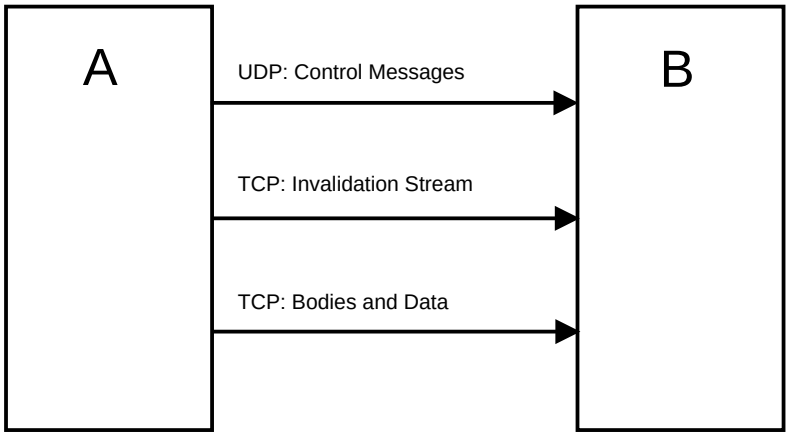


Figure 4.6: Communication between two nodes

4.8.1 Invalidation Subscriptions

Replicas subscribe to each other to receive invalidation streams. Our implementation assumes that each replica subscribes to all other replicas. The PRACTI system [4] shows how this requirement can be relaxed to directly offer topology independence. We rely on Rhizoma's routing capabilities for topology independence.
A subscription carries a version vector that denotes the already known version vector state, an interest set specification, and a set of rules. This information is sufficient for a sending replica to determine which invalidations to send to the subscriber and whether they must be sent in precise or imprecise form. A subscription can be changed by informing the other replicas about the changed interest set, rules and known version vector state. Also, a cancellation option is available.
Subscription handling is implemented in the controller. Table 4.4 shows the involved calls when subscription events are received. Our implementation uses basic dictionaries to encode the parameters and sends these commands by using the control message facilities offered by the communication. The ControllerConnector is used to send and receive these commands. Such subscriptions can be initiated directly in the core by using the *subscribe* call.
A replica can also be requested to subscribe to another replica. This functionality is useful when new replicas are started and they have to subscribe to other replicas in the system.

Subscription Events	Corresponding Call
received invalidation subscription	receivedRequestInvalidations
modification of the subscription	receivedChangeInvalidations
cancellation of the subscription	receivedCancelInvalidations
subscribe to another replica	receivedSubscribeOther

Table 4.4: Remote events for subscription handling

4.8.2 Fetching of Bodies and Data

In addition to the subscription mechanism, it is possible to request bodies or persisted data. After a request for a specific body or persisted item is received, the necessary actions to fulfill this request are initiated by the controller. The ability to fetch bodies is needed if bodies are not pushed together with the invalidations. Selectively fetching data items is needed if the interest set of a replica changes and already persisted missing items must be replicated. The executed calls in response to received fetch actions are shown in table 4.5. Again, the commands themselves are sent and received by using the control message capabilities of the communication. The actual data transfer is then completed by using the body stream.

Fetch Request Type	Corresponding Call
body	receivedRequestBody
item	receivedFetchContent

Table 4.5: Fetch operations

4.9 Persistent Data Storage

The ultimate goal is to persist the stored data items. This is accomplished with the datastore module. The interface IDatastore (shown in Listing B.10 in the appendix) defines an interface which every implementation that provides the possibility to persist data must offer. It is inspired by Amazon's Simple Storage Service (Amazon S3)[2] which offers the concept of buckets and key - value pairs that are stored in these buckets.
We have implemented a basic file-based persistent data store, which stores the items on the local disk. An implementation using Amazon S3 as storage backend is also available.

Chapter 5

Evaluation

5.1 Testbed

We have conducted tests on a variety of computing machines. In the following, we list the machines we used in the tests and summarize their technical specifications.

Desktop A local desktop computer with an Intel Core i5 CPU clocked at 2.67 GHz and 8 GB of RAM running the 64 bit edition of Ubuntu 9.10.

Laptop A local Lenovo Thinkpad X61s laptop with an Intel Core 2 Duo CPU clocked at 1.6 GHz and 2 GB of RAM running Ubuntu 9.10 (32 bit).

Server 1 - 7 Shared server machines with AMD Opteron 250 CPUs clocked at 2.4 GHz and 2 GB of RAM running Debian Linux and located at ETH Zurich.

Planetlab 1 - 4 Four PlanetLab [20] machines located in Switzerland, USA, Spain, and Austria.

5.2 Experiments Overview

We conducted a series of experiments to evaluate the system performance. A short description of each experiment together with its goal is given below.

Write Propagation We measure the time needed for a consistent write with a varying number of replica instances. The experiment is executed using the Desktop and 5 server machines (Server 2 - 6).

Consistency We show two consistency related experiments which show how the send and consistency bounds affect the perceived consistency levels on the replicas.

Replica Changes We evaluate the performance when new replicas join the system to understand which parts of the join process dominate the join time.

Partial Replication We show how partial replication decreases the total size of data that is transmitted and show how the use of imprecise invalidations decreases the number of messages exchanged.

5.3 Write Propagation

Writing an item with a consistency bound of 0 triggers a consensus execution. Figure 5.1 shows the mean time and the standard deviations for such a write with consistency bound set to zero. We have measured the write on the proposer from the moment when the write is issued until Paxos indicates that consensus is reached. The measurement spans configurations ranging from two to six replicas. We have used the Desktop machine running a proposer instance, and up to five server machines (Server 2 - 6) running acceptor instances. The machines were connected with a LAN. For each configuration ranging from 2 to 6 replicas we have executed a series of 10 writes with a delay of one second between them. The writes were executed at the proposer running on the Desktop machine. The detailed results are shown in Table 5.1.
The results indicate linear scaling with increasing numbers of replica instances. We attribute this to the processing overhead at the proposer, which has to send the writes and the Paxos messages to all acceptors. In Section 5.5 we will show an experiment in which Paxos benefits from the diversity of the machines and consensus is reached faster with an increasing number of replicas.

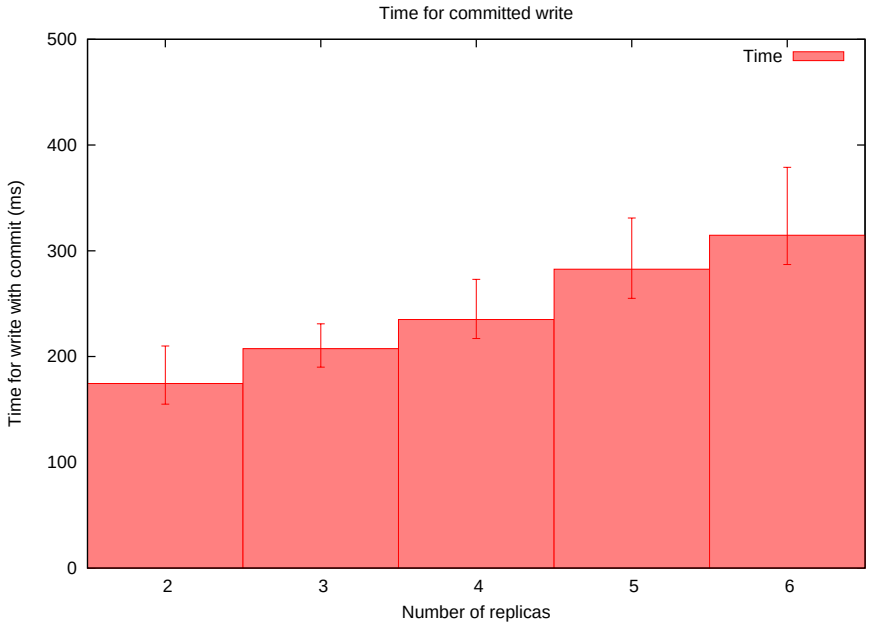


Figure 5.1: Total time for a consistent write

Replicas	Mean (ms)	Median (ms)	σ (ms)	Min (ms)	Max (ms)
2	174.5	169.5	17.05	155	210
3	207.4	202	14.92	190	231
4	235	227	16.40	217	273
5	282.6	280.5	18.58	255	331
6	314.7	312	23.80	287	379

Table 5.1: Mean, median and standard deviation for a consistent write.

5.4 Consistency

Adaption of send and consistency error bounds In a first test, we run a replica set with two replica instances. One is running on the Desktop machine and the other on the Laptop. They are connected with a LAN. The instance on the Desktop machine is the proposer and executes the writes in five phases shown in Table 5.2. The send and consistency bounds are adapted for each phase.
We measure the conit-state, i.e., the current number of inconsistent items with a resolution of 0.4 seconds and execute one write each second. The consistency states of the two replicas are shown in Figure 5.2. Figure 5.2(a) shows the inconsistencies of the Desktop, while Figure 5.2(b) shows the state of the receiving Laptop. Table 5.2 also gives approximate values for the start and end times of each phase on the proposer side. As the two machines are not synchronized, the two time slaces do not exactly match, but they have a gap of 10 to 15 seconds.

Phase 1 In the first phase, 50 writes are executed and immediately committed. The number of inconsistent items ranges from zero to one. It can reach one if our conit-state inspection happens to fall exactly between the arrival of the write and the end of the consensus protocol.

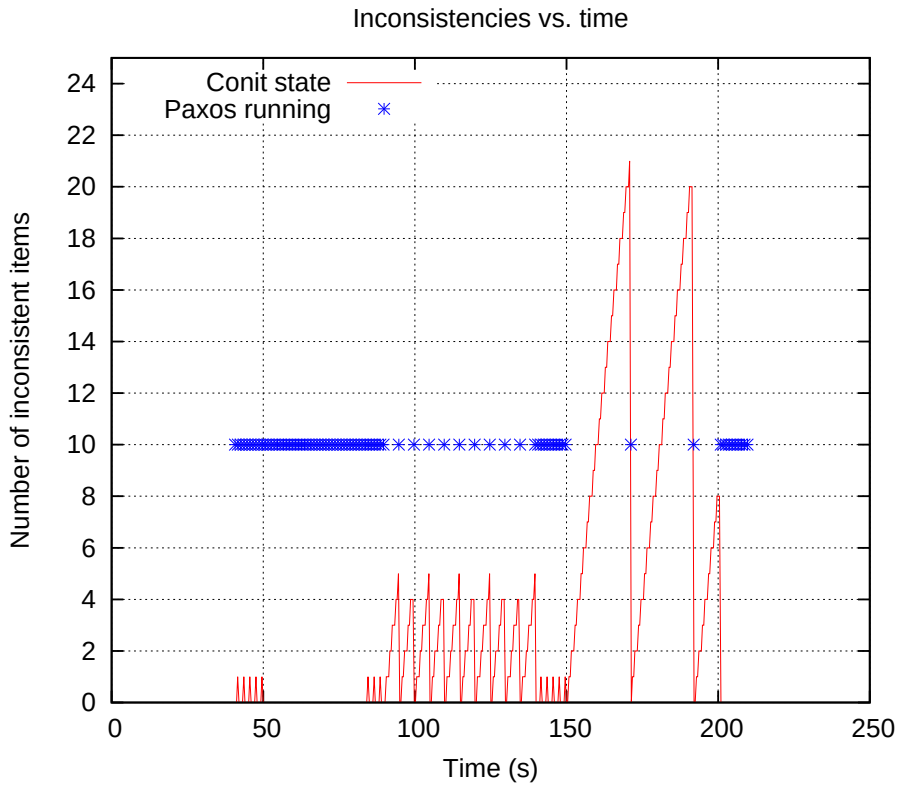
Phase 2 The second phase has a consistency bound of 4, every fifth write triggers a commit.

Phase 3 In the third phase, we again enforce again strong consistency.

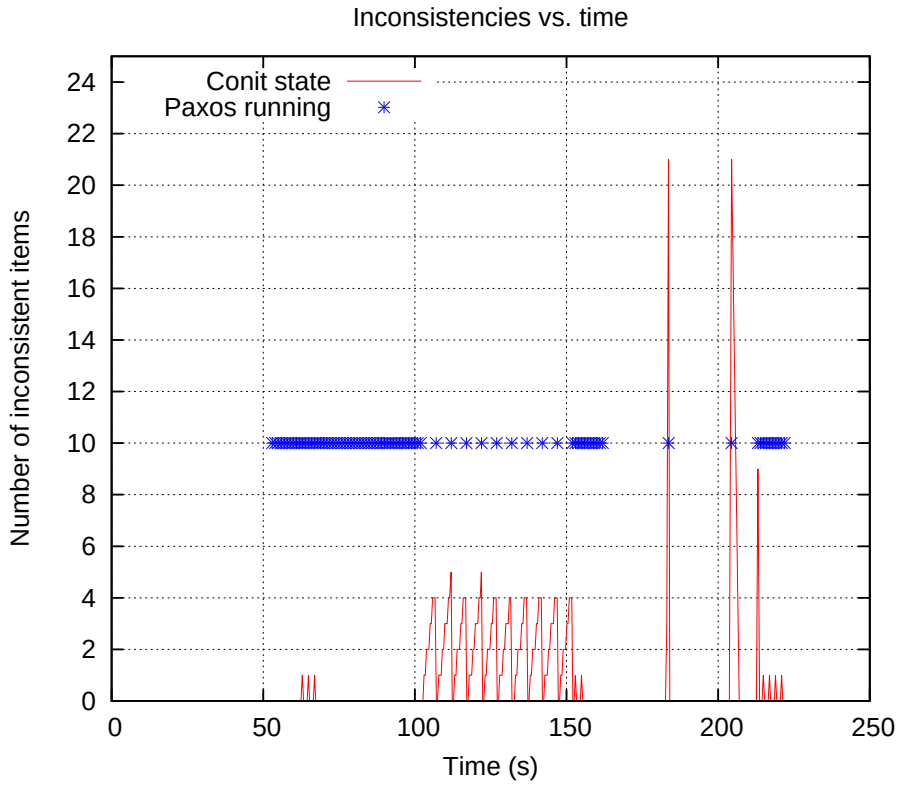
Phase 4 In the fourth phase, we tune the consistency by setting the send and consistency bounds both to 20. The replica on the Desktop executes 50 writes, but the receiving replica on the Laptop only sees them in a bulk. The writes are not immediately propagated. This (artificial) scenario could occur when the device is disconnected for a small period of time. By increasing the bounds, the system continues to be available at the cost of reduced consistency.

Phase 5 The last phase is again used to stabilize the system.

Writes on several replicas In Figure 5.3, we show a trace of the consistency state of a system consisting of three replicas all running locally on the Desktop machine. All three replicas write the item location every 60 seconds. Additionally, two replicas execute workloads with 20 writes each. The writes occur every two seconds. The send bound is zero, and the consistency bound



(a) Desktop: executing writes with varying send and consistency bounds



(b) Laptop: receiving writes with varying send and consistency bounds

Figure 5.2: Writes with differing consistency and send bounds

Phase	Number of Writes	S	C	Write Start (s)	Write End (s)
1	50	0	0	40	90
2	50	0	4	90	100
3	10	0	0	140	150
4	50	20	20	150	200
5	10	0	0	200	210

Table 5.2: Send (S) and consistency (C) bounds for the five phases and the approximate start and end times on the proposer

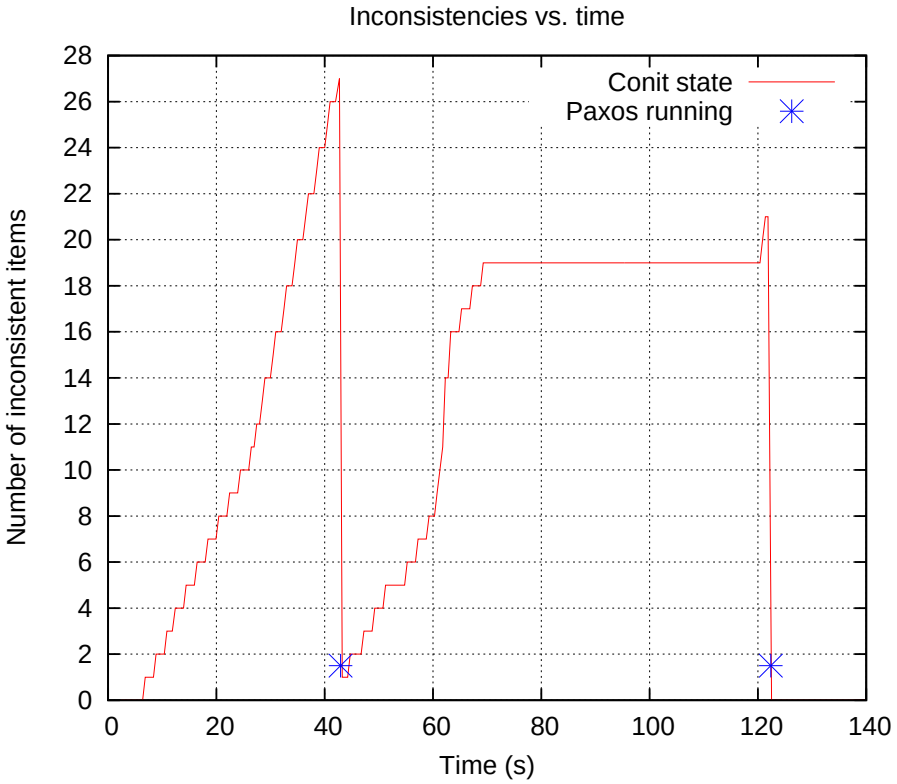


Figure 5.3: Writes on different replicas: consistency bound = 25, send bound = 0

25. In Figure 5.3, only the state of one replica is shown. This replica executes 20 writes and one additional write every 60 seconds. It can be seen how the 40 writes from the two replicas lead to an increase in the number of inconsistent writes, until the consistency bound is exceeded. Also, the depicted replica shows a total of 27 inconsistent writes, which indicates that the last two writes have been received and have not been executed locally (because only local reads or local writes enforce the consistency bounds). After a period with no activity, new writes are received and issued and then a single consistent write is executed locally. The number of inconsistent writes drops to zero.

5.5 Replica Changes

Paxos Replica Join on Planetlab 1 - 4: We assess the overhead incurred by a change in the replica set. This involves invoking a change-view in the Paxos consensus protocol. The test is run on four Planetlab machines (1, 2, 3, and 4). The ping latencies for the machines from ETH Zurich were 1ms, 150ms, 28ms, and 25ms, respectively. Planetlab 1 runs the proposer, the others are all acceptors. The results of starting a new replica which joins the Paxos replica set are shown in Figure 5.4 with the detailed numbers in Tables 5.3 and 5.4.
We measure the elapsed time between the moment a proposer initiates the view change until the proposer has informed all acceptors about the new system configuration. On a new acceptor, we measure the time from the arrival of the first Paxos message, until the last message that confirms successful joining. The experiment has been repeated ten times.
The results show a high variability which is expected on Planetlab machines because they are shared instances. The results for the proposer show a slight decrease. We attribute this to the fact that with more replicas in the system the time Paxos needs to reach consensus benefits from more powerful replicas. In this experiment the third and fourth Planetlab machines have a lower ping latency than the second one, so that their presence helps reaching consensus faster. The results for acceptors are constant.

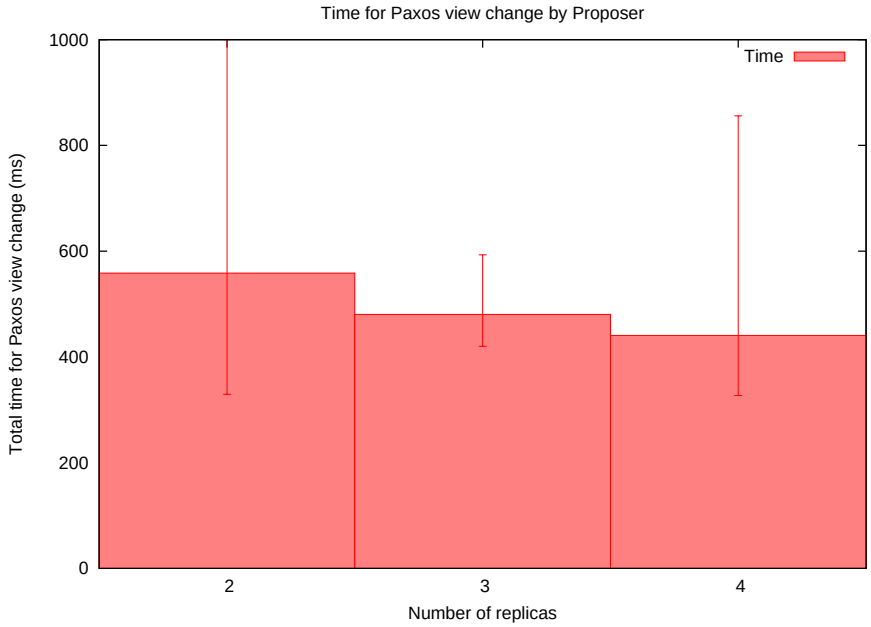
Replicas	Mean (ms)	Median (ms)	σ (ms)	Min (ms)	Max (ms)
2	558.50	423.5	398.34	329	1737
3	480.40	470.0	52.16	420	593
4	440.56	364.0	159.73	327	856

Table 5.3: Paxos Replica Join on Planetlab 1 - 4: mean, median, σ, minimum and maximum times for the proposer to add a replica

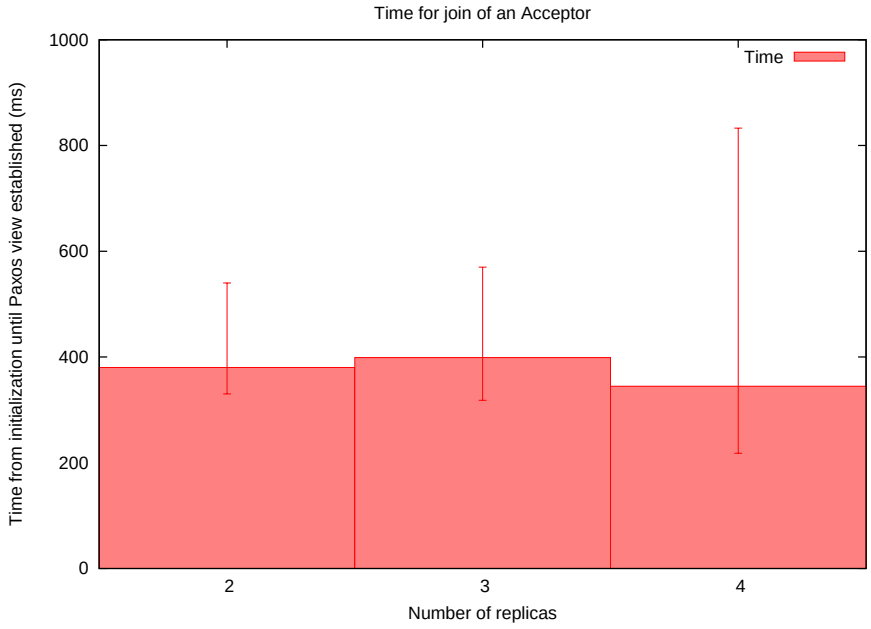
Replicas	Mean (ms)	Median (ms)	σ (ms)	Min (ms)	Max (ms)
2	380.20	363.0	61.22	330	540
3	398.78	381.0	85.68	318	570
4	344.70	239.0	223.16	218	833

Table 5.4: Paxos Replica Join on Plaenetlab 1 - 4: mean, median, σ, minimum and maximum times for an acceptor to be added to the replica set

Copy for new Replicas We expect that in typical deployments new replicas will join a running replication system which already persistently stores data. In such cases, the replica can either request a copy of the full data content of an existing replica or only request a subset of the data, e.g., when the new replica intends to serve only as a partial replica. We analyze the time from the start of a new replica until it has obtained all data and is ready to start operating. We use 2 replicas running on the Desktop and the Laptop machine, which are connected via LAN.
Results for the scenario where the Laptop joins the system and the persistent storage is copied from a desktop machine are shown in Table 5.5 and Figure 5.5.



(a) Proposer



(b) Acceptor

Figure 5.4: Paxos Replica Join on Planetlab 1 - 4: Time needed for a replica to join the system. Experiments on machines Planetlab 1 to 4

Both are connected to a LAN. We have measured different data store contents

between 10 KB and 100 MB.

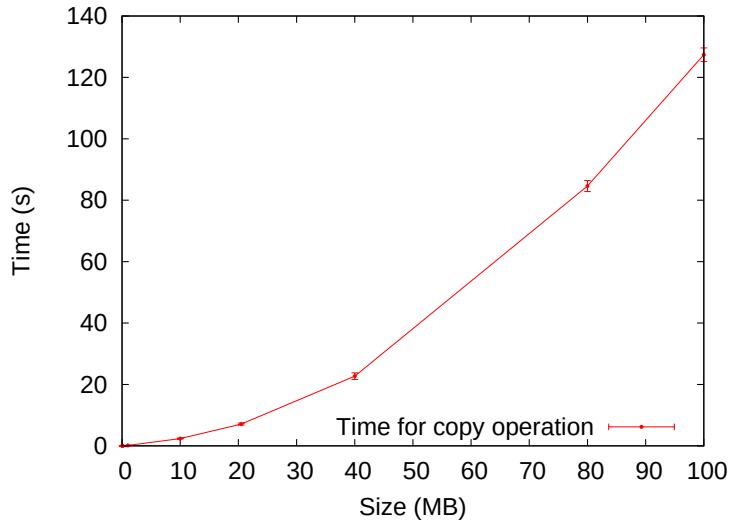


Figure 5.5: Copy of Data to new replica

Size	Time	σ (ms)
10 KB	3 ms	-
80 KB	17 ms	4
100 KB	20 ms	1
1 MB	128 ms	4
10 MB	2241 ms	111
10 MB	2371 ms	156
20 MB	7021 ms	350
40 MB	22.7 s	1082
80 MB	84.6 s	1791
100 MB	127 s	2226

Table 5.5: Time for the copy operation with different sizes of the persistent storage

It is important to note that with data sizes that might be expected in the envisioned scenarios (e.g., [24] presents an example of a user's photography collection reaching 30 GB), the time required to provision a new replica will be dominated by the available bandwidth. The overhead for joining the Paxos replica set and subscribing to other replicas becomes negligible.

5.6 Partial Replication

Partial replication promises benefits to devices which are only interested in a subset of all items. It is useful if a device with limited storage capacity does not have to store all data. To assess the overhead our system has, we are running a

system with five replica instances all running locally on the Desktop machine, and an example file set with ten photos and ten textfiles. Each photo has a size of 1 MB, each text 100 KB. The proposer writes these items with a send bound of 19, meaning that after the twentieth write the log content with all writes is propagated to the other replicas. Furthermore, the consistency bound is also set to 19, so that the propagation of the writes is followed by a consensus phase and the commit. Two replicas participate as acceptors (Acceptor 1 and 2) with full replication. Two other replicas participate as learners (Learner 1 and 2) and are only interested in the photos or the texts, respectively. We measure the number of messages for each replica and report the message numbers separately for the control messages (which include Paxos messages), exchanged version vectors, precise and imprecise invalidations, and bodies. In addition, we show the message sizes and the difference between full and the two different partial replicas.
Figure 5.6 shows the overall number of messages for the two acceptors and the two learners. In Figure 5.7 the sizes of these messages are depicted. While in Figure 5.7(a) it appears as if both learners have received messages of the same size, Figure 5.7(b) shows the size of received body messages. It is apparent that Learner 2 with its partial replica consisting only of textfiles benefits largely.

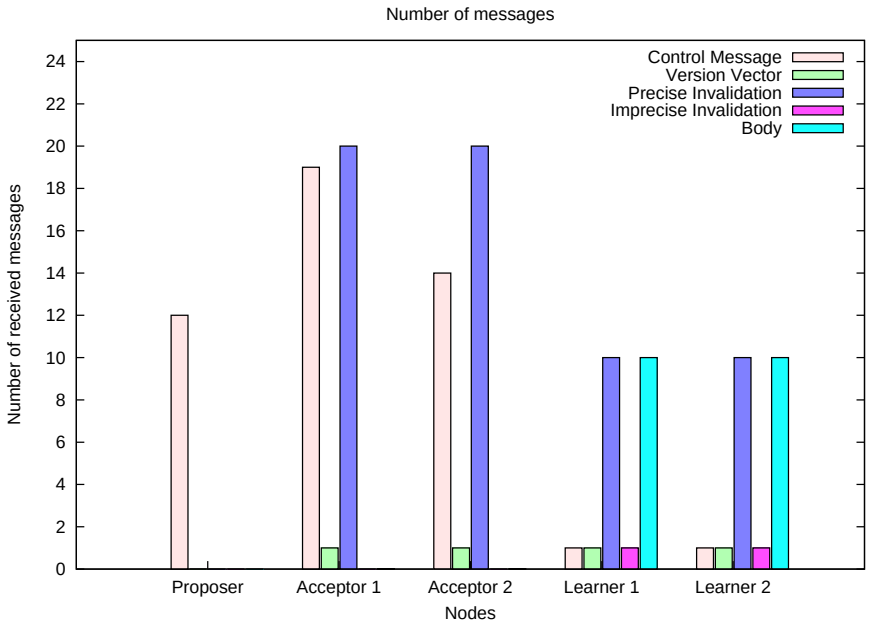
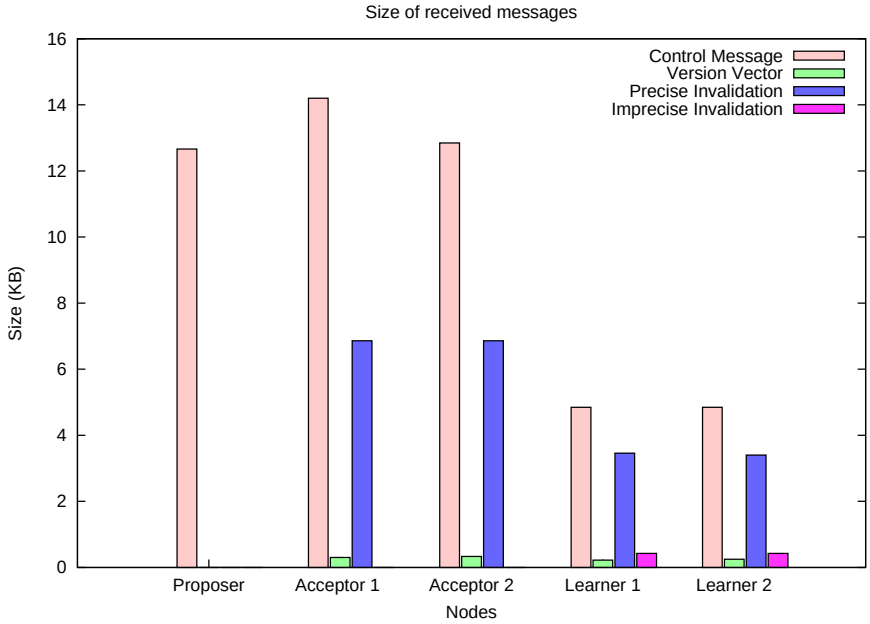
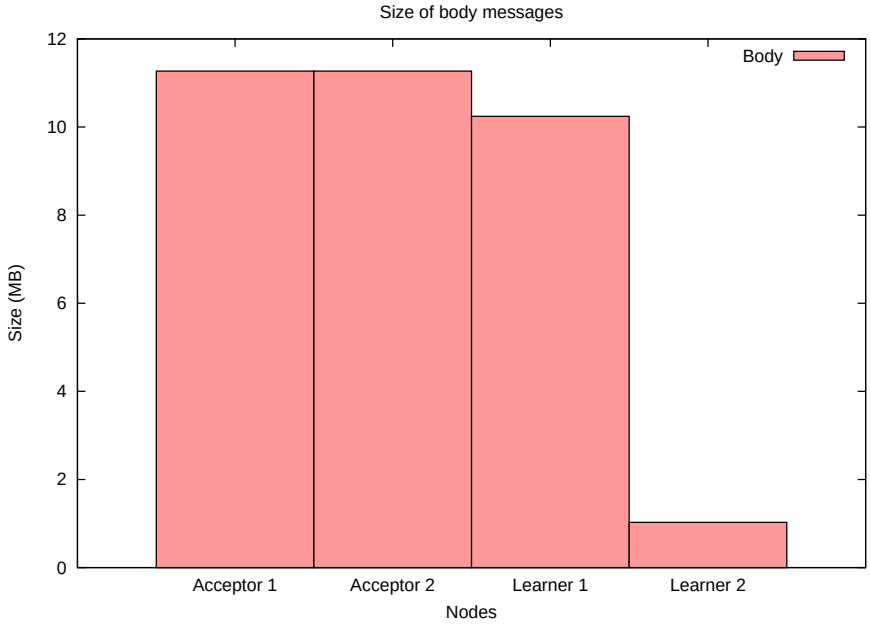


Figure 5.6: Number of messages received



(a) Size of all received messages excluding bodies



(b) Size of the received bodies

Figure 5.7: Sizes of received messages

Chapter 6

Conclusions

6.1 Contributions

We have presented a flexible replication system which fulfills the requirements of Personal Clouds. The system offers partial replication by using the concept of imprecise invalidations which summarize sets of writes. Required consistency guarantees can be flexibly adapted to the needs of applications by using configurable error bounds to limit inconsistencies and staleness. The basic replication mechanism ensures causal ordering of all writes. We have used Paxos to ensure sequential consistency. Topology independence is achieved using the routing facilities of Rhizoma.
To specify partial replication, in addition to the interest set definition, the system allows data to be replicated according to various content-based filters. These filters are specified as constraint logic programs and processed by Rhizoma's optimization engine. Furthermore, the constraint-based optimization of Rhizoma enables the system to adapt to changed execution conditions and ensure fault-tolerance.
We have shown that such a replication system is feasible and can be implemented to fulfill the requirements of Personal Clouds. Our modular architecture leaves room for changes and further research with different approaches. The integration with Rhizoma offers a powerful environment to implement replication policies and execute them in our replication environment.

6.2 Limitations and Open Problems

Flexibility comes at the cost of increased complexity. While the system is capable of satisfying the needs of a broad family of applications, its design can quickly become very complex. To manage this complexity, we make simplified assumptions and also limit the possible replication scenarios. The following sections discuss the current limitations.

6.2.1 Global Log

The system offers partial replication with imprecise invalidations that summarize the content of several precise invalidations. This use of imprecise invalida-

tions prevents the system from efficiently scaling to a large number of replicas that contain partially replicated but non-overlapping data sets. Every replica must be informed about all writes. Depending on the specific configuration of send and consistency error bounds, this could lead to a high overhead for replicas that have no interest in these writes, even if only imprecise invalidations are received. More independence could be offered if replicas without interest in certain data items would not have to receive these imprecise invalidations. A solution to this problem is not obvious and would likely require changes to the mechanism of invalidation sending, the processing of these invalidations in the core and log of our system, and the consistency handling and the guarantees given from the use of them.

6.2.2 Single Paxos Replica Set

A similar problem to having one log for all invalidations is the use of a single replica set for our consensus algorithm. As we have shown in Section 3.4.4, we have analyzed the possibility of using several instances of Paxos which would independently be used for different replica sets. Although we would prefer such a solution where different instances can be used and where other consensus algorithms could be utilized, we have abandoned this idea because of the added complexity which would have required changes to the log handling and probably also changes to the actual handling of invalidations. We believe that a solution allowing several replica sets must also solve the above mentioned problems of having only one global log.

6.2.3 Strong Consistency

Achieving strong consistency in our system requires the cooperation of all replicas and an accordingly laid out policy. One replica not obeying these rules can prevent the system from offering these guarantees. This can happen when a replica specifies higher send error bounds than the rest of the system requires. Reading an item, for which another replica has invalidations in the log that have not been sent to any other replica, will lead to stale data. Such a scenario can only be prevented by rigorous enforcement of the consistency policy on each replica.

6.2.4 Body Invalidation Separation

Separating bodies and invalidations offers greater flexibility in the routing and allows us to exploit network-link specific properties. Our system allows the delayed arrival of bodies and even uses demand fetching of bodies if they are not available. But our approach also raises new problems: a controller implementing the policy must ensure that the bodies are actually correctly sent and received and must ensure that replicas that received precise invalidations for a write also have access to the corresponding body. An extreme scenario would be a system where a write is executed on one node, and all other interested replicas have received the precise invalidation but not the corresponding body. If the replica that initiated the write fails, the information about the write will be correctly replicated by the use of the invalidation. The data of the write will not be available anymore if the original replica cannot recover from the failure. An

additional solution to this problem would be to allow certain invalidations to carry the corresponding body as well. While this would defeat the initial benefit of the separation, it could be used selectively on only a subset of all replicas which ensure fault-tolerance. PRACTI [4] offers such an approach with *bound invalidations*.

6.2.5 Conflict Detection and Resolution

Conflicts are solved by using the last writer's data. The current system offers no mechanism for inspecting and resolving conflicts. We assume that applications using the system will either use a priory consistency specifications that will prevent conflicts (by using bounds of zero), or tolerate our policy where the last writer wins. A possibility is to integrate mechanisms for conflict management like the one used in Bayou [19].

6.3 Future Work

Several prospective areas for future work can be identified. They range from extensions to the overall system including Rhizoma, but also include changes and adaptations to the replicated data storage itself. These extensions are made possible by the architecture of the system, which allows different controllers to be plugged into the system.

Agreement The current use of Paxos with a single replica set could be reconsidered. We have investigated different uses – as explained in Section 3.4.4 – which proved to be too expensive in the required computation and communication and too complex to implement efficiently. We still think that an approach where several overlapping but otherwise mostly independent replica sets would be interesting, because it would allow for easier scaling with more replicas holding partially replicated but mostly non-overlapping data. Such an approach could also better profit from the usage of Rhizoma, which could be used for the management of the partial replicas and the placement of data.

Replication Instead of replicating the actual data in the bodies, the system could be adapted to replicate update operations. The guarantees given by the system with respect to the ordering of the operations in the log are sufficient to support it.

Consistency measures In addition to the order error, numerical error and staleness could also be taken into consideration when specifying the allowed inconsistency and determining the current inconsistency state. In particular the numerical error could be used in various ways. If update operations would be replicated, a numerical deviation from the currently replicated value could be easily computed. For replication of the actual data, numerical error could be defined on universal attributes of the replicated data, for example its size. Additionally, using data- or metadata-specific measures could be an interesting approach.

Routing The separation of bodies from invalidations offers the possibility to implement different routing strategies. The routes taken by invalidations and their corresponding bodies could be determined according to the cost of the chosen link, and optimized accordingly. An example would be a replica set that includes a mobile phone. The invalidations could be sent through the mobile phone's data connection while the bodies could be sent when the mobile phone is connected with a lower-cost wireless LAN. Another possibility is to examine routing potential within constrained environments like data centers. An efficient strategy to send invalidations and bodies via different links could exploit the characteristics of a fully switched network and offer better performance.

Replication protocols To exchange our replication policy and the corresponding protocols, only a new controller must be implemented. This allows to implement vastly different policies, for example by using a primary-approach or other ways to ensure consistency. Different policies could be implemented and then compared to our chosen approach.

Consistency definition Our system offers ways to define the desired consistency by means of error bounds and staleness guarantees. This approach does not easily map to application's needs. Therefore, an interesting area of work is to analyze how an application can describe its consistency requirements and how the mapping to the available implementation with the error bounds and staleness can be accomplished.

Personal cloud applications Since Rhizoma allows the writing of distributed applications, and the replicated data storage is only one part that is also available to application developers, several interesting applications could be built. Examples include personal distributed media systems, that allow to store photos, music and other similar items.

Security No security measures are taken in the current system. Incorporating security and adding encryption of the data could be interesting. Also, no authentication, authorization or encryption mechanisms are used in the communication. The security might also be dynamically adapted to the present environment, for example by enabling it only when the system operates or communicates on public or uncontrolled machines, but disabling it in personal secure environments.

Appendix A

Test Environment

An interactive test console is available for easier testing, which allows to use the replicated storage system. Items can be created, written, read, and deleted. The current system state can be inspected, including the Log and Store contents and the persistent storage.

A.1 Startup

The console is implemented as a Rhizoma application. It can be started from the generated application source directory:

Listing A.1: Starting the Test Console

```
python2.5 rhizoma.py --- -xargs init.xargs  -Drhizoma.
    resourceid=mylaptop
```

This command will start the POSGi runtime, load all necessary modules, wire all modules together and start the replication instance. After startup, a command line interface is available to interact with the replication system.
The command `startPaxos NODE1,NODE2,..` is used to start the first Paxos replica set. Each individual machine participating in the replica set must be named in a comma-delimited list of the form `NODE1,NODE2` where `NODEx` is replaced with the IP address or host and the port of the replica instance. A minimum of 2 machines must be given.

Listing A.2: Starting a Paxos replica set

```
>startpaxos 192.168.1.101:64320,192.168.1.102:12345
```

Additionally to initializing Paxos, `startPaxos` takes care of the needed subscriptions for the replicas.
New instances can be added to or removed from the Paxos replica set by using `addpaxos NODE1,NODE2` and `rmpaxos NODE1,NODE2`, respectively.

Listing A.3: Adding and removing instances

```
>addpaxos 192.168.1.103:23456,192.168.1.104:23458
>rmpaxos 192.168.1.102:12345
```

A.2 Basic Operations

Before writing an item, it must be created. The `create key` command sets up the state to allow writes to the item with the given key.

Listing A.4: Creating an item with key *foo/bar*

```
>create foo/bar
```

To write an item, the `write key value` command can be used. The value can be any string.

Listing A.5: Writing *Hello World* to item with key *foo/bar*

```
>write foo/bar Hello World
```

To read the written value, the command `read key [blocking]` is available. A key must be given. The `blocking` parameter is optional and defaults to `T` for True. A non-blocking read can be requested by specifying it as `F` for False. The read value is directly written to the console.

Listing A.6: Blocking and non-blocking read of item with key *foo/bar*

```
>read foo/bar
Hello World
>read foo/bar F
Hello World
```

Items can be deleted with the `delete key` command.

Listing A.7: Deleting the item with key *foo/bar*

```
>delete foo/bar
```

A.3 Send and Consistency Bounds

The interactive environment offers the ability to review and change the current send and consistency bounds. This can be accomplished with the `ss` command, which is used for send bounds, and the `sc` command for consistency bounds. Invoking them without any arguments will return the current send or consistency bounds, respectively. To change the bounds, an additional numerical parameter can be given. If the given value is -1, the bound is disabled.

Listing A.8: Reviewing and setting the send and consistency bounds

```
>ss
Send bounds: 0
>ss 3
>ss
Send bounds: 3

>sc
Consistency bound: 11
```

```
>sc 7
>sc
Consistency bound: 7
>sc -1

>sc
Consistency bound: -1
```

A.4 State Inspection

The test console also offers several state inspection commands. To retrieve the current known version vector state of the replica, `currentvv` can be used. An equivalent command `persistentvv` is also available for the version vector state of the persistent storage. The following output is taken after a random workload is executed on three replicas. At the moment when these command have been executed, the Log contained 4 invalidations which had not yet been committed. This is also visible in the difference of the version vectors.

Listing A.9: Inspecting the currently known version vector time

```
>persistentvv
PersistentVV:
VersionVector: : originator = ('127.0.0.1', 64322)
{('127.0.0.1', 64321): 41, ('127.0.0.1', 64322): 85,
    ('127.0.0.1', 64320): 20}

>currentvv
CurrentVV:
VersionVector: : originator = ('127.0.0.1', 64322)
{('127.0.0.1', 64321): 41, ('127.0.0.1', 64322): 89,
    ('127.0.0.1', 64320): 20}
```

To inspect the Log, `log` can be used, which lists all invalidations. Additionally, it prints the current state of the Log as triplet *(x, y, z)* where x is the number of precise invalidations in the log, y is the number of imprecise invalidations, and z is the number of unprocessed invalidations in the committedlog.

Listing A.10: Inspecting the Log contents and state

```
>log

MemLog: MemLog:
writer is ('127.0.0.1', 64321)

writer is ('127.0.0.1', 64322)
prevVV=VersionVector: : originator = ('127.0.0.1', 64322)
{('127.0.0.1', 64321): 41, ('127.0.0.1', 64322): 85,
    ('127.0.0.1', 64320): 20},
inval=Invalidation: objid=a/b,
  Timestamp: logical_time='86', originator='('127.0.0.1',
      64322)'
```

```
prevVV=VersionVector: : originator = ('127.0.0.1', 64322)
{('127.0.0.1', 64321): 41, ('127.0.0.1', 64322): 86,
    ('127.0.0.1', 64320): 20},
inval=Invalidation: objid=a/b,
  Timestamp: logical_time='87', originator='('127.0.0.1',
      64322)'
...

writer is ('127.0.0.1', 64320)

LOG STATE: (4, 0, 0)
```

`store` allows to inspect the Store contents. Each entry shows the corresponding invalidation and the validity. If the entry is valid, the corresponding body has been received.

Listing A.11: Inspecting the Store contents

```
>store

MemStore: MemStore:
Item: objid=Invalidation: objid=a/b,
  Timestamp: Timestamp: logical_time='86',
  originator='('127.0.0.1', 64322)',
  valid=True
Item: objid=Invalidation: objid=a/b,
  Timestamp: Timestamp: logical_time='87',
  originator='('127.0.0.1', 64322)',
  valid=True
...
```

A commit can be forced with the `commit` command. The Log contents will be processed and removed from the log. To list the contents in the persistent storage, the command `ls` can be used. It lists all buckets with the corresponding entries. A list of all entries can be obtained using the `list` command.

Listing A.12: Inspecting the Store contents

```
>ls

Bucket: a
    entry: b
```

A.5 Subscription Handling

`subscribers` allows to examine all current subscribers together with their interest set and rules. Subscriptions between replicas can manually be set up.

Listing A.13: Examining the subscribers

```
>subscribers

subscriber_list = {
  ('127.0.0.1', 64320): (set(['*']), None),
  ('127.0.0.1', 64321): (set(['*']), None)}
```

To subscribe to another node, the command `subi HOST POST INTEREST_SET` can be used. The host and port work as expected, the interest set can be specified as a comma-separated list of strings.

Listing A.14: Subscribing to another replica with the specified interest set

```
>subi 127.0.0.1 64321 foo/*,a/*
>subi 127.0.0.1 64322 *
```

A subscriber can also be added (the opposite operation of subscribing) with `addsub HOST PORT [INTEREST_SET]`. The individual parameters work as in the `subi` case.

Listing A.15: Adding a subscriber to the current replica

```
>addsub 127.0.0.1 64321 foo/*,a/*
>addsub 127.0.0.1 64322 *
```

Subscriptions can be changed or deleted with the command `chgi HOST PORT INTEREST_SET` and `cancel HOST PORT`, respectively.

Listing A.16: Changing and cancelling a subscription

```
>chgi 127.0.0.1 64321 new/*
>cancel 127.0.0.1 64322
```

A.6 Copy and Fetch

State copying and fetching of individual items or interest sets is also possible from the command line.
Copying the whole replica to another instance is possible with `copy NODE:PORT`. This copy operation will start a copy of the current replica and also copy all subscriptions. The new replica will be able to participate as full member of the system.
It is also possible to only partially copy the current node's state by `copyis INTEREST_SET` or to copy a single item with `copyobj OBJID`

Listing A.17: Copying to another instance

```
>copy 127.0.0.1:64321
>copyis 127.0.0.1:64322 foo/*
>copyobj 127.0.0.1:64323 foo/test
```

The equivalent operations are also available for fetching, except that they only fetch the content without setting up any subscriptions.

Listing A.18: Fetchin content from another instance

```
>fetchis 127.0.0.1:64322 foo/*
>fetchobj 127.0.0.1:64323 foo/test
```

Appendix B

Interfaces

In this chapter we list the interfaces of the various modules in our system. We take excerpts only of some interfaces for clarity.

B.1 Core

Listing B.1: Local API

```
def create(self, objid, replica_set = None, conit_bounds = None,
    metadata = None):
    '''
    Create the given item and set up all relevant state that is
        needed.
    '''

def read(self, objid, blockingread = True, conit_id = None,
    numerical_error = -1, order_error = -1, staleness = -1):
    '''
    Read the value for the given objid.
    '''

def write(self, objid, value, conit_id = None, write_weight = None,
     metadata = None):
    '''
    Write the given value with objid.
    '''

def delete(self, objid, conit_id = None, write_weight = None ,
    metadata = None):
    '''
    Delete the item with the given objid.
    '''
```

Listing B.2: Inform API: sending of bodies and invalidations

```
    def sendInvalidationList(receiver_list):
        '''
        Send the invalidations to all receivers.
        '''

    def sendInvalidations(receivers, fromVV, interest_sets, rules):
        '''
```

```
        Send the invalidations to the receivers.
        '''

    def sendBodies(receivers):
        '''
        Send the bodies to the receivers.
        '''

    def sendBodyForInvalidation(receivers, invalidation):
        '''
        Send the body for the given invalidation to the receivers.
        '''
```

Listing B.3: Inform API: interaction with the Log

```
    def commit(logorder = None):
        '''
        Commit all tentative writes from the Log and Store to the
        persistent datastore.
        '''

    def getMemLogContent():
        '''
        Return the current ordered Log content as list.
        '''
```

B.2 Controller

The controller implements the IController interface. We show some essential parts of the interface in the following excerpts.

Listing B.4: Notification API: Local API notifications

```
def notifyCreate(key, **params):

def notifyRead(key, withinbounds = True, **params):

def notifyWrite(body, metadata = None, within_general_bounds =
    False, within_send_bounds = False):

def notifyDelete(invalidation, metadata = None,
    within_general_bounds = False, within_send_bounds = False):
```

Listing B.5: Notification API: remote writes

```
def notifyInvalidationStreamStarted(sender):

def notifyInvalidationStreamClosed(sender):

def notifyVersionVectorReceived(version_vector, node):

def notifyInvalidationReceived(invalidation, node_id):

def notifyBodyArrived(invalidation):
```

Listing B.6: Notification API: internal source notifications

```
def notifyStartReplication():
    '''
    Notify the controller that the environment and the core are
        ready. Start the actual replication operation.
    '''

def notifyNodeStarted(id):
    '''
    Notify the controller that this node has started and the core
        is ready to work.

    '''

def notifyNodeAdded(node_id):
    '''
    Notify the controller that another node has been added.
    '''

def notifyNodeRemoved(node_id):
    '''
    Notify the controller that another node has been removed.
    '''

def notifyCopyItemRequested(other_node, objid):
    '''
    Notify the controller that the current node's specified item
        shall
    be copied to the other node.
    '''

def notifyCopyInterestRequested(other_node, interest_set):
    '''
    Notify the controller that the current node's state and data
    for the giben interest set shall be copied to the other node.
    '''

def notifyCopyRequested(other_node):
    '''
    Notify the controller that the current node's persistent state
        and data
    shall be copied to other_node.
    '''

def notifyFetchContent(other_node, objid = None, interest_set =
    None):
    '''
    Notify the controller that the content corresponding to the
        interest shall
    be fetched from other_node.
    '''

def notifySendContent(other_node, interest = None):
    '''
    Notify the controller that the content corresponding to the
        interest shall be sent to the other node.
    '''

def notifySubscriberAdded(node_id, fromVV = None, interest_sets =
    None, rules = None):
    '''
    Notify the controller that a new subscriber has been added to
        this node.
```

```
    '''

def notifySubscriberRemoved(node_id):
    '''
    Notify the controller that the subscriber has been removed.
    '''

def notifyRequestInvalidations(knownVV, interest_sets = None,
    sender_node = None, rules = None):
    '''
    Request that the invalidations be send to this (current) node.
    The controller must ensure that the right requests are sent to
        the right nodes.

    This serves as subscription to the particular invalidation
        stream.
    '''

def notifyCancelInvalidations(sender_node = None):
    '''
    Unsubscribe the invalidation subscription.
    '''

def notifyChangeInvalidationSubscription(knownVV, interest_sets =
    None, sender_node = None, rules = None):
    '''
    Change the invalidation subscription of sender_node. Will
        modify what the other node knows
        about our invalidation subscription.
    '''

def notifyInstructSubscribeOther(node_id, other, interest_sets =
    None, rules = None):
    '''
    Instruct the node 'node_id' to subscribe to 'other'
    '''

def notifyRequestBody(invalidation, sender_node = None):
    '''
    Request that a body corresponding to the given invalidation be
        sent to this node.
    '''

def notifyBoundsExceeded(conit_id = None, key = None):
    '''
    Notifies the controller that the consistency bounds have been
        exceeded.
    '''
```

Listing B.7: Notication API: Remote API notifications

```
def receivedRequestInvalidations(sender, knownVV, interest_sets =
    None, rules = None):
    '''
    Serve the request by the sender to receive invalidations after
        knownVV.
    '''

def receivedCancelInvalidations(sender):
    '''
    Cancel the sender's invalidation stream subscription.
    '''
```

```
def receivedChangeInvalidations(sender, knownVV = None,
    interest_sets = None, rules = None):
    '''
    Change the senders invalidation subscription to the new
        parameters.
    '''

def receivedSubscribeOther(sender, other, interest_sets = None,
    rules = None):
    '''
    Received a request to subscribe to the specified other node(s).
         The other replica will
    serve the current node with invalidations.
    '''

def receivedRequestBody(sender, invalidation):
    '''
    Serve the request by the sender to receive the body for the
        given invalidation.
    '''

def receivedFetchContent(sender, objid, interest_set):
    '''
    Received a request to fetch content. We have to send all
        content corresponding to interest to
    the sender.
    '''

def receivedProtParamsRequest(sender, item_id):
    '''
    Process request for protocol parameters. This is invoked to
        find out the replica set for
    a certain item_id as well as its primary.
    '''

def receivedProtParamsResponse(sender, item_id, params):
    '''
    Process the protocol parameters response.
    '''
```

B.3 Communication

Listing B.8: ICommunication interface

```
class ICommunication(Interface):
    """
    Communication manager interface.
    """

    def start():
        '''
        Start listening on the specified port.
        '''

    def stop():
        '''
        Stop listening to the specified port. (Cleanup)
        '''
```

```
    def has_transport():
        ''' Is a transport set? '''

    def registerConnector(connector, id):
        '''
        Register the receiver with 'id'.
        Messages for this receiver will be dispatched to this
            receiver.
        '''

    def removeConnector(id):
        '''
        Remove the connector with the specified id.
        '''

    def getConnector(id):
        '''
        Return the connector with the specified id.
        '''

    def receiveMsg(msg, id):
        '''
        Receives the message and dispatches it to the proper
            handler and component.
        '''

    def sendLargeMessages(self, content, receiver_id, receivers):
        '''
        Send content that would not fit within a UDP datagram.
        '''

    def sendMessage(self, content, receiver_id, receivers):
        '''
        Send the content to the specified receivers. This can be
            used for
        replication protocol specific messages.
        '''

    def sendMessagesOrdered(self, messages, receiver_id, receivers)
        :
        '''
        Send the messages in the order of the list to the receivers
            .
        '''
```

Listing B.9: IConnector interface

```
class IConnector(Interface):
    '''
    Connector interface for the communication.

    Connectors must implement this interface to be registered with
    the ICommunication.
    '''

    def receiveMessage(sender, content):
        '''
        Receive the message with the specified content.
        '''

    def receiveFromStream(sender, instance):
```

```
        '''
        Receive an instance from the stream. This is guaranteed to
            be delivered in order.
        '''

    def registerCommunication(communication):
        '''
        Register the communication with this connector.
        '''

    def hasCommunication():
        '''
        Has a communication instance been registered?
        '''

    def start():
        '''
        Communication operation is started. Ensure that the
            connector
        is set up correctly to receive messages. Sending is
            possible now.
        '''

    def stop():
        '''
        Informs that all communication has been stopped.
        The connector must not send anything, no further
            communication will
        occur until start() has not been called again.
        '''

    def isActive():
        '''
        Is the communication active?
        '''
```

B.4 Datastore

Listing B.10: IDatastore interface

```
class IDatastore(Interface):

    def getName():
        '''
        Return the name of the datastore (e.g., local file system,
            S3, etc.).
        '''

    def getSize():
        '''
        Return the size of the datastore
        '''

    def createBucket(bucketname, **kw):
        '''
        Allocate a bucket with the specified requirements.
        '''

    def getBucket(bucketname):
```

```
        '''
        Retrieve a bucket with the specified name.
        '''

    def getBuckets():
        '''
        Retrieve all buckets.
        '''

    def deleteBucket(bucketname):
        '''
        Delete a previously allocated bucket (can be local or
            remote).
        '''

    def getBucketContents(bucketname):
        '''
        Returns the content of this bucket as list of key/value
            pairs
        '''

    def getBucketItemsMetadata(bucketname):
        '''
        Returns the metadata for all items in this bucket as list
            of key/metadata pairs
        '''

    def getBucketMetadata(bucketname):
        '''
        Returns the metadata for this bucket (not for the items!)
        '''

    def itemExists(bucketname, itemkey):
        '''
        Check if a specific item exists in the specified bucket.
        '''

    def createItem(bucketname, itemkey, value, metadata = None):
        '''
        Create an item in the specified bucket.
        '''

    def deleteItem(bucketname, itemkey):
        '''
        Delete a previously allocated item (can be local or remote)
            .
        '''

    def getItem(bucketname, itemkey):
        '''
        Return the item object.
        '''

    def getItemMetadata(bucketname, itemkey):
        '''
        Return the item metadata.
        '''

    def updateItem(bucketname, itemkey, value, metadata = None):
        '''
        Update an item with the specified key and specified value.
        '''
```

Bibliography

[1] Amazon Elastic Compute Cloud. `http://aws.amazon.com/ec2/`.

[2] Amazon Simple Storage Service. `http://aws.amazon.com/s3/`.

[3] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, 1994.

[4] Nalini Belaramani, Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. Practi replication. In *NSDI'06: Proceedings of the 3rd conference on Networked Systems Design & Implementation*, pages 5–5, Berkeley, CA, USA, 2006. USENIX Association.

[5] Eric A. Brewer. Towards robust distributed systems (abstract). In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, page 7, New York, NY, USA, 2000. ACM.

[6] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[7] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent available partition-tolerant web services. In *In ACM SIGACT News*, page 2002, 2002.

[8] Richard A. Golding. A weak-consistency architecture for distributed information services. Technical report, University of California at Santa Cruz, Santa Cruz, CA, USA, 1992.

[9] Maurice P. Herlihy and Jeannette M. Wing. Axioms for concurrent objects. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 13–26, New York, NY, USA, 1987. ACM.

[10] Philip W. Hutto and Mustaque Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *In Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 302–311, 1990.

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

[12] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programm. *IEEE Transactions on Computers*, 28:690–691, 1979.

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

[14] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *PODC '09: Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 312–313, New York, NY, USA, 2009. ACM.

[15] Wei Lin, Mao Yang, Lintao Zhang, and Lidong Zhou. Pacifica: Replication in log-based distributed storage systems. Technical report, Microsoft Research, 2008.

[16] Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[17] David Maziéres. Paxos made practical. `http://www.scs.stanford.edu/~dm/home/papers/paxos.pdf`, January 2007.

[18] OSGi. `http://www.osgi.org/`.

[19] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 288–301, New York, NY, USA, 1997. ACM.

[20] PlanetLab: An open platform for developing, deploying, and accessing planetary-scale services. `http://www.planet-lab.org/`.

[21] Python Programming Language. `http://www.python.org/`.

[22] Venugopalan Ramasubramanian, Thomas L. Rodeheffer, Douglas B. Terry, Meg Walraed-Sullivan, Ted Wobber, Catherine C. Marshall, and Amin Vahdat. Cimbiosys: a platform for content-based partial replication. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 261–276, Berkeley, CA, USA, 2009. USENIX Association.

[23] Brandon Salmon, Steven W. Schlosser, Lorrie Faith Cranor, and Gregory R. Ganger. Perspective: semantic data management for the home. In *FAST '09: Proccedings of the 7th conference on File and storage technologies*, pages 167–182, Berkeley, CA, USA, 2009. USENIX Association.

[24] Jacob Strauss, Chris Lesniewski-Laas, Justin Mazzola Paluska, Bryan Ford, Robert Morris, and Frans Kaashoek. Device transparency: a new model for mobile storage. *SIGOPS Oper. Syst. Rev.*, 44(1):5–9, 2010.

[25] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice Hall, 2 edition, 10 2006.

[26] Twisted Networking Engine. `http://twistedmatrix.com/`.

[27] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 7–7, Berkeley, CA, USA, 2004. USENIX Association.

[28] Werner Vogels. Eventually consistent. *Queue*, 6(6):14–19, 2008.

[29] Qin Yin, Adrian Schüpbach, Justin Cappos, Andrew Baumann, and Timothy Roscoe. Rhizoma: a runtime for self-deploying, self-managing overlays. In *Middleware '09: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, pages 1–20, New York, NY, USA, 2009. Springer-Verlag New York, Inc.

[30] Haifeng Yu and Amin Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.*, 20(3):239–282, 2002.