JPaxos: State machine replication based on the Paxos protocol

Jan Kończak[2], Nuno Santos[1], Tomasz Żurkowski[2], Paweł T. Wojciechowski[2], and André Schiper[1]

[1]EPFL, Switzerland[*]
[2]Poznań University of Technology, Poland[†]

July 31, 2011

Abstract

State machine replication is a technique for making services fault-tolerant by replicating them over a group of machines. Although the theory of state machine replication has been studied extensively, the engineering challenges of converting a theoretical description into a fully functional system is less understood. This creates difficulties to implementors, because in designing such a system they face many engineering challenges which are crucial to ensure good performance and stability of a replicated system. In this report, we address this problem by describing the design and implementation of *JPaxos*, a fully-functional implementation of state machine replication based on the MultiPaxos protocol. Our description includes the basic modules of a state machine replication implementation, like snapshotting of service state, state-transfer and keeping up-to-date all replicas, but focus mainly on three aspects: recovery mechanisms, batching and pipelining optimizations, and a scalable threading architecture. We present several recovery algorithms that vary in the usage of stable storage and on the system assumptions, including some that use stable storage only once per recovery. Batching and pipelining are well-known optimizations commonly used in state machine replication. With JPaxos we have studied their interaction in detail, and provide guidelines to tune these mechanisms for a variety of systems. Finally, the threading architecture of JPaxos was designed to scale with the number of cores, while at the same time minimizing complexity to reduce the risk of concurrency bugs.

1 Introduction

As the reliance on Internet services in everyday life increases, so does the need of ensuring their reliability and availability in the presence of faults. Software replication is a widely used techniques to ensure fault tolerance. It works by replicating the service and its data among several nodes, so that the system is able to continue operating correctly even if some of the nodes fail.

Among the software replication techniques, state machine replication is the one that has received the most attention in the recent years. This technique can be applied to services that can be implemented as a deterministic state machine, i.e., where the next state of the service is a function solely of the current state and the command executed. The service is then replicated on several nodes, all of them running a state machine replication middleware. This layer is responsible for receiving the requests from the clients, establishing a total order among the requests, and executing the sequence of requests independently at all replicas. As the service is deterministic and every replica executes the same sequence of requests, the state of the different replicas is kept consistent. Therefore, if some replica fails, the others can continue operating the service.

In spite of its conceptual simplicity, state machine replication is remarkably hard to design and implement, requiring both a solid theoretical background and careful engineering. On the theoretical side, the difficulties are ensuring the correctness of the replication algorithms, which tend to be complex

[*]{firstname.secondname}@epfl.ch
[†]jan.konczak@student.put.poznan.pl, tomasz.zurkowski@gmail.com, Pawel.T.Wojciechowski@cs.put.poznan.pl

and subtle, and to design efficient algorithms for a wide range of systems and workloads. Fortunately, there has been a lot of work on the topic during the last 30 years, which can be used as a starting point to build practical systems. In particular, the Paxos family of algorithms [1] has received much attention in the literature, which includes several variations of the basic algorithm exploring different parts of the design space (and therefore, different tradeoffs) [2, 3, 4, 5, 6], and detailed performance analsysis [7, 8]. Paxos is particularly interesting due to being optimal in the number of message delays [9] (3 message delays, starting from client sending request until decision) and in having optimal resilience ($2f + 1$ with crash faults and $3f + 1$ with byzantine faults, given the corresponding variations of Paxos, with f being the maximum number of faulty nodes). Thus, in the following we will focus on the Paxos algorithm. We consider only non-byzantine faults, that is, replicas can only fail by crashing, never performing actions that are not specified by the algorithm.

Contrary to the theoretical aspects, the engineering challenges of implementing Paxos have received a lot less attention. This is unfortunate, as the theoretical work on Paxos leaves unspecified many important practical details that are essential to achieve high-performance and stable operation in demanding real-world environments. This includes, among others, details like managing a finite replication log (in theoretical work Paxos is often described using an infinite log), recovering the state of replicas that crashed, ensuring that replicas stay "close together" in the sequence of executed commands in spite of message loss and of varying execution speeds between replicas, and maximizing utilization of system resources like network and CPU cores. This lack of relevant literature has become more of a problem recently with the large growth in the use of Internet services, which in order to decrease costs and improve time-to-market use software fault-tolerance on top of inexpensive commodity hardware. It is therefore necessary to complement the theoretical work on Paxos with a detailed analysis of the engineering challenges. There is some recent work in this direction [10, 11], but there is still much left to be done.

In the work presented in this report, we discuss, propose and evaluate solutions to the major engineering challenges of creating a Paxos implementation, with the goal of helping bridging the gap between theory and practice. As part of our work, we have implemented JPaxos, a full-fledged, high-performance Java implementation of state machine replication based on Paxos. In order to allow others to extend our work or use it either for research or to build fault-tolerant services, we have released JPaxos as open-source.

In our work, we focused on the following main aspects: 1) batching and parallel instances as a mechanism to maximize performance in a wide range of operating conditions, 2) recovery mechanisms for the crash-recovery model, with or without stable storage, 3) and scalability of the middleware with multi-core CPUs.

Batching multiple client requests in a single consensus instance is a well-known technique to improve the throughput of state machine replication, by amortizing the fixed-costs of a consensus instance over several client requests. The parallel instances optimization takes advantage of the ability of the Paxos coordinator to execute multiple instances of the ordering protocol concurrently, which in some conditions, especially high-latency networks, can greatly increase performance. Although both techniques are well-known and are part of most Paxos implementations, they had not been studied in detail in the context of Paxos. We have done so, by implementing both optimizations in JPaxos and evaluating them in a variety of situations, both analytically and experimentally [12].

The second major contribution of our work is the exploration of recovery techniques for the crash-recovery model. Any Paxos system intended to operate for long periods of time must reintegrate replicas that crashed, otherwise the system will eventually stop as replicas crash without being replaced. As mentioned in the original Paxos paper, by using stable storage Paxos can be easily extended to tolerate catastrophic failures, i.e., *simultaneous* failure of an arbitrary number of replicas. However, this requires one stable storage access at every replica for each consensus instance, which usually decreases the performance by an order of magnitude as compared to the crash-stop version of Paxos. For services that do not need to tolerate catastrophic failures, it is possible to retain a performance very close to the one in the crash-stop model while still tolerating an arbitrary number of faults over the lifetime of the system. This works by having replicas that recovered from a crash rebuild their state using the state of the other replicas that are are fully operational, at which point they can re-join the system as a full member, thereby restoring the resilience level. This approach has a lower resilience, because it stops if the number of replicas with full state drops below a majority at any point, but on the other hand it uses stable storage less frequently, in particular, it does not require a stable storage access per consensus instance, making it an attractive alternative. Below we study two algorithms for this restricted crash-recovery model, one that was previously proposed in the literature and another that is novel. We have implemented both algorithms in JPaxos, together with the traditional algorithm for the crash-recovery model with catastrophic failures (in addition to the version of Paxos for the crash-stop model), and evaluated them experimentally.

On the engineering side, we put special attention in parallelizing JPaxos so that its performance scales with the number of cores. Parallelizing programs is a challenging problem in general, for which there is no general solution. Indeed, although general principles do exist, parallelism must still be addressed mostly on a case-by-case basis, as each class of applications has its own potential for parallelism and, therefore, requires a tailored threading design to maximize its scalability with the number of cores. We have done such analysis for JPaxos, and we believe that our results can be generalized to other implementations of state machine replication. In an application like JPaxos, where a large amount of shared state (the replicated log) is accessed by a variety of logical modules, it is tempting to implement the system as a single-thread event-driven architecture, as this sidesteps all the concurrency issues. However, this architecture clearly does not scale. On the other side of the design spectrum, we find the fully threaded architectures. But these require careful locking, which is remarkably hard to get right, as it must ensure safety while at the same time avoiding excessive lock contention that can easily limit scalability. The architecture of JPaxos is a compromise between the two extremes, with some modules implemented as a sequence of event-driven stages (with one or more threads) communicating by queues (inspired by SEDA[13]) and others using traditional threads with blocking operations. This hybrid architecture is motivated by the observation that while some tasks have much to gain in terms of simplified concurrency or parallelism by being implemented as event-driven stages, others have little to gain while at the same time having much simpler single threaded implementations (for instance, retransmission and failure detection). The resulting architecture balances scalability with implementation complexity.

This report is organized as follows. Section 2 briefly presents our model and defines the relevant terms, Section 3 presents the Paxos algorithm. In Section 4 we present the architecture of JPaxos; describing the internal structure, how the clients and replicas interact, and how a request is handled. Section 5 describes the implementation of the main modules of JPaxos, including the details of our MultiPaxos implementation, the mechanisms for stale replicas to catch-up with the up-to-date replicas, and the snapshotting system. Section 6 studies the optimizations of batching and pipelining, with an analytical analysis that provides the optimal configuration parameters for a given system and workload. In Section 7 we present the recovery algorithms of JPaxos. Section 8 explains the threading architecture of JPaxos. Section 9 shows a summary of experimental performance results of JPaxos, with a focus on the tradeoffs between batching and pipelining. Finally, Section 10 discusses related work and Section 11 concludes the report.

2 Definitions and Model

We consider the partially synchronous model with benign faults, consisting of a set of n processes that communicate by message passing. The system alternates between periods of synchrony and asynchrony. During periods of asynchrony, messages may be lost or delayed, and processes execute at arbitrary speeds. During synchronous periods, there is no message loss, and the relative execution speed of processes and the message transmission time are bounded.

JPaxos implements variants of Paxos for both crash-stop and crash-recovery faults, so in the following we describe both.

2.1 Crash-stop model

In the crash-stop model processes can fail by crashing and crashes are permanent, *i.e.*, a faulty process eventually stops executing the algorithm permanently. Thus, a *correct process* is one that never crashes and executes the algorithm forever. The Consensus problem is defined by two primitives: $propose(v)$ and $decide(v)$. A process initiates consensus by proposing its initial value by calling $propose(v)$ and eventually executes $decide(v)$ to decide on a value. In the crash-stop model, these primitives must satisfy the following properties:

- **Termination** - Every correct process eventually decides some value.

- **Uniform Integrity** - Every process decides at most once.
- **Uniform Agreement** - No two processes (correct or not) decide a different value,
- **Uniform Validity** - If a process decides v, then v was proposed by some process.

2.2 Crash-recovery model

In the crash-recovery model a process can resume execution of the algorithm after crashing. In this model, we distinguish between volatile and stable storage. Any state stored in volatile storage is lost upon a crash, while the state in stable storage is preserved.

In the crash-recovery model processes may be faulty in more ways than just crashing permanently, *i.e.*, they may alternate between being up (executing the algorithm) and down (crashed). Using the terminology of [14], we say that a process is *always-up* if it never crashes, *eventually-up* if it crashes a finite number of times, *eventually-down* if it eventually crashes permanently, and *unstable* if it crashes (and recovers) an infinite number of times.

For practical purposes, a process needs only to be up for long enough to participate in one or more instances of the protocol in order to be useful. Therefore, in the rest of this report, we implicitly assume that during a period of synchrony (a good period), there are enough processes up to advance the protocol ($f + 1$ for MultiPaxos), regardless of whether they crash or not after the good period.

2.3 State-machine replication

In state-machine replication, the service is modeled as a deterministic state machine and executed simultaneously at several processes (replicas). The replication protocol is responsible for receiving commands from the clients and executing them at every replica in the same order. As the service is deterministic, *i.e.*, the next state of the service depends only on the current state and the command executed, the replicas' state remain consistent.

State machine replication is closely related to the atomic broadcast problem, as both share the same core problem of ordering a sequence of values. In fact, MultiPaxos can be seen as a *sequencer-based* atomic broadcast protocol [15], where the sequencer orders requests received from the clients. We present next the formal definition of atomic broadcast, because it contains the core elements of state machine replication while being simpler to express.

Atomic broadcast in the crash-stop model is defined in terms of two primitives: *abcast*(m) and *adeliver*(m), that satisfy the following properties:

- **Validity** - If a correct process *abcasts* a message m, then it eventually *adelivers* m.
- **Uniform agreement** - If a process *adeliver*s a message m, then all correct processes eventually *adeliver* m.
- **Uniform integrity** - For any message m, every process *adelivers* m at most once, and only if m was previously *abcasted.*
- **Total order** - If some process, correct or faulty, adelivers m before m', then every process adelivers m' only after it has adelivered m.

The atomic broadcast problem is equivalent to consensus [16], *i.e.*, if in a given system model one can be solved, then the other can also be solved.

3 The MultiPaxos state machine replication algorithm

MultiPaxos is an algorithm for a group of processes to agree on a sequence of values in the presence of faults. It works in the partially synchronous model with benign faults, with variants for the crash-stop and crash-recovery model. It requires $2f + 1$ processes to tolerate f faults. We start our description of MultiPaxos by describing the single instance Paxos consensus algorithm that lies at the heart of MultiPaxos. Both algorithms were first described in [1].

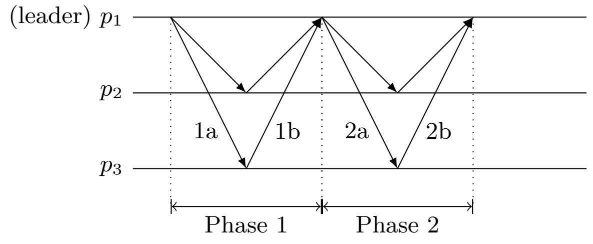


Figure 1: A Paxos ballot

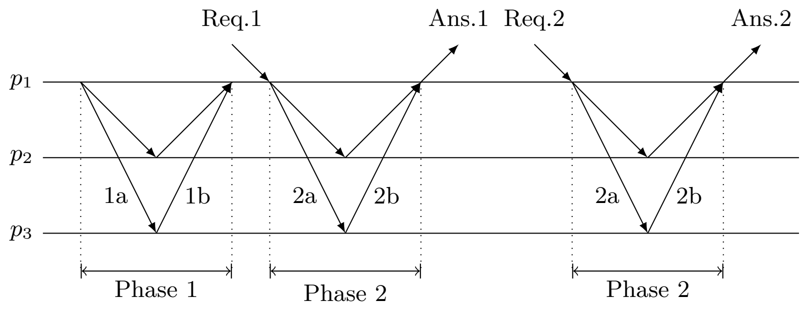


Figure 2: Message pattern of MultiPaxos

3.1 The Paxos algorithm

Paxos is traditionally described as the interaction of processes in three different roles: proposer, acceptor and learner. For logical clarity, we use these roles in our description of Paxos, but in our implementation each process assumes all three roles.

In Paxos, processes execute a series of one or more ballots (Fig 1). In each ballot one of the proposers, called the leader of the ballot, tries to get the others to agree on a value proposed by it. The ballot may succeed, in which case the value proposed is decided, or it may fail, in which case some other process (or the same) starts a new ballot. Ballots are numbered with increasing numbers, with higher-numbered ballots superseding lower-numbered ballots. Proposers chose ballot numbers among non-overlapping sequences, that is, each ballot number can only be assigned to a single proposer. This is necessary to ensure that there is a single leader for each ballot number.

Each ballot consists of two phases, as shown in Figure 1. In the first, the Proposer sends a PREPARE message to the acceptors (Phase 1a message) asking them to abandon all lower numbered ballots and to reply with the last value they accepted and the ballot number where they accepted it, or null if they did not accept any value (Phase 1b message). The acceptors answer to this message only if they have not participated in any higher numbered ballot. Once the proposer receives a majority of replies, it enters the second phase. It choses a value to propose based on the messages received from the acceptors. If some Phase 1b messages contain a value, then it takes the value associated to the highest ballot number, otherwise the proposer is free to chose any value. Once the value is chosen, the proposer proposes this value with a Phase 2a message sent to all acceptors. The acceptors will once again answer only if they have not participated in an higher numbered ballot, in which case they send a Phase 2b message to all learners informing them that they have accepted the proposal. A learner decides the value once it receives a majority of Phase 2b messages for the same ballot number.

3.2 MultiPaxos

MultiPaxos is an extension of Paxos for state machine replication, where processes agree on a sequence of values instead of a single value. It would be possible, but inefficient, to use a sequence of independent Paxos instances. Instead, MultiPaxos provides a substantially lower message complexity by “merging” the execution of several instances.

MultiPaxos is based on the observation that when executing a series of consensus instances, a proposer

```
Initialisation
    $view \leftarrow 0$                          used to recognize voting rounds
    $\{view_v, value\} \leftarrow \{0, \perp\}$   last accepted value and the view when the accept took place
    $procId \leftarrow$ ID of the process
    $accepted \leftarrow \{\perp\}$               set of processes which accepted the value in current view
    let $leader(view)$ be function that for a $view$ returns the leader process ID

Prepare phase
    $view \leftarrow v$ such that $v > view$ and $leader(v) = procId$
    send PREPARE$< view_m >$ where $view_m \leftarrow view$ to all
    wait for majority of PREPAREOK$< view_m, \{view_p, value_p\} >$ where $view_m = view$
    $\{view_v, value\} \leftarrow \{view_p, value_p\}$ from PREPAREOK with highest $view_p$
    begin Propose phase

Propose phase
    if $value = \perp$ then $value \leftarrow$ the value the process wants to propose
    $view_v \leftarrow view$
    send PROPOSE$< view_m, value_m >$ where $view_m \leftarrow view;\ value_m \leftarrow value$ to all

Upon PREPARE$< view_m >$ where $view_m \geq view$ from $p$
    $view \leftarrow view_m$
    send PREPAREOK$< view_m, \{view_p, value_p\} >$
                        where $view_m \leftarrow view;\ \{view_p, value_p\} \leftarrow \{view_v, value\}$ to $p$
    $accepted \leftarrow \perp$
    leave Propose or Prepare phase if process is in any of these

Upon MESSAGE$< view_p, \dots >$ where $view_p \neq view$
    if $view_p > view$ then
        $view \leftarrow view_p$
        $accepted \leftarrow \{\perp\}$
        leave Propose or Prepare phase if process is in any of these
        react according to message type
    else drop the message

Upon PROPOSE$< view_p, value_p >$ where $view_p = view$ from $p$
    $\{view_v, value\} \leftarrow \{view_p, value_p\}$
    send ACCEPT$< view_m, value_m >$ where $view_m \leftarrow view_p;\ value_m \leftarrow value_p$ to all
    $accepted \leftarrow accepted \cup \{p, procId\}$
    if $accepted$ contains majority of processes then
        decided on $value$

Upon ACCEPT$< view_p, value_p >$ where $view_p = view$ from $p$
    if $view_v \neq view_p$ then
        execute Upon PROPOSE$< view_p, value_p >$
    $accepted \leftarrow accepted \cup \{p\}$
    if $accepted$ contains majority of processes then
        decided on $value$

Upon Value for voting received
    begin Prepare phase

Upon no decision taken and the leader crashed
    begin Prepare phase
```

Table 1: Pseudocode of the Paxos algorithm

can execute Phase 1 for an arbitrary number of instances using a single prepare phase. Afterwards, it only needs to execute Phase 2 of each instance, therefore reducing the number of communication delays for an instance from 4 to 2. Figure 2 illustrates the message pattern of MultiPaxos.

Instead of describing MultiPaxos in its most abstract form, we will instead present it as implemented by JPaxos (Table 1), which matches the way it is commonly described in more practical oriented works [17, 11]. This description specifies some design decisions that are left open by the original description in [1].

In MultiPaxos the system advances through a series of views, which play a similar role as ballots in single instance Paxos. The leader of each view is determined by a rotating coordinator scheme, that is, the leader of view v is process $v \bmod n$. Once a process p is elected leader (by an external leader oracle module), it advances to the next view number v such that p coordinates this view (*i.e.*, $v \bmod n = p$) and v is higher than any view previously observed by p. Process p then executes Phase 1 for all instances that, according to the local knowledge of p, were not yet decided. It does this by sending to the acceptors a message $\langle \textsc{Prepare}, v, i \rangle$, where v is the view number and i the first instance for which it does not know the decision. The acceptors answer with a message containing the Phase 1b message for every instance of consensus higher than i, *i.e.*, for every instance $i' \geq i$, the acceptors send the last value they accepted and the corresponding view number, or null if they have not accepted any value for this instance. Although this message covers an infinite number of instances, only a bounded number of them will have non-null values, so this information can be represented as a message with finite size (proportional to the number of non-null instances higher than i). After completing the Phase 1, the proposer can then execute Phase 2 for every instance equal or higher than i.

In state machine replication, when the leader receives new requests from the clients, it executes the Phase 2 of a new instance, which requires a single round of communication from leader to acceptors and then from acceptors to learners.

4 Architecture

4.1 Main modules

JPaxos consists of two main modules: Replica and Client. The *Replica* module executes the service as a replicated state machine, while the *Client* module is a library that is used by client applications to access the service. JPaxos also defines a Service interface that must be implemented by the service being replicated. Figure 3 shows a typical deployment of JPaxos.

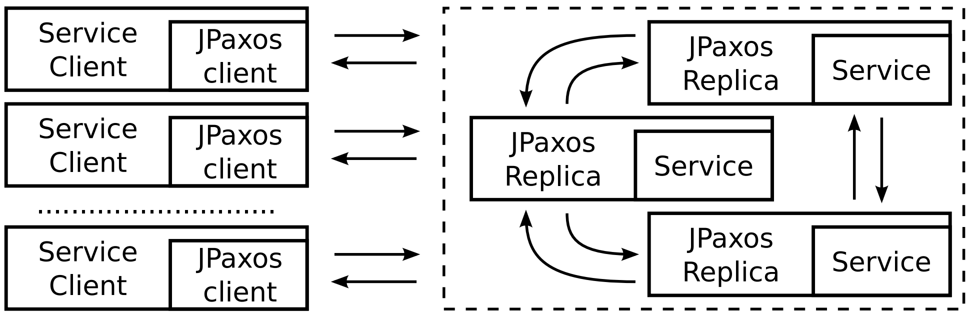


Figure 3: A service replicated in three replicas accessed by several clients. Arrows indicate communication flows.

Below we describe each of these modules.

4.1.1 Client

The `Client` module is a light-weight module that mediates the interaction between the application and the replicated service. This component hides from the client application much of the complexity of involved in sending requests to a replicated service. As requests must be sent directly to the leader, the client library discovers and connects to the leader using a persistent TCP connection. It then sends the request and waits for an answer. If the connection to the leader fails or the answer does not arrive within

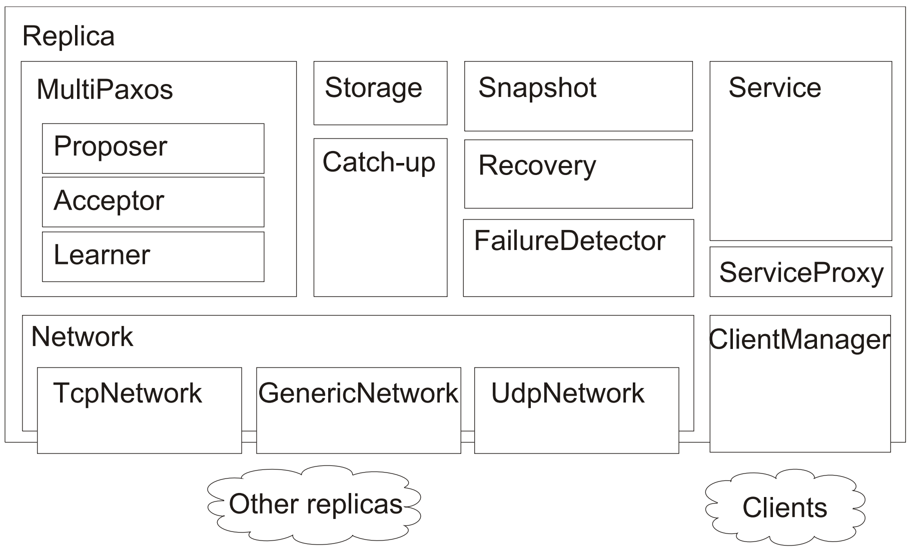


Figure 4: Block diagram of JPaxos modules

a certain time, the client library assumes that the leader failed, discovers the new leader, connects to it and retransmits the request. The client is also responsible for ensuring that every request is assigned a unique global id, which is required to detect duplicate requests.

4.1.2 Service

The service implementation is provided by the user of JPaxos and is the module that is replicated. As mentioned before, it must be deterministic. The service must implement several lifetime management methods including request execution, creating a snapshot of the state and recovering the state from a snapshot.

4.1.3 Replica

The Replica module replicates the service using the MultiPaxos protocol. Additionally, it implements many other modules responsible for managing the state and lifetime of the service, and interacts with the clients. Figure 4 shows the sub-modules of the Replica module.

We now briefly describe each module.

ClientManager This module manages the communication between clients and the replica. In particular, it listens for new TCP connections from the clients, receives the client requests, forwards them to the other modules for execution, and finally sends the answers back to the client. Client connections are persistent, which amortizes the cost of connection establishment over several requests.

MultiPaxos This module is responsible for establishing a total order among the client requests using the MultiPaxos algorithm. It is organized in three sub-modules, the Proposer, Acceptor and Learner, matching the traditional roles used to describe Paxos.

Storage Manages the state of the MultiPaxos protocol, including the replicated log with the state of all instances started and auxiliary variables, like view number. The state is centralized in this module in order to simplify its management.

CatchUp During execution some replicas might miss the decision of some ballots, due to message loss or a crash, in which case they need to fill up the resulting gaps in the command sequence in order to continue executing requests. The CatchUp module is responsible for discovering the decisions of missing ballots, by retrieving them from other replicas. This service is also used for recovery after a crash; after

Variable	Description
log	the Paxos Log (see section 4.2.1)
view	current view
firstUncommitted	first instance not decided yet
snapshot	the most recent snapshot (see section 5.4)
epoch	the current epoch vector (see section 7.5)

Table 2: State kept by Storage module

Variable	Description
view	a view of the last received message related with this instance.
value	the value which is held by this instance.
state	one of UNKNOWN, KNOWN or DECIDED.
accepts	set of known replicas which accepted the (view, value) pair.

Table 3: State of a consensus instance

the service state is reinitialized from a snapshot, the CatchUp service brings up to date the state of the replica with any new commands that may have been decided since the snapshot was taken.

Snapshot This module performs periodic snapshots of the state of the service to stable storage.

Recovery Responsible for implementing recovery after a crash. This module supports several recovery algorithms, with different trade offs between resilience and performance. During normal execution, and depending on the algorithm chosen, this module logs to stable storage some information about the progress of the algorithm and adds some additional fields to the messages sent between replicas. During recovery, this module restores the state of the service from a snapshot or from the state of other replicas. We provide more details in Section 7.

Service Proxy Interposes between the *Replica* and the *Service* provided by the user, and performs bookkeeping work that is required to manage snapshotting. Section 5.4 describes it in more detail.

4.2 Storage and data structures

The protocol state kept by JPaxos consists mainly of the replicated log and a few auxiliary variables describing the current state of the process. These data structures are accessed by almost all modules of JPaxos, including the ordering protocol, snapshotting and catchup. In order to simplify management of the state, we aggregated all the state that is shared between modules in the *Storage* module. Table 2 shows the state kept in the *Storage* module.

Depending on the recovery algorithm used, the *Storage* module decides which elements to keep only in volatile memory and which to write also in the stable storage.

4.2.1 The Paxos Log

The Paxos log is the data structure that is actually replicated among the replicas. Each replica contains its own copy of the log, with the replication algorithm being responsible to update the log consistently in all replicas. The log contains a series of entries, each describing what the replica knows about the state of a particular consensus instance. Table 3 shows the information that is kept on the log for each consensus instance.

An instance is in the Unknown state if the replica did not start this instance yet, in the Known state if the instance was started but not yet decided and finally it changes to the Decided state when the final decision is known.

Paxos is often presented using an infinite log, which simplifies its description considerably. But in practice the log must be bounded. JPaxos achieves this by performing periodic snapshots of the service state and keeping in the log only the instances that had not been executed at the time of the snapshot.

This hybrid representation of the state affects the catch-up and recovery mechanisms, as they must be ready to restore the state from a combination of a snapshot and replaying log instances instead of replaying only log instances. We discuss these details below: the catchup mechanism is described in Section 5.3, snapshotting in Section 5.4, and recovery in Section 7.

4.2.2 Auxiliary Storage state

The other variables kept by the Storage module are used to track the current state of the protocol execution: *view* is the current view of the process, *firstUncommitted* is the id of the lowest instance that was not yet decided, *snapshot* is the last snapshot taken by the local replica, and *epoch* is a vector with the epoch number of the replica (used by the recovery algorithms).

4.3 Communication

4.3.1 Client-Replica communication

JPaxos uses TCP to communicate between the clients and the replicas. Compared to UDP, TCP provides high-quality implementations of important functionality that is required in this scenario, like flow control and support for messages of arbitrary size, thereby simplifying our implementation. It also provides reliable retransmission within a TCP connection, although if a connection fails, it does not provide any guarantees for data sent but now acknowledged. JPaxos client library handles this case by retransmitting the last request sent but unanswered when recovering from a broken connection.

The client library hides from the application most of the complexity of accessing a replicated service. The first time the client tries to execute a request, the client library selects a random replica and tries to establish a connection. If this fails, the client library tries to connect to the next replica, looping back to replica 0 after trying $n-1$, until it connects successfully to a replica. After connecting to a replica, it sends the request and waits for the answer. If JPaxos is configured to require the replicas to connect directly to the leader (See Section 4.3.3), then non-leader replicas answer with a REDIRECT reply containing the id of their current leader. The client then reconnects to the replica with the id given in the reply, and resends the request. This process is repeated until the client obtains the answer to the request.

4.3.2 Communication between replicas

JPaxos supports TCP and UDP for inter-replica communication. This flexibility is necessary because of the tradeoffs between the different communication protocols in the context of state machine replication; there is no single protocol that will perform optimally in all deployment scenarios. For instance, UDP may have lower overhead and latency but are more susceptible to network congestion, while TCP offers more stability but may limit the maximum throughput. Additionally, depending on the network and on the application, some protocols may behave better than others. Since JPaxos is also a research project, supporting multiple protocols opens the door to conducting research into this topic.

TCP network Connections between the replicas are persistent. If a connection fails, the replicas will try to reestablish periodically, as the other replica may recover or it may have been only a connectivity problem. JPaxos retransmits the last message sent to a replica if the previous connection failed and the message must still be delivered (*i.e.*, if the corresponding consensus instances was not yet decided).

UDP network When using UDP, JPaxos retransmits the messages that must be delivered for the protocol to proceed until they expire. This is the case of the Phase 1a and 2a messages, which are retransmitted to the replicas that have not yet answered with the corresponding Phase 1b or 2b message. Retransmission stops when the leader does not need any more answers, which happens when it receives a majority of messages in the corresponding Phase.

4.3.3 Request forwarding or client redirection

In order to execute a request, the leader must first receive the request. This can be done in two ways in JPaxos: directly or with request forwarding. In the first option, clients connect directly to the leader,

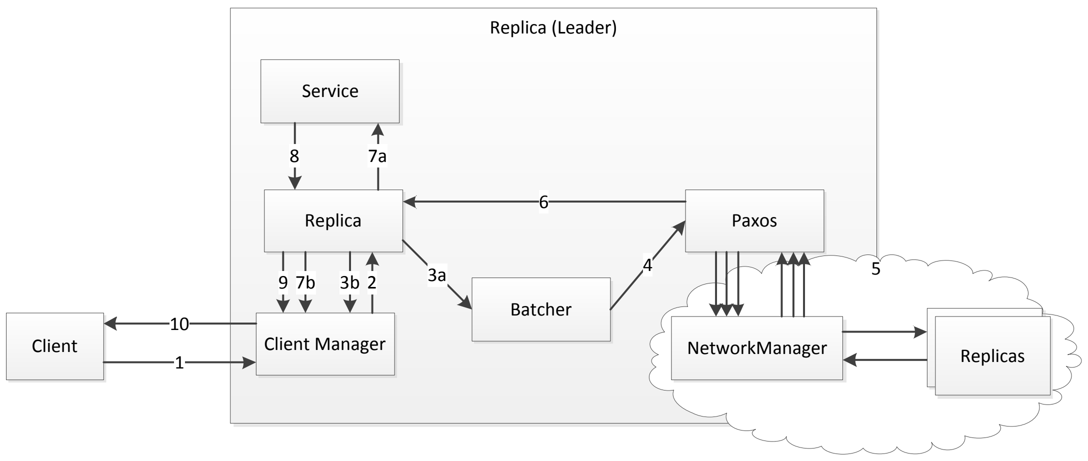


Figure 5: Request handling, when client is connected directly to the leader. 1) Client sends request, 2) request read and forwarded to Replica module, 3a) request added to batch queue (new request) or 3b) send cached answer (repeated request), 4) propose request as part of a batch, 5) order batch using MultiPaxos, 6) after being ordered, batch is given to Replica for execution, 7a) Replica executes request in service if the request is new, or 7b) answers with cached reply if request is repeated, 8), 9) and 10) answer is sent back to client.

which is then responsible to receive the requests and send the answers back to the clients. As the leader can change dynamically, the clients must discover the leader, which is done by having non-leader replicas redirect clients to the current leader. In the second option, the clients are allowed to connect to any replica. When a replica receives a request, it assumes ownership of the request: it will keep trying order and execute the request. It does so by forwarding the request to the current leader, retransmitting if necessary, either to the same leader to recover from message loss, or to a new leader when the view changes. The replica that received the request will also send the answer back to the client when it finally executes the request.

4.4 Request Handling

Figure 5 shows how a request is handled by JPaxos in the case of a client connected directly to the leader. The only difference in the case where the client is connected to a non-leader replica, is that the non-leader replica acts as a client with the leader, using the same mechanisms as a client would use. Clients send requests sequentially, waiting for the answer of the previous request before transmitting the next (1). When a replica receives a request (2), it first checks if the request was already executed, in which case it answers with the cached reply (3b and 10). In Section 5.1.2 we discuss in more detail how duplicate and lost requests are handled. If the request is new, it is dispatched for ordering and execution. For performance reasons, JPaxos executes the ordering protocol on groups of one or more requests, batches, instead of on individual requests. Therefore, before being proposed, the request is added to the Batcher module queue (3a), where it waits until it is included in a batch (as explained in Section 6). The batch is then passed to the Paxos module (4), which proposes and eventually decides on a order for the batch (5). Once ordered, the batch is given back to the Replica (6), which extracts the requests contained within. The individual requests in a batch are ordered in relation to each using their request id. The replica must once again check if the request has been executed previously and, if so, either ignore it or answer with the cached reply (7b) (Section 6 describes how this can happen). If the request was not executed previously, it is executed in the service (7a) and the answer is sent back to the client (9 and 10).

5 Implementation

This section describes the implementation of the main modules of JPaxos.

5.1 Replica

The replica module is central in JPaxos, coordinating the work of all other modules. It handles the interaction with the clients and with the service: receives requests from the clients, passes them to the Paxos core for ordering and once they are ordered, executes the requests and sends back the answers to the client. Although the fault-tolerance properties of state machine replication come primarily from the MultiPaxos algorithm, the replica module plays also an important role in ensuring consistency and liveness of the system. In particular, it is responsible to redirect clients to the leader, to retransmit answers to duplicate requests sent by the clients, to ensure that every request is executed only once and to manage the lifecycle of a service, including keeping enough information to perform regular snapshots, and restoring the service from a snapshot.

Next we describe in more detail the main tasks performed by the Replica module.

5.1.1 Generate unique request identifiers

To cope with message loss, clients may retransmit a request multiple times. This may result in a request being received multiple times by the same replica or by different replicas (if the leader changes between retransmissions). Although the safety properties of the replicated state machine would not be violated by executing the same request multiple times, this is usually not the semantic that applications are expecting. Therefore, JPaxos enforces once-and-only-once execution.

Each request must have a unique identifier in order to be distinguished from the others. A common method of doing so is by using the pair $\langle \mathit{clientID}, \mathit{seqNumber} \rangle$, where *clientID* is a unique identifier for the client that sent the request and *seqNumber* a sequence number private to the client. The main difficulty is ensuring that uniqueness of the *clientID*.

Using the IP address of the machine or even the MAC address is not a good solution, as this prevents multiple clients from operating in the same machine. Additionally, configuration errors or the use of DHCP may result in the same IP address being assigned to different machines, violating uniqueness. Using statically assigned IDs would also be possible, but requires some administrative work and is susceptible to human error.

In JPaxos we opted for making the replicas responsible for granting IDs to clients: when the client first establishes a connection to a replica, it will ask for a unique id. JPaxos supports two policies to generate unique client IDs, as described below.

Based on the number of replicas Each replica grant numbers incremented by a number of replicas, starting from ID of a local replica. For example, if we have three replicas, replica with ID 0 can grant numbers 0, 3, 6, 9, ... and replica with ID 2 can grant 2, 5, 8, ... It guarantees that two different replicas will not grant the same ID. This solution works with the crash-stop and crash-recovery with stable storage. This will not work when replica can recover from crash without stable storage, because it does not know what was the last granted ID.

Time based This policy uses the system time as the source of unique numbers. To each client ID we also add information when the replica was started (in milliseconds). For example, if a replica has been started at time t, it will grant to the new clients the following IDs: $(t{+}\mathit{localId})$, $(t{+}\mathit{localId}{+}n)$, $(t{+}\mathit{localId}{+}2n)$ ... This method assumes that the local system timer is not set back into past and also that the replica will not recover in less time than what it takes for the clock to advance once. If these assumptions are fulfilled, then unique IDs are guaranteed also in the crash-recovery model without stable storage.

5.1.2 Ensuring once-and-only-once execution

To handle retransmission of requests, the replica module keeps a cache of the last answer sent to each client. When a request is received, it first checks this cache and, if the request is present, sends the answer back to the client immediately. This cache is checked again before executing an ordered request because as explained in Section 6, the same request may be ordered twice.

It is enough to keep the last request executed from each client because the client library sends request synchronously, that is, it must receive the answer to request r before sending request $r + 1$. Therefore, if a replica receives request $r + 1$ from a client, it knows that this client already has the answers to all requests lower than $r + 1$.

5.1.3 Service lifecycle

A final task of the Replica module is in assisting the service in performing snapshots.

Every checkpoint must be tagged with some additional information that indicates what was the last request executed before the checkpoint, so that on recovery the replica knows from where to continue executing requests. Instance numbers seem like a natural candidate for this, but this solution has a significant drawback. The problem is that JPaxos assigns instance numbers to batches of requests, instead of to individual requests. Therefore, if using instance numbers alone, the service can only make a snapshot at the boundaries of batches. Although this may not be an issue for most services, some may only be able to take snapshots at particular points in their execution.

In order to allow the service to perform checkpoints after any request, the Replica module assigns a sequence number to every request, therefore mapping the sequence of batches into a continuous sequence of requests. This information is stored together with the snapshots.

The Replica module also stores with the snapshot the cache with the last request executed from each client. This is necessary to ensure once-and-only-once execution even between crashes and recoveries.

5.2 Multi-Paxos

Our implementation of MultiPaxos is based on the description in [1], following closely the structure of Proposer, Acceptor and Learner, that later became the standard way of presenting Paxos. Here we discuss only the details that are left unspecified in the original document and present some additional optimizations.

5.2.1 Leader Election and Proposal Numbers based on views

MultiPaxos requires both a leader election oracle and a mechanism to assign to each process an infinite number of exclusive proposal numbers. Both details are left unspecified in [1], as their implementation details are not relevant for the correctness of the core protocol.

JPaxos uses view numbers to implement both leader election and to generate proposal numbers. The ordering protocol is organized as a sequence of views with increasing numbers; processes keep track of their current view v and tag all their messages with the view number where they were sent. Any message from a lower view is ignored, and receiving a message from a higher view forces the process to advance to the higher view immediately.

For each view v, the process $v \bmod n$ is pre-assigned as the leader of that view. Proposal numbers are generated using the view number and adding a sequence number internal to that view, *i.e.*, the leader of view v uses the numbers $\langle v, i\rangle$, where i is the sequence number generated by the leader. The order among proposal numbers is defined first by the view number then, in the case of a tie, by the sequence number, that is, if $v_1 < v_2$ then all proposals $\langle v_1, -\rangle$ are lower than proposals $\langle v_2, -\rangle$.

Leader election is implemented by advancing view whenever the leader of the current view is suspected to have failed. When a process suspects the leader, it tries to become the leader, by advancing to the next view v that is assigned to it and sending a PREPARE message to all. It may happen that several processes suspect the leader at approximately the same time and try to become leaders themselves. In this case, the process that chooses the highest view number will be the one that succeeds in becoming leader, as higher views take precedence over lower views.

To detect the failure of the leader, JPaxos uses a simple failure detector based on heartbeats. The leader sends an ALIVE message every τ time to all processes using UDP. When a replica does not receive an heartbeat from the leader for more than η time, it suspects the leader and tries to become the new leader. Both τ and η are configuration parameters. In a typical configuration, η is at least 2 times larger than τ, in order to be immune to delays and single message losses.

5.2.2 Optimizations

JPaxos uses several optimizations to reduce the number of messages sent. Many of these optimizations are possible because in JPaxos every process is at once Proposer, Acceptor and Learner, so the different actors running in a process share the same replicated log. By compromising modularity somewhat and exposing the state of each actor to the other actors, it is possible to suppress many messages and improve performance significantly.

Sending to self The Proposer has to send the Phase 1a and Phase 2a message to all Acceptors. In JPaxos, as the leader plays the role of both Proposer and Acceptor, the leader can suppress the message to itself, instead updating directly its state. A similar optimization can be applied when the Acceptor sends the Phase 2b message to all Learners. These optimizations reduce from n to $n-1$ the number of Phase 1a, Phase 2a and 2b messages sent, which is a significant reduction when n is small.

Merging the Phase 2a and 2b messages of the leader In Phase 2, the leader has to send, as the Proposer a Phase 2a message to all, and immediately after, as an Acceptor, a Phase 2b message also to all. Since inside the leader the Acceptor role will always accept the message sent by its Proposer role, these two messages can be merged in one. Therefore, in JPaxos the leader sends only a Phase 2a message to all other processes, which is understood as an implicit Phase 2b message. Notice, that if the system consists of three nodes, every process decides on the value by receiving a single Phase 2a message, because it knows that the leader and itself accept the message. This optimization reduces in half the messages sent by the leader during Phase 2.

Minimizing the count of messages carrying the value For simplicity, most descriptions of Paxos state that both the PROPOSE and the ACCEPT messages carry the value being agreed upon, which in our case are client requests. The size of the requests is arbitrary, as it depends only on the service. If the size of the request is large, then including the consensus value in all Phase 2 messages is likely to be inefficient as the value is sent two times to each process: the leader sends it to all in the PROPOSE message, and each acceptor resends it to all in the ACCEPT message. In these situations, it is possible to perform consensus on value identifiers and use some sort of reliable broadcast to propagate the value to all replicas only once[18]. However, this approach introduces additional complexity in the protocol.

JPaxos ensures that the value is sent only once to each process by omitting it from the ACCEPT messages and relying on the PROPOSE message of the leader to distribute the value to all replicas. To preserve correctness, the protocol of the Acceptor has to be modified slightly: if the Acceptor receives an ACCEPT message before the corresponding PROPOSE message, it must wait until receiving the corresponding PROPOSE message before sending its own ACCEPT. This is necessary because in Paxos, a process must know the value of a proposal before accepting it.

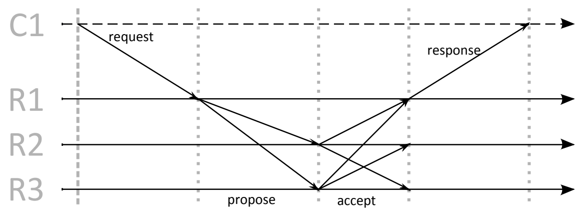
In the good case, this optimization will reduce in half the amount of data sent over the network without any additional delay. Even if the ACCEPT message is delivered before the PROPOSE message, the delay will be minimal. The only drawback is that if the PROPOSE message is lost, the Acceptor has to wait until the leader retransmits this message while without this optimization the Acceptor could proceed immediately. This may delay decision with message loss[1]. But since message loss is usually the result of congestion and this optimization reduces the amount of data sent over the network, overall it should improve performance in all cases.

Best-case messages Figure 6 shows the message pattern and of a Phase 2 instance in the good case.

For each Phase 2 instance, the replicas exchange a total of $(n-1)+(n-1)^2$ messages, but out of those only $n-1$ contain the value. The other $(n-1)^2$ messages should not be a problem in the common case, as they are small (less than 20 bytes) and most Paxos deployments use a small number of replicas (3 or 5).

It is possible to decrease the number of messages in Phase 2 to $2(n-1)$ by using the leader as a distinguished learner, in which case the Acceptors send a single ACCEPT message to the leader. The leader then has the responsibility to inform the other processes of the decision, usually by piggybacking the decision on the Propose message for the next consensus. We have not implemented this optimization,

[1]Only if a majority of PROPOSE messages are lost



(a) Best-case scenario

Message type	Count	Size of each msg
PROPOSE	$n - 1$	|value|
ACCEPT	$(n - 1)^2$	$c(< 20\text{bytes})$

(b) Message count

Figure 6: Messages sent in Phase 2 instance

as it adds additional complexity to the protocol and the potential gains did not seem to be sufficient to justify it.

Suppressing heartbeat JPaxos uses the protocol messages (Phase 1 and 2) received from the leader as implicit heartbeats. Consequently, the leader sends explicit heartbeats only if it has been idle for more than τ time (recall the τ is the heartbeat send interval). Although this optimization reduces slightly the network load, its main advantage is preventing false suspicions during periods of heavy load. Recall that heartbeats are sent using UDP, so they are susceptible of being lost when the load is high. But by using the protocol messages as implicit heartbeats, the leader will not be suspected as long as at least a few protocol messages arrive to the other replicas, which is likely even when the load is very high.

5.3 Catch-up

MultiPaxos must guarantee that all learners eventually learn the decision of every instance. This is particularly important because requests must be executed sequentially, so a gap in the sequence at one process will block that process from executing further requests. This can lead to instability, because until the gap is closed, the process must either cache all the subsequent decisions which can lead to unbounded memory usage, or it has to stop participating in the protocol.

In the good case, the learners decide an instance when they receive the PROPOSE message and a majority of ACCEPT messages, and therefore any gap that may happen due to out-of-order decisions [2] is only temporary as the required messages will arrive in at most one message delay. But in networks with message loss, some of these messages might be lost. Although the leader will keep retransmitting the PROPOSE message until it receives a majority of ACCEPT messages, this does not guarantee that all processes will receive enough messages to decide.

For these situations JPaxos includes a catch-up mechanism. The catch-up mechanism is based on the following observation: if the leader is correct (as indicated by its heartbeats) and a process knows of an instance that is started some time ago but was not yet decided, then the value is likely already decided and the process should contact some other process to learn the decision.

5.3.1 Log-based vs state-based catchup

A replica can catch-up either by copying the missing decisions or by a combination of transferring the state plus the most recent decisions. In the rest of this report *log-based catchup* refers to the first option while *state-based catchup* refers to the second option.

Log-based catchup is done by transferring the missing commands and executing them at the replica. Transferring and executing the requests can be done concurrently, so the total duration can be approximated by the maximum time required to transfer all the updates or to execute them.

State-based recovery is performed in four steps:

- **Serialization** - Create a snapshot with the current state of the service.
- **Transfer state** - Transfer the snapshot to the other replica.
- **Deserialization** - Restore the state of the destination replica using the snapshot.

[2]Executing instances in parallel can lead to out-of-order decisions, see Section 6

- **Residual log-based catchup** - Execute commands already ordered that are not included in the snapshot.

The first three tasks may be done concurrently.

Which method is faster depends on the several factors, like the size of the state, size of the requests, the execution time of requests, and the bandwidth available. Log-based catchup transfers only the commands that are missing, therefore avoiding transferring the full state. The time it takes to transfer the commands depends on the number of commands to be transfered and on the size of each command. With state-based recovery, the time to transfer the state depends only on the size of the state. Another factor to consider is that in log-based recovery the commands must be executed at the destination replica, which takes time, while in state-based recovery it is only necessary to replace the state, which is typically fast. The above discussion suggests that the strategy must be chosen according to the characteristics of the service. As described below, JPaxos implements both strategies, choosing dynamically the one that is likely to perform better based on the current conditions.

5.3.2 Starting and stopping catch-up

Starting Catch-up Activating the mechanism should not happen too early, that is when the process might still able to decide by receiving enough Propose and Accept messages. On the other hand late initiating leads to delaying command execution, and thus significant performance loss.

JPaxos uses a combination of the following triggers to initiate catch-up:

- periodical Activate periodically the mechanism.
- higher instance decided An instance with higher ID (i.e. newer instance) has already been decided
- window Instance with ID higher than α has been started, and the implementation ensures that only instances up to α are started as $\alpha - ws$ stays undecided.
- timeout No traffic for the ballot during a particular period of time

The simplest strategy is to activate catchup periodically, regardless of the state of the system, thereby guaranteeing that from time to time the replica will bring itself up-to-date. However this method may trigger unnecessary catchups and may take too long to catchup, leaving the replica with stale state for too long.

A better strategy is to start catchup when the replica detects a gap on the sequence of commands, that is, instance i is undecided while there are instances higher than i already decided. We may assume the previous voting took similar amount of time – that is, it should be already finished. This trigger is susceptible to false positives, because varying network latency may cause the situation that a newer consensus is decided as the older is still in progress. The false positives may be minimized by waiting until at least a certain number of instances higher than the missing instance are decided before initiating catchup.

The window method relies on the fact that JPaxos executes in parallel at most *WND* instances, where *WND* is the maximum window size (See Section 6). Therefore, if a replica receives a message for instance $i + \mathit{WND}$ or higher, then it can infer that instances i and lower were already decided at least by the leader. Like in the previous strategies, the missing messages might be delayed and arrive in the future, so the replica waits some time before initiating catch-up.

The timeout trigger handles the case where some messages from the last instance were lost and no further instances were started. In this case, the previous two triggers would never occur and the replica would remain without knowing the last decision.

A different problem is when a replica does not know that the instance already exists (for example, a burst of message loss caused all messages for the replica to be dropped). In this case, the catch-up should be started as well. JPaxos addresses this case by including in the Alive message from the leader the highest instance ID started, which guarantees that every replica learns what instances already exist within the failure detector timeout, and therefore can start catch-up if they have gaps on their command log.

Stopping catch-up As the algorithm runs, the state of Paxos may not be stable, *i.e.*, new ballots may start. Therefore selecting the moment when catch-up should be deactivated requires some thought.

One method is to calculate conditions a priori (for example the IDs of missing instances), and continue the process as long as needed. However, this can easily lead to constant switching on and off the catch-up – voting for new instances may be faster than catching up, so as the predefined conditions are met, new event already caused catch-up activation.

A better solution is to determine dynamically if catch-up is still needed. The method used by JPaxos consists of checking if all instances outside the window are already decided. This ensures that all instances outside the window as is currently known by the recovering replica will be discovered by the catchup mechanism. The instances that are inside the window might not be decided yet, and in this case it is preferable if the replica learns them by participating in the voting than by relying on the catchup.

5.3.3 Transport protocol

The catch-up may use a different transport protocol than the consensus algorithm. Choice of the transport layer must take into account the characteristics of this mechanism as well.

Both TCP and UDP may be used – both having their pros and cons, presented in the table below:

TCP	**UDP**
automatic retransmission guaranteed by protocol	retransmission must be implemented manually
flow control provided by protocol	no flow control, it *is* easy to congest network
big messages automatically fragmented, merged and managed	splitting and handling big messages must be self implemented
request cannot be changed at retransmission	request can be changed each retransmission
default retransmission time	custom retransmission time
another message may be sent once the previous was delivered	new datagram may be sent anytime, no matter if the previous reached the target

When catch-up is needed, our network must have (probably high) message loss. That means the catch-up messages may also get easily lost.

We have noticed, that it is more efficient to use UDP for all smaller messages – if the package gets lost, we are retransmitting it with updated query. Sometimes a UDP message retransmitted even twice reaches its target faster then the same message sent with TCP. In TCP, the timeout for the ACK message grows automatically – and that may cause big delays. During experiments, with 30% message loss, transmitting a message using TCP that is smaller than the UDP-datagram size took even more than 4 seconds to reach the other side.

In UDP, a single message delay or loss does not block the communication to the replica – but in TCP, it does. In this case, no new catch-up query may be sent to this replica, as long as the previous will be processed. The request cannot of course be changed. It means that once we got the response, the core protocol may have already missed another instances to catch-up.

5.3.4 Requirements

The speed and the resource consumption must be correctly balanced.

In theory, catching-up can be delayed any finite time. On the other hand, it is important for the view changes and for bounding the size of the log. A view change will take longer if the new leader is missing many requests. Since during a view change no new requests are being satisfied, we need to keep its length as short as possible. Additionally, if a replica does not know the instance i, it cannot execute any request higher than $i - 1$. This also means that it cannot remove entries from the log.

Catch-up must perform well while at the same time consuming few resources, so that it does not slow down significantly the core protocol. Good performance demand has one main reason: the arrival time of the tasks for the state machine is dependent on being up to date. If catch-up would get the information too late, the state machine would get long list of, possibly high CPU-consuming, tasks at once, instead of doing them in spare time before.

Therefore, it is desirable to run the catch-up mechanism as often as possible, but without slowing down the service significantly.

It should be noticed, that only the followers would have to catch-up, since the primary learns all previous decisions on a view change and then it is the one proposing new values and leading the protocol.

5.3.5 Catch-up algorithm

The catch-up algorithm should contact other replicas and copy the decisions that are not known. The question when this should be done is discussed in Section 5.3.2. Here the main idea of JPaxos catch-up algorithm is presented.

To activate the catch-up, we use WINDOW and PERIODICAL methods (see Section 5.3.2). The leader also sends in each ALIVE message the information about highest started instance. This ensures that the replica will catch-up eventually, even if there are no new requests, and that a replica never stays too much behind the others, at most *WND*.

There are three messages that may be sent during the algorithm: CATCHUPQUERY, CATCHUP-RESPONSE and CATCHUPSNAPSHOT.

- CATCHUPQUERY - a request for missing instances. It carries a list of missing IDs and may have one of two flags set: the first flag indicates if this was a periodical catch-up, second one – if we want to get the last snapshot, not the missing instances.
- CATCHUPRESPONSE - response sent for every received request. It has a list of decided instances for requested numbers. The list may be empty. This message may have two flags: periodical flag and snapshot only flag. The periodical flag is set if this is response to the periodical query. The snapshot only flag is set if the replica does not have an old enough instances, and transferring a snapshot is the only possibility.
- CATCHUPSNAPSHOT - if another replica asked for snapshot, this message is sent with most recent snapshot.

The list describing missing instances is constructed by replica in straightforward way: it contains all undecided instance numbers plus the (highestID+1) as the first instance, the replica has no knowledge about. The responder sends therefore all decided instances from the list and all decided instances that are higher or equal the additional number.

In order to make the messages smaller, a trick has been used: the list contains two sublists. One, called a range list and the other, an instance list. The range list contains intervals we miss, while the instance list – single numbers. For example, if we would miss instances 1, 2, 4, 6, 7, 8, 9, 11, and the highest instance we know is 12 (state decided) our lists would look like:

$$[\langle 1, 2\rangle; \langle 6, 9\rangle]; [4, 11, 13]$$

Notice the number 13 – it is the first we have no idea of existing, as mentioned above.

Algorithm 1 presents the catch-up algorithm used in JPaxos.

In line 2 a best replica is chosen. We have implemented it as follows: a rating for each replica is kept. When sending a message, the rating decreases; when receiving response it rises. If an empty response is received (except for the PERIODIC mode), we request asking the leader next time. The best replica for us is a follower with the highest positive rating, or the leader if all followers have negative rating. The line 6 executes a predicate checking if the catch-up shall finish (see Section 5.3.2).

5.4 Snapshotting

As presented before, the support of storing and transmitting state of replicated machine is a very useful and important part of a replica. In practical systems, it is necessary – it allows log truncating, faster recovery and catch-up.

5.4.1 When to snapshot

Snapshots are needed to enable catch-up and recovery, and to truncate the log. For both uses there is a different *best moment* for making snapshot.

Algorithm 1 Catch-up Algorithm

```
1: repeat
2:    Choosing target replica
3:    Creating list of missing instances
4:    Send a CatchUpQuery
5:    Wait for timeout or for response
6: until catch-up succeeds

7: upon receiving CatchUpQuery query
8: if query has snapshot flag set then
9:    Get last snapshot
10:   Prepare CatchUpSnapshot message
11:   Send the message
12: else if requested instances already not in log then
13:   Send CatchUpResponse with snapshot flag
14: else
15:   Gather all decided instances
16:   Send CatchUpResponse

17: upon receiving CatchUpResponse response
18: if response has snapshot flag set then
19:   Prepare CatchUpQuery with snapshot flag
20:   Send the query
21: else
22:   Merge received log (if any)
23:   Wake up the catch-up loop (line 5)

24: upon receiving CatchUpSnapshot snapshot
25: Check if snapshot is newer than current one
26: Replace the current snapshot with snapshot
27: Truncate logs (to stop catch-up)
28: Wake up the catch-up loop (line 5)
```

For recovery, frequent snapshots improve the recovery time, as this would improve the chances of a replica obtaining a recent snapshot of the system, thereby reducing the duration of the second part of recovery, *e.g.*, executing missing commands.

For catch-up, the moment when another replica requests snapshot is the best for creating one. However some services cannot create snapshot anytime. So demanding a snapshot from the service is not acceptable. The snapshot should be created periodically to truncate the log without any additional requirements. Therefore, we assume the snapshot should be made when the cost of catch-up from the log becomes bigger than the cost of catch-up from the snapshot. Note that catch-up time also includes time of state/log transfer.

5.4.2 Replica vs service responsibility

There are two approaches to the problem who is in charge for the initiating of snapshot creation. Either the JPaxos `Replica` module issues a snapshot, or the service chooses an appropriate moment.

Embedding this functionality into JPaxos would surely provide more secure work (the service may simply not deliver the snapshots, which means forever growing log). It is easier for the `Replica` to measure the size of the messages that should be transferred in order to catch-up (or recover).

However, JPaxos does not know anything about the service. We can also assume, that the service is not aware of network conditions. Therefore choosing the proper moment (when the cost of log-based catch-up becomes more expensive than the state-based catch-up) is impossible for both. It is clearly visible, that service is better informed – it knows not only the size of requests, but also it may estimate the size of its current state and the resources that are needed to execute all commands from the log.

The latter is most significant difference: JPaxos has no idea how long the log execution from previous snapshot would take. It seems possible to estimate this: measuring the time between a request and the reply to that request might solve the problem. However, such estimate can give mistaken results, especially in a multi-process environment. If another process is consuming CPU, or the replica waits a long time for granting some resources (like an access to a file or even a printer), the estimate surely will not reflect the real value.

Below we present a table showing in compact way the state of knowledge needed to select the best moment for a next snapshot:

	Service	JPaxos `Replica`
Size of requests	known	known
Size of state	known	estimate
Log execution time	good estimate	poor estimate
Time for sending message	unknown	estimate

Table 4: Comparing knowledge of the service and the `Replica`

5.4.3 Snapshotting in JPaxos

Snapshotting, as described above, may be done in a variety of ways. Moreover, how often the snapshots are made depends on the implementation. Here we give some main clues how the snapshotting is implemented.

The decision who orders a snapshot creation has been left to the future user of the library. To achieve this, some assumptions have been taken – mainly concerning the architecture of the service.

The service must implement three methods: `askForSnapshot`, `forceSnapshot` and `updateToSnapshot`. Also it is required to implement adding and removing snapshot listeners – objects that implement the `onSnapshotMade` function. When a snapshot is made on the state machine, method `onSnapshotMade` with the snapshot as a parameter must be called on all snapshot listeners.

Replica measures the size of the log after every n-th instance, and calculates an average size of the snapshot based on the previous ones. By every log measurement, a ratio is calculated: $\frac{\text{log size}}{\text{snapshot estimate}}$. As the ratio exceeds one constant, method `askForSnapshot` is called. After another constant, `forceSnapshot` is executed.

There are several approaches possible on who decides the proper time for the snapshot:

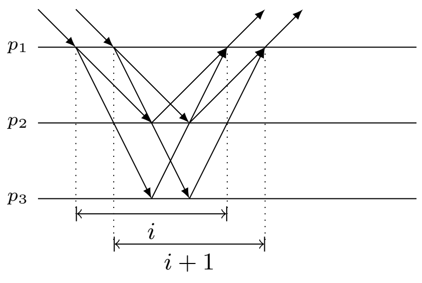
- **State machine only** - Service ignores the functions `askForSnapshot` and `forceSnapshot` and does the snapshot at its convenience,
- **Using replica calls as hints** – service takes under consideration `askForSnapshot` and `forceSnapshot` functions, but decides itself when to do snapshot,
- **Balanced responsibility** – service uses both `askForSnapshot` and `forceSnapshot`; the first as a hint, the latter treats as an order,
- **Replica only** - each time `askForSnapshot` is called, the state machine does snapshot.

Snapshotting requires also additional data exchanged between replica and service: the state machine must know the request number, to allow snapshot identification in the replica. For example, if the snapshotting would be done completely on service's side, how would the replica know after which command the snapshot was taken.

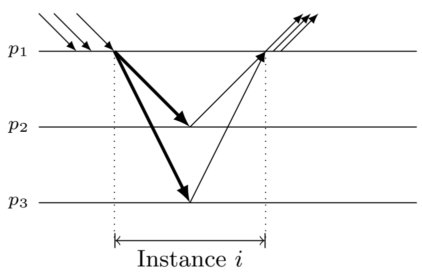
5.5 Log truncation

The replicated log cannot be allowed to grow forever, it must be bounded in any practical system. After a replica executes some command, it no longer needs the corresponding log entry locally. However, other replicas might not have learned the decision yet. In this case, these late replicas need to learn the decision by asking the corresponding log entry from a replica that still has it.

Having the old log entries is needed in three cases: for the catch-up, for view change and, in the crash-recovery model, for recovery. The catch-up needs old log entries in order to send them to a replica which is not up-to-date, once the replica requests them. During the view change, the leader requests log entries of all instances it considers undecided. By recovery, a replica needs the log of other replicas in order to know what has been decided at least since its crash.



(a) Paxos with pipelining



(b) Paxos with batching.

Figure 7: Paxos optimizations.

Especially the latter is problematic. The view change always requires a bounded number of instances in the log. However, if we assume recovery with minimal or no stable storage (as in the case of the epoch-based or view-based recovery algorithm), after the recovery a replica needs to recover the state from scratch. For this, it needs the whole log or complete state.

Global Commit Point A command can be deleted from the log after being executed by all replicas in crash-stop and crash-recovery models with stable storage. The highest instance executed by all replicas is called a *global commit point.*

However, removing all instances prior to the global commit point is not enough to ensure that the log is kept bounded, because a correct replica may be disconnected from the system for some time. When communication is reestablished, the replica needs to learn about all the decisions taken in the meantime. If one replica crashes, the other replicas have no way of knowing that it is a crash and not a disconnection, so they would have to keep the logs forever. Therefore, JPaxos relies solely on snapshots for log-truncation.

6 Paxos optimizations: Batching and pipelining

Batching and pipelining are two optimizations that can be applied to MultiPaxos to greatly improve performance in many common situations. This section describes the JPaxos implementation of these two optimizations and summarizes the results of an experimental study evaluating the gains offered by these optimizations. Further details of this study, including an analytical analysis, can be found in [12].

6.1 Pipelining

Pipelining is an optimization originally described in [1] that allows the leader to execute several instances in parallel. It is based on the observation that when a leader receives a new request, it can start a new instance at once, even if there are other that are still undecided, as shown in Figure 7a.

Executing parallel instances improves the utilization of resources by pipelining the different instances. This optimization is especially effective in high-latency networks, as the leader might have to wait a long time to receive the Phase 2b messages.

JPaxos limits the number of instances that can be executed in parallel by a configuration parameter named *window size.* This is necessary because setting the window size to an excessive value can degrade performance. On the one hand, each instance requires additional resources from the system. If too many instances are started in parallel, they may overload the system, either by maxing out the leader's CPU or by causing network congestion, resulting in a more or less severe performance degradation. A large window size will also tend to increase the time required for view change, as the new leader has to learn the state of all instances that were started but not decided. Finally, although instances can be decided out-of-order, requests must be executed in order which means that an undecided instance will delay the execution of all the requests ordered in later instances, potentially stalling the service until the gaps are filled. Although having some requests prepared ahead of time for execution may improve performance as it allows execution to proceed quickly after the gaps are filled, a large window size may result in the system spending resources ordering requests that cannot be executed instead of deciding the instances that are next in execution order.

The optimal value for the window size depends on many factors, including the network latency, the size of the requests, the speed of the replicas, and the expected workload. In [12] we have explored this problem, with an analytical analysis and an experimental evaluation of parallel instances and batching in the context of Paxos.

A non-obvious consequence of pipelining is that the same request may be ordered twice.

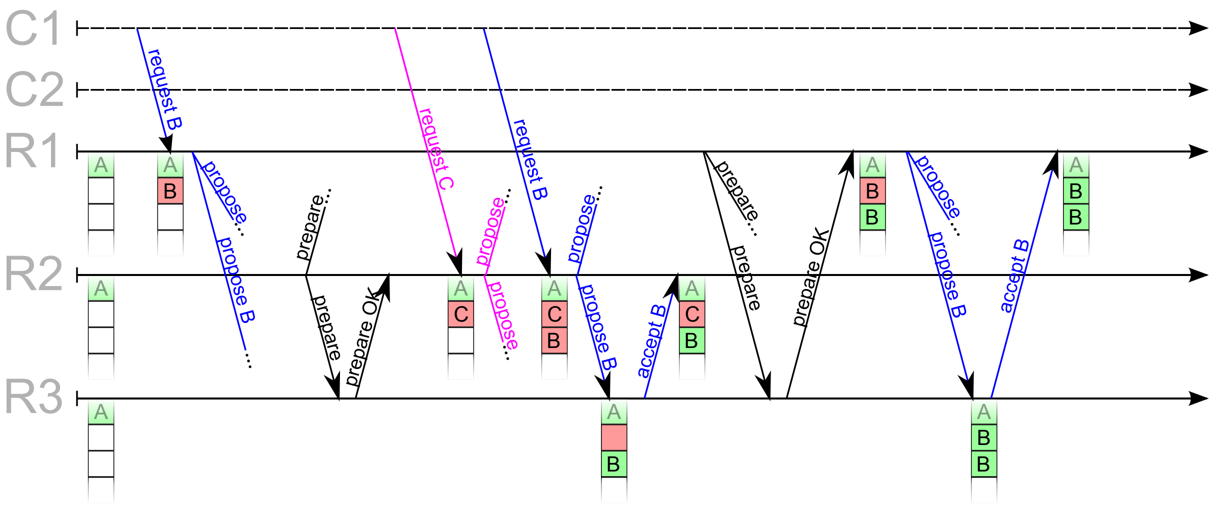


Figure 8: Duplicating messages – the example scenario

Consider the following scenario, illustrated in Figure 8. There are three replicas - R_1, R_2 and R_3. R_1 is the leader, proposes <1:B> (value B in first consensus instance). No other processes receives these messages and R_2 becomes the leader. R_2 proposes <1:C> and <2:B>. All proposes for <1:C> are lost, but <2:B> is decided. R_2 fails and R_1 becomes the leader again. While preparing, it learns about <2:B>. It has <1:B> as a previously accepted value. The Paxos algorithm requires R_1 to propose <1:B> again, since its the accepted value with the highest timestamp. In addition, this will result in the request being decided twice.

The scenario above is fairly unlikely because of the mechanism used to deal with lost replies. Recall that a replica keeps the answer to the last request executed from each client, so that it can retransmit it to the client. Therefore, if the retransmitted request arrives after the replica has executed the request, the replica will know it is a duplicate and will not try to order it.

For the rare cases where the request is ordered multiple times, JPaxos repeats the check for duplications just before executing the request. This is done by comparing the sequence id of the request with the one of the last request executed from that client, and executing the request only if the sequence id of the new request is higher.

6.2 Batching

Batching is a common optimization in communication systems, which generally provides large gains in throughput and, indirectly, in response time [19]. It can also be applied to Paxos, as illustrated by Figure 7b. Instead of proposing one request per instance, the leader packs several requests in a single instance. Once the order of a batch is established, the order of the individual requests is decided by a deterministic rule applied to the request identifiers. Packing and unpacking of requests is further illustrated by Figure 9.

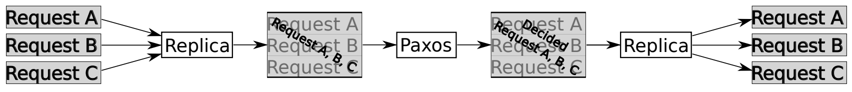


Figure 9: The batching mechanism

The gains of batching come from spreading the fixed costs of an instance over several requests, thereby decreasing the average per-request overhead. For each instance, the system performs several tasks that take a constant time regardless of the size of the proposal, or whose time increases only residually as the size of the proposal increases. These include interrupt handling and context switching as a result of reading and writing data to the network card, allocating buffers, updating the replicated log and the internal data structures, and executing the protocol logic. In [20], the authors show that the fixed costs of sending a packet over a Ethernet network are dominant for small packet sizes, and that for larger packets the total processing time grows significantly slower than the packet size. In the case of Paxos, the fixed costs of an instance are an even larger fraction of the total costs because, in addition to processing individual messages, processes also have to execute the ordering algorithm. Additionally, batching decreases dramatically the cost of using stable storage, because a single stable storage access is enough to log the state of all requests in a batch.

Although batching has a trivial implementation, ensuring that it performs well across a wide-range of workloads is challenging. The main difficulty is choosing a policy to decide when to stop waiting for more client requests and propose a batch in a new instance. This policy is critical to ensure a good performance in terms of response time observed by the clients and in throughput. In general, the larger the batches, the bigger the gains in throughput. But in practice, there are several reasons to limit the size of a batch. First, the system may have physical limits on the maximum packet size (for instance, the maximum UDP packet size is 64KB) or may perform better if UDP packets are kept under other physical limits like the maximum Ethernet frame size. Second, larger batches take longer to build because the leader has to wait for more requests, possibly delaying the ones that are already waiting and increasing the average time to order each request. This is especially problematic with low load, as it may take a long time to form a large batch. Finally, a large value takes longer to transfer and process, further increasing the latency. Therefore, a batching policy must strike a balance between creating large batches (to improve throughput) and deciding when to stop waiting for additional requests and send the batch (to keep latency within acceptable bounds). This problem has been studied in the general context of communication protocols by [21, 20, 22].

JPaxos has a Batcher module that receives the requests from clients, forms batches according to a given policy, and then passes them to the Proposer to start a new instance. This modules uses a policy that starts batches using a combination of a size and time-based trigger: a batch is started either if it is large enough or if it has been waiting for long enough, according to two configurable parameters. This policy is described below, together with the pipelining implementation as the two are intimately related.

6.3 Batching and pipelining algorithm

In JPaxos, the Batcher module controls both the batching and pipelining optimizations, *i.e.*, it decides when to create a new batch and initiate a new instance. The algorithm used by this module (Algorithm 2) is based on the following three parameters: WND, BSZ and Δ_B. The parameter WND is the maximum number of instances that can be executed in parallel (maximum window size), BSZ is the maximum batch size (in bytes), and Δ_B is the batch timeout.

The algorithm starts by waiting until there are free slots in the window size, that is, the current number of instances executing is smaller than WND.

It then tries to form a new batch, by retrieving values from the request queue, which holds the requests received from the client that are waiting for. The leader then waits until either it has enough requests to fill the batch or the age of the batch is greater than Δ_B, with age of a batch being the largest waiting time in the system among all requests in the batch.

6.4 Discussion

In [12], we present the results of an experimental study evaluating batching and pipelining. Here we summarize the conclusions of that study.

Our experiments show clearly that batching by itself provides the largest gains both in high and low latency networks. Since it is fairly simple to implement, it should be one of the first optimizations considered in Paxos and, more generally, in any implementation of a replicated state machine.

Pipelining is useful only in some systems, as its potential for throughput gains depends on the ratio between the speed of the nodes and the network latency: the more time the leader spends idle waiting

Algorithm 2 Algorithm used to batch requests and start consensus instances. $size(Batch) = \sum_{r' \in Batch} |r'|$, $age(Batch) = \min\{r'.ts : r' \in Batch\}$ where $r.ts$ is the local time when the request was received from the client, and $C_t()$ returns the local time.

```
Initialization:
  $reqQueue$ {queue with client requests waiting to be ordered}
  $w \leftarrow 0$ {number of active instances}
  fork main() task

upon instance decided
  $w \leftarrow w - 1$

procedure buildBatch()
  $Batch \leftarrow \emptyset$
  while true do
    wait until $reqQueue$ not empty or $age(Batch) + \varDelta_B \geq C_t()$
    while $reqQueue$ not empty do
      $r \leftarrow reqQueue$.first()
      if $size(Batch) + |r| \leq BSZ$ then
        $Batch \leftarrow Batch \cup \{r\}$
        $reqQueue$.removeFirst()
      else
        return $Batch$
    if $size(Batch) \geq BSZ$ or $age(Batch) + \varDelta_B \geq C_t()$ then
      return $Batch$

procedure main()
  while true do
    wait as long as $w = WND$
    $Batch \leftarrow$ buildBatch()
    start new instance with $Batch$
    $w \leftarrow w + 1$
```

for messages from other replicas, the greater the potential for gains of executing instances in parallel. Thus, in general, it will provide minimal performance gains over batching alone in low latency networks, but it provides substantial gains when latency is high.

While batching decreases the CPU overhead of the replication stack, executing parallel instances has the opposite effect because of the overhead associated with switching between many small tasks. This reduces the CPU time available for the service running on top of the replication task and, in the worst case, can lead to a performance collapse if too many instances are started simultaneously (see Emulab experiments). This problem can be avoided by carefully setting the limit on the number of parallel instances, taking in consideration the available CPU time on the leader.

The analytical model in [12] can be used to chose the parameters for these two optimizations.

7 Recovery

In this section, we describe the issues concerning recovery from crash. We start with short overview of recovery and then present four different algorithms – crash-stop, crash-recovery with stable storage, epoch-based recovery and view-based recovery. After that, we summarize and compare performance and fault tolerance of the algorithms.

Descriptions and proofs of algorithms presented in this chapter can also be found in [23]. The pseudo code provided in this chapter is a slightly modified version of pseudo code from [23]. The modifications reflect the actual implementation of the algorithms in the JPaxos library.

7.1 Overview

When reasoning about the crash-recovery, it is usual to assume that processes have access to volatile and stable storage. Any data stored in volatile memory is lost during a crash, while data on stable storage is preserved.

In practice, an hard disk is the typical media used to store data. An hard drive can be used as a stable storage media by using a write protocol that ensures that data will be consistent (based on write-ahead logging) and by using synchronous writes. Asynchronous writes provide a semantics of volatile storage, since due to buffering at the operating system level, the data may not have been written to the physical media by the time the process regains control after executing the write system call. A synchronous write is orders of magnitude slower than an asynchronous write, therefore it is important to minimize the number of stable storage writes on the critical path.

In this section, we use n to denote the number of processes, f for the number of faulty processes and p for the ID of local process.

7.2 Crash stop

In the crash-stop model, crashed process will never be up again. Because of that, there is no recovery phase and we do not need stable storage. Because processes do not perform any synchronous writes to stable storage, the protocol can execute at the maximum speed allowed by the CPU and the network, which is usually one order of magnitude higher than when stable storage is used.

However, in this model, if a majority of processes crashes, the algorithm stops, which makes this model unpractical for long lived systems. Below we present three algorithms, which allow for process recovery.

7.3 Recovery phase

A recovery phase is the state of JPaxos replica after starting the process. It must be decided, if this is the first run of this replica or recovery from crash.

Of course, if this is the first start, the process can just go to normal state. Otherwise, the process must stay in the Recovery phase as long as it does not fulfill requirements of the recovery model.

The process cannot join the Paxos protocol until the recovery phase is finished. So recovering process cannot respond to any message received, like Propose, Accept, Prepare and PrepareOK. However, these messages may be received and processed, so the process can passively join the protocol. The moment when the process joins (actively) Paxos protocol is also a moment when the process is considered as correct.

7.4 Crash-recovery with stable storage

First algorithm with ability to recover a process is crash recovery with stable storage. In this algorithm, a process saves enough information to stable storage so that upon recovery, it can restore its state using only information stored on local storage, and rejoin the protocol without executing any additional recovery protocol involving the other replicas. With this algorithm, processes write to stable storage often, once per instance of the ordering protocol, which results in a significant overhead.

The following are the variables used in the algorithm.

log_p – the array of consecutive instances – $id \rightarrow < view, value >$
$view_p$ – the current view number
$state_p$ – the last snapshot made by service or received from catch-up

On recovery, the algorithm reads the view, values for all consensus instances and last snapshot from stable storage and can join a Paxos protocol (the process cannot respond to any message until it loads everything from stable storage).

This synchronous writes are made on critical path, so the performance of this algorithm will be much lower comparing to crash-stop model. However, the crash recovery with stable storage algorithm tolerates catastrophic failures – the failure when all processes crashed. Of course, when the majority of the processes are crashed, the replicated service is unavailable. However, we can start all processes again and they will recover to a state before the crash.

As we can see, this algorithm is already deployable in environment with crashes. However, because the performance is low, we describe two more algorithms. These algorithms will not tolerate catastrophic failures but their performance should be close to the performance of crash-stop model.

Algorithm 3 Crash-recovery with stable storage [23].

```
Initialization:
  $log_p \leftarrow \bot$
  $view_p \leftarrow 0$
  if $view_p$ mod $n = p$ then
    $view_p \leftarrow 1$
  $state_p \leftarrow \bot$

  if recovery after crash then
    $view_p \leftarrow$ last view number written to stable storage
    for $< id, view, value >$ from stable storage do
      $log_p[id] \leftarrow < view, value >$
    $state_p \leftarrow$ last snapshot saved to stable storage

  join Paxos protocol

procedure advanceView(view)
  write $view$ to stable storage
  $view_p \leftarrow view$

procedure updateValue(id, view, value)
  $log_p[id] \leftarrow < view, value >$
  write $< id, view, value >$ to stable storage

procedure newSnapshot(snapshot)
  write $snapshot$ to stable storage
  $state_p \leftarrow snapshot$
```

7.5 Epoch-based recovery

In this section, we describe the algorithm based on epoch numbers. This algorithm makes only one synchronous write to stable storage on process startup, but the recovery phase is more complicated compared to the previous algorithm.

We will use following variables in the pseudocode:

$epoch_p$	– vector of epoch numbers
e	– epoch number of single process
$highestId$	– the id of highest known consensus instance

Algorithm 4 presents the pseudo-code.

Epoch-based recovery algorithms tolerates $f = \left\lfloor \frac{n-1}{2} \right\rfloor$ crashed processes. It requires that at least a majority of up processes at any time - otherwise the algorithm will block forever.

In the recovery phase the process first notifies the majority of other replicas about its recovery. As the process must know when its state becomes correct, it must wait for recovery answer from the leader, who has the most recent state. Once it gets the acknowledgments, it is downloading all information from other replicas using the catch-up mechanism. When all necessary information is transferred, the replica can join the Paxos protocol and is considered as correct.

On normal execution, this algorithm should be as fast as the algorithm in the crash-stop model. The size of PrepareOK message is increased by epoch vector, which contains n numbers but it should not have big impact on performance or network usage. This algorithm is a compromise between crash-stop and crash-recovery with stable storage.

Example

The following example illustrates the algorithm (See Figure 10). Let 5 processes, numbered from 0 to 4, which are all up and in view 4 (where process 4 is a leader). The epoch vectors are initially set to $[0, 0, 0, 0, 0]$.

Algorithm 4 Epoch-based recovery [23].

```
Initialization:
  $log_p \leftarrow \perp$
  $view_p \leftarrow 0$
  if $view_p \bmod n = p$ then
    $view_p \leftarrow view_p + 1$
  $state_p \leftarrow \perp$
  $\forall q : epoch_p[q] \leftarrow 0$

  if recovery after crash then
    $epoch_p[p] \leftarrow$ last epoch number written to stable storage
    $epoch_p[p] \leftarrow epoch_p[p] + 1$
    send $<$ RECOVERY $, epoch_p[p] >$ to all except $p$
    wait for $n - f$ $<$ RECOVERYANSWER $, epoch, view, highestId >$ messages including from primary of highest view received
    $\forall s \in \Pi : epoch_p[s] \leftarrow \max\{epoch[s]$ received$\}$
    $view_p \leftarrow \max\{view$ received$\}$
    download all instance up to $highestId$ from leader using catch-up module
  write $epoch_p[p]$ to stable storage

  join Paxos protocol

upon receive $<$ RECOVERY $, e >$ from $q$
  if $q$ is primary of $view_p$ then
    change to a higher view where $q$ is not primary
    wait until view change is complete
  $epoch_p[q] \leftarrow e$
  send $<$ RECOVERYANSWER $, epoch_p, view_p, highestId_p >$ to $q$

upon $<$ PREPAREOK $, epoch, view, highestId >$ from $q$
  if $view > view_p$ then
    $view_p \leftarrow view$
    send $<$ PREPARE $, view_p >$ to all
  else
    $\forall q : epoch_p[q] \leftarrow \max\{epoch[q], epoch_q[q]\}$
    for all $s \in \Pi$ do
      discard all messages $<$ PREPAREOK $, epoch, view_p, -, - >$ from $s$ where $epoch[s] < epoch_q[p]$
    execute normal handler
```

Process 0 suspects the leader and advances to view 5 by sending PREPARE message to all. Process 1 receives PREPARE and respond with $\langle \text{PREPAREOK}, [0,0,0,0,0], 5, - \rangle$. The process 0 receives the response and process 1 crashes. After the crash process 1 recovers, increases epoch number to 1 and sends $\langle \text{RECOVERY}, 1 \rangle$ message to all. Processes [2, 3, 4] receive RECOVERY message, update epoch vector to [0, 1, 0, 0, 0] and responds with RECOVERYANSWER and view 4. After receiving three RECOVERYANSWER messages with view 4 (also from leader of view 4), process 1 joins the protocol with view 4. Now process 2 receives the PREPARE from process 0 and responds with $\langle \text{PREPAREOK}, [0,1,0,0,0], 5, - \rangle$.

Process 0 already received three PREPAREOK responses, from [0, 1, 2]. However, message from process 1 will be discarded because it was sent when process 1 was in epoch 0 and now from process 2 we know that process 1 is in epoch 1. If we would not discard this message, process 0 would be prepared leader in view 5 but processes [1, 3, 4] would be in view 4 and that would cause violation of Paxos protocol.

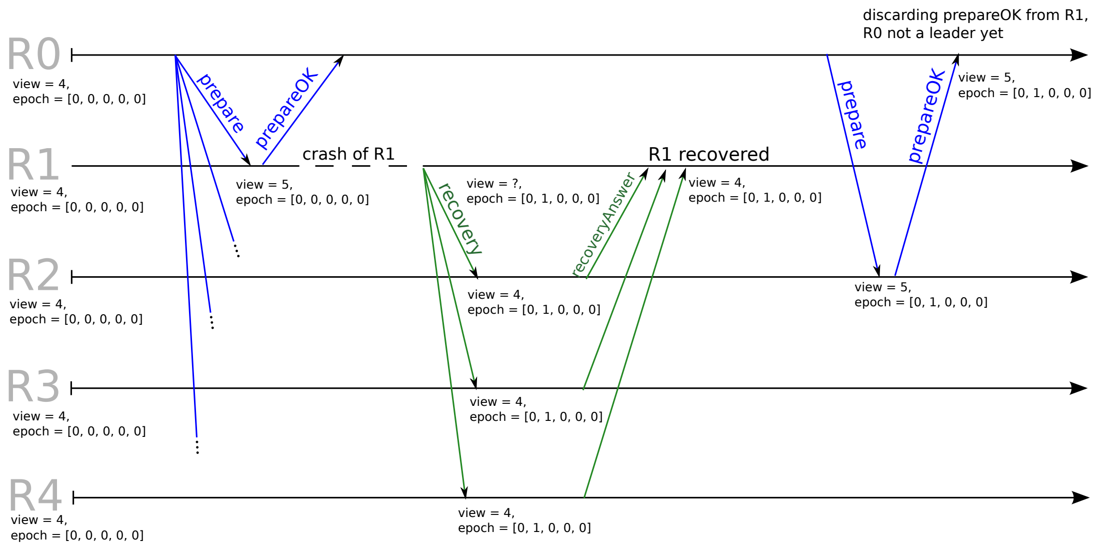


Figure 10: Example of epoch-based recovery.

7.6 View-based recovery

The next algorithm is view-based recovery. Like epoch-based recovery, it requires a majority of up processes at any given moment, but instead of using epoch numbers, the view number is written to stable storage on every change. The pseudocode of this algorithm is presented in Algorithm 5.

This algorithm writes to stable storage on every view change. If the network is stable and the leader does not crash, performance of the algorithm should be the same as in crash-stop model. If the view is changed often because of message loss or leader crash, the performance can decrease.

The recovery phase is quite similar to the epoch-based recovery. The recovering process sends RECOVERY message to all, waits for a majority of RECOVERYANSWER , including one from the leader from which it obtains the *highestId* value. Then catch-up mechanism is used to retrieve missing instances from correct processes. This algorithm is easier than based on epoch, because it does not introduce any changes in handling the PREPAREOK message.

7.7 Comparison

We presented four different algorithms implemented in JPaxos library. Each algorithm has different performance and fault tolerance. To choose the best algorithm for user service, we need to decide what fault tolerance and performance is required. Table 5, compares the four algorithms.

Crash recovery with full stable storage is an algorithm with the best fault tolerance because it tolerates catastrophic failures. However, to achieve that it makes a synchronous write to stable storage on every round (on every view and value change) what decreases the performance.

Algorithm 5 View-based recovery [23].

```
Initialization:
    $log_p \leftarrow \perp$
    $view_p \leftarrow 0$
    if $view_p$ mod $n = p$ then
        $view_p \leftarrow view_p + 1$
    $state_p \leftarrow \perp$

    if recovery after crash then
        $view_p \leftarrow$ last view number written to stable storage
        if $view_p$ mod $n = p$ then
            $view_p \leftarrow view_p + 1$
        send $<$ RECOVERY, $view_p >$ to all except $p$
        wait for $n - f <$ RECOVERYANSWER, $view, highestId >$ messages including from primary of highest view received
        $view_p \leftarrow \max\{view$ received$\}$
        download all instance up to $highestId$ from leader using catch-up module

    join Paxos protocol

procedure advanceView($view$)
    write $view$ to stable storage
    $view_p \leftarrow view$

upon receive $<$ RECOVERY, $view >$ from $q$
    if $q$ is primary of $view_p$ then
        change to a higher view where $q$ is not primary
        wait until view change is complete
    if $view > view_p$ then
        advanceView(view)
    send $<$ RECOVERYANSWER, $view_p, highestId_p >$ to $q$
```

	Crash-stop	Full stable storage	View-based	Epoch-based
Fault tolerance	$2f+1$	f	$2f+1$	$2f+1$
Process can recover	no	yes	yes	yes
Stable storage use	no	per round	per view change	on recovery
Additional messages	—	—	RECOVERY	RECOVERY
State transfer	no	if needed	yes	yes
Increased msg size	—	—	—	PREPAREOK by n

Table 5: Comparison of recovery algorithms [23].

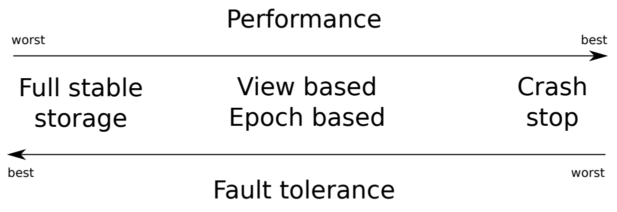


Figure 11: Comparison of recovery algorithms.

Crash stop does not provide any recovery (crashed processes will never be up again) and a majority of processes must be correct. However, no synchronous writes to stable storage are made so it provides the best performance.

The compromise between crash recovery with stable storage and crash stop are view-based and epoch-based recovery. They support recovery of crashed process and their performance should be close to crash-stop model. If the service does not have to tolerate catastrophic failures, it is the best to use them in production environment.

8 Threading and Scalability with multi-core

One of the main goals of JPaxos is to be scalable with multi-core CPUs. There are two main difficulties in achieving this goal: 1) finding and exploiting opportunities for parallel execution and 2) ensuring thread safety.

Parallelism and thread safety are often in conflict, as extracting parallelism requires the use of multiple threads which, in turn, introduces the danger of race conditions. These are particularly problematic in an implementation of state-machine replication, which has a complex state that is accessed and manipulated in response to a variety of external and internal events that for the most part happen concurrently.

It is possible to side-step all race conditions with a single-thread design, using non-blocking I/O and an event-driven architecture. Because race-conditions cannot occur in the presence of a single thread, this is a trivially thread-safe design. Additionally, it may be more efficient than a threaded design, as it eliminates all the overhead from managing threads. The main drawback is that this design does not take advantage of multi-core CPUs, which – given the current trend to an increasing number of cores – is a severe limitation. A second concern is the complexity of the design, which can be significantly higher in a purely event-driven model. Some modules have simple designs when implemented as a thread, but are complex to express in an event-driven model (*e.g.*, failure detection).

JPaxos uses a hybrid design, with a mixture of event-driven and thread-based modules. This design is loosely based on the concept of Actors in languages like Erlang and Scala, where an Actor encapsulates both state and threading. Each module consists of private state, one or more threads, and a well-defined interface for communicating with threads from other modules (usually through message queues). The private state is accessible only by the threads managed by the module. This organization keeps complex state isolated in a module, with well-defined access points, making it easier to reason about thread safety. Some modules may contain a single thread, which automatically provides thread-safety inside the module. Other modules may contain multiple threads, but as the shared state is confined to the module, it is usually easier to enforce thread safety (either through locking or by state partitioning among threads).

The design for each module is chosen based on several factors, like the potential for parallelism, the complexity of its state and the complexity of its implementation. In the modules with the greatest

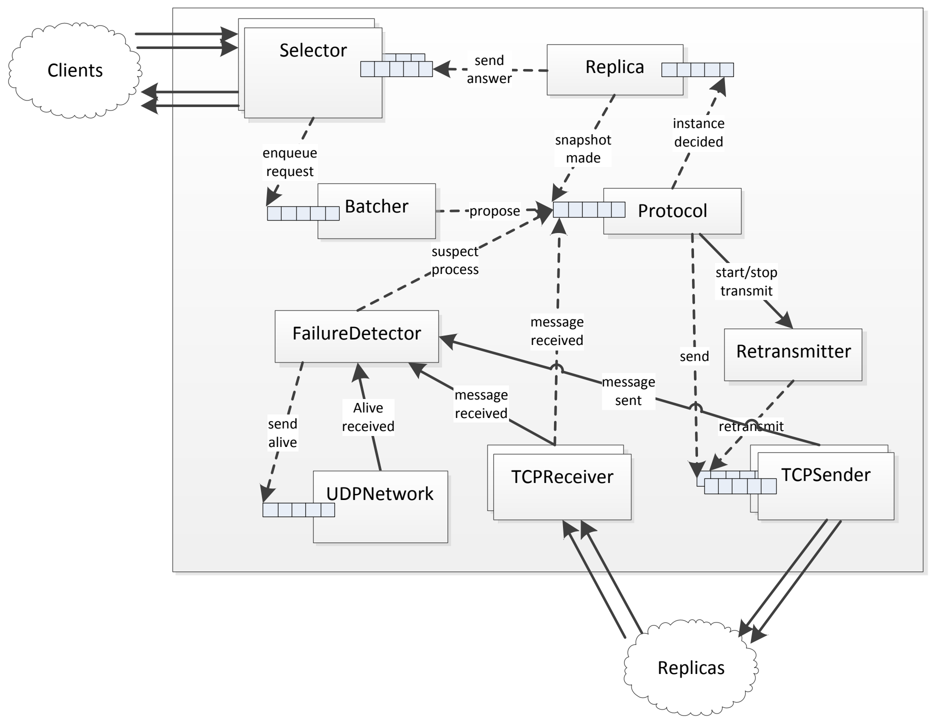


Figure 12: Threading architecture of JPaxos. Solid lines represent synchronous calls and dashed lines represent asynchronous calls.

potential for parallelism, we used several threads to extract this parallelism. This is the case of receiving/sending messages from/to the clients and the replicas, which corresponds to the bulk of the CPU usage. When the module has little or no potential for parallelism, we chose the design with the simplest implementation, in terms both of code complexity and thread safety. For instance some tasks, like retransmission and failure detection, have simpler implementations using a dedicated thread and blocking operations rather than using an event-driven model. Other tasks, like executing the core MultiPaxos protocol, have more natural implementations using an event-driven architecture, as they consist mainly of event handlers.

Below we describe the threading design of JPaxos. The results of an experimental evaluation of the scalability in multicore systems can be found in [24].

8.1 Threading architecture

Figure 12 shows the threading architecture of JPaxos.

Each module contains one or more threads. A module with a single box contains only one thread, while those with multiple overlapping boxes contain several. The arrows represent the type of interaction between the modules. A solid arrow represents a synchronous call or communication with an external process, while a dashed arrow represents an asynchronous call (putting a message on the message queue of the module).

Next we describe each module.

Selector This module handles all connections with the clients, being responsible for receiving new connection requests, receiving requests and sending answers. It uses non-blocking I/O (Java NIO) in order to scale efficiently to several thousands of concurrent client connections. The profiling tests have shown that reading and writing requests represent a significant fraction of the CPU utilization of JPaxos, so we have parallelized this module using state partitioning among threads. New connections are assigned to one of multiple Selector threads using a round-robin strategy; these threads will then be responsible to handle the connection during its lifetime. The number of threads is configurable.

Batcher This thread manages a queue with incoming requests, packs them into proposals according to the batching policy and feeds them to the Protocol thread once this thread is ready to send additional proposals. The Selector thread places new request in the Batcher queue, where the requests will remain until there is space for them in batch. This queue is also a key part of flow control in JPaxos. If the system is overloaded, this queue eventually fills up, which in turn blocks the Selector threads when they try to add new requests. As these threads stop reading requests from the clients, the TCP buffers become full which in turn stops the clients from sending more requests.

Protocol This thread is responsible for executing the core replication protocol and has exclusive access to the protocol state, *i.e.*, replicated log and associated variables, as well as the view number. It is implemented as a single-thread event loop with an associated queue for incoming events. The main events handled by the protocol thread are reception of new protocol message, batch ready to be proposed, leader suspected, and snapshot performed. It is also responsible to manage catch-up internally. The complexity of the state and of the operations performed by this module were the main reason to make it single-threaded. Additionally, there would be little to gain in making this module multi-threaded, as most operations execute quickly.

FailureDetector JPaxos uses a dedicated thread for failure detection. This is a very natural design for this type of task. Additionally, as compared to an event-driven architecture, it provides more precise timing of suspicions or sending heartbeats, as the scheduling is handled directly by the operating system. This thread simply alternates between sending heartbeats or suspecting the leader, in both cases blocking until it is time to send an heartbeat or to suspect a leader. Since protocol messages are also used as heartbeats, this module receives callbacks whenever a message is sent or received. If the message comes from the leader or is sent by the leader to all, then the corresponding timestamps are updated. To minimize context switches, this is done by the callback thread without waking up the failure detector thread. This is safe, because updating these fields will always result in delaying either the sending of the heartbeat or the suspicion of the leader, so we can let the failure detector thread wait for the original delay and then recompute what to do.

Retransmitter The retransmitter is responsible to ensure that messages essential to the progress of the protocol are eventually delivered. When the Protocol thread sends a message for the first time, it also hands it over to the Retransmitter thread which will start retransmitting it periodically. As the protocol advances, the Protocol thread cancels the retransmission of messages that no longer need to be delivered. This module consists of a thread and a blocking queue containing the messages to be retransmitted sorted by next time of retransmission.

TCPSender and TCPReceiver When using TCP to communicate between the replicas, each replica keeps a connection to each other replica. For each connection, JPaxos uses one thread for receiving messages and another to send. The TCPReceiver threads read and deserialize the request and pass it to the Protocol thread. With this design, deserialization of messages from different replicas can be done in parallel, without slowing down the Protocol thread, which increases the potential parallelism of JPaxos.

Each TCPSender thread keeps a queue with messages to be sent, writing them to the socket as fast as the socket can accept them. The purpose of this thread is to prevent the Protocol thread from blocking. This can easily occur when sending bursts of messages faster than the TCP stack can send them, which will result in filling up the TCP buffers of the operating system thereby blocking the thread writing to the socket. If the protocol thread is allowed to block in these situations, then any problem in communicating with a single replica could stop the whole system. We experienced this last situation in a early prototype that used such design: when the TCP connection to one replica stopped accepting more data, the protocol thread would block and stop processing messages from other replicas. This problem continued until the TCP connection finally failed, which can take up to a few minutes. By having a dedicated thread sending to every replica, then in the worst case only this thread is blocked. If the queue of a TCPSender becomes full, then the connection is closed.

UDPNetwork This thread receives and deserializes all the messages received using UDP, and forwards them to the appropriate module. There is always a single UDP socket for each replica, which is used for the heartbeats of the failure detector and, optionally, for inter-replica communication. Contrary to when using TCP for inter-replica communication, there is no dedicated UDP sender thread. This is because a UDP send will never block, so there is no risk of blocking the whole system when sending through UDP.

Replica After a batch is ordered by the Protocol thread, it is handed over to the Replica thread incoming queue. This thread will then read the batches from the queue, unpack them and execute them sequentially, and pass the answers to the Selector thread for sending to the clients.

9 Performance Evaluation

This section presents a summary of the results of a performance evaluation of JPaxos, focusing on the effect of batching and pipelining on the throughput of JPaxos.

In all experiments we assume the crash-stop model, and either 3 or 5 replicas.

The replicated service keeps no state. It receives requests containing an array of S_{req} bytes and answers with an 8 bytes array. We chose a simple service as this puts the most stress on the replication mechanisms. In order to show the limits of JPaxos, all the experiments were performed with the system under high load.

All communication is done over TCP. We did not use IP multicast because it is not generally available in WAN-like topologies. Initially we considered UDP, but rejected it because in our tests it did not provide any performance advantage over TCP. TCP has the advantage of providing flow and congestion control, and of having no limits on message size, therefore saving us the tedious work of reimplementing these features. The replicas open the connections at startup and keep them open until the end of the run. Each data point in the plots corresponds to a 3 minutes run, excluding the first 10%. For clarity, the plots below do not include error bars for the 95% confidence interval, as the errors are usually very small.

9.1 Batching versus pipelining

The experiments below examine the impact of batching and pipelining on the performance of JPaxos. The full results can be found in [12].

We used 900 clients in the cluster and 1200 in Emulab, which is enough for the leader to form new batches without having to wait for additional requests. All experiments were run using a single-core, by using the `taskset` Linux utility to instruct the OS scheduler to run the process only on a single core. This was done to better isolate the parameters in study.

For each experiment, we show six metrics: client latency, instance latency, request throughput, instance throughput, average batch size and average number of parallel instances. The *client latency* is the time the client waits for the reply to one request, which includes the transmission time from the client to the leader, the queuing time of the request at the leader, the time to order the request, and the time to send the answer back to the client.

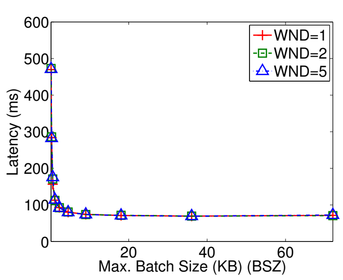
The *latency per instance* is the time elapsed at the leader from proposal to decision of an instance, *i.e.*, from sending the Phase 2a message to receiving a majority of Phase 2b messages.

The *throughput of instances* is the number of Phase 2 executed per second, and the *throughput of requests* is the number of requests ordered per second. Note that the throughput of requests is equal to the throughput of instances multiplied by the average number of requests per instance.

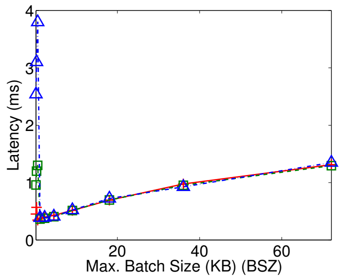
The *average batch size* (*bsz*) and *average window size* (*w*) show how well the system is taking advantage of the optimizations. As mentioned previously, the leader might not always fill the batches completely (*i.e.*, up to *BSZ*) or to execute the maximum number of parallel instances. This can happen either because of the lack of sufficient client requests queued for ordering or because the leader is not fast enough to execute *WND* parallel instances simultaneously (previous instances finish before the leader is able to start additional ones). Therefore, we measured the average size of the batches and the average number of parallel instances and show the results below, in order to evaluate the effectiveness of the optimizations in each scenario.

9.1.1 Cluster

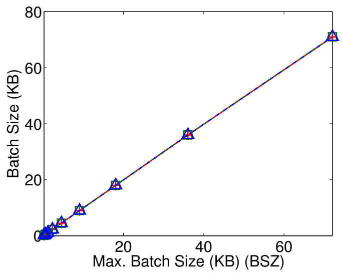
The following experiments were run on a cluster of Pentium 4 at 3GHz with 1GB memory connected by a Gigabit Ethernet. The effective bandwidth of a TCP stream between two nodes measured by `netperf`



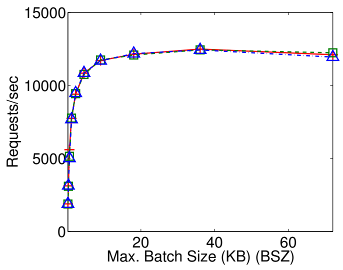
(a) Client latency



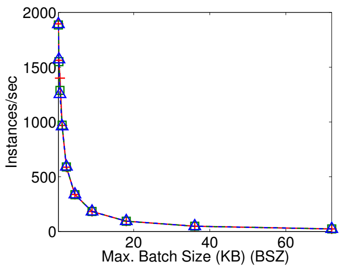
(b) Instance latency



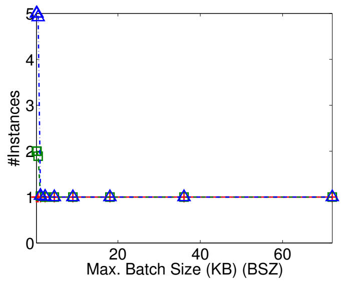
(c) Avg batch size (bsz)



(d) Requests/sec



(e) Instances/sec



(f) Avg window size (w)

Figure 13: Cluster, experimental results with $S_{req} = 128$.

is 940 Mbit/s.

Figure 13 shows the results as a function of batch size, for request sizes of 128 bytes, and maximum window sizes of 1, 2 and 5.

The throughput reaches a maximum of 12K requests for batch sizes of 20KB and higher (Fig 13d). At this point, the bottleneck is the CPU of the leader. The network shows an utilization of around 24Mbits/s, which is far from the limit of 940Mbits/s. The response time as seen by the client is under 100ms (Fig 13a), while the latency of each instance is below 1ms, increasing slightly as the batch size increases (Fig 13b). The reason for such a large difference between these two values is the large queuing time of client requests at the leader before being proposed as part of a batch, caused by the clients generating more load than the capacity of the system.

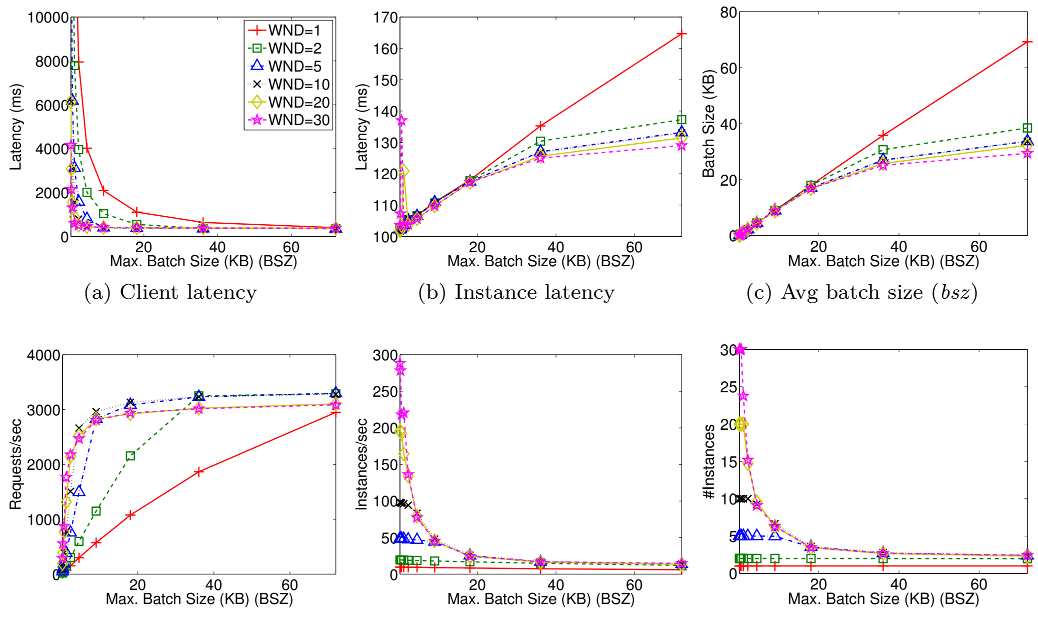
The results show the importance of batching for small requests. The throughput increases almost 10 times, until a maximum of 12K req/s with 20KB batches. Nevertheless, as *BSZ* is increased, the number of instances executed per second drops (Fig 13e). This is expected, because larger batches take longer to create and to transmit over the network. This drop in instance throughput is compensated by a larger increase in the number of requests per batch, resulting in an overall gain in request throughput.

Contrary to batching, pipelining does not improve throughput. This is evident in Figure 13f, which shows that the leader is not fast enough to start more than one instance at a time, except for the smallest batch sizes. This is because in these conditions it takes longer to form a new batch than to execute an instance. The performance does not drop if the *BSZ* or *WND* are increased past their optimal values. This is a desirable behavior, because the system will perform optimally with a wide range of configuration parameters, making it easier to tune.

9.1.2 Emulab - WAN environment

We used Emulab [25] to emulate the conditions of a typical WAN environment with geographically distributed nodes. The replicas are connected point-to-point by a 10Mbits link with 50ms of latency. To keep the system under high load the clients are connected directly to each replica and communicate at the speed of the physical network (1Gbits). The physical cluster used to run the experiments consisted of nodes of Pentium III at 850MHz with 512MB of memory, connected by a 100Mbps Ethernet.

The maximum throughput in the Emulab experiments is reached at 3K requests/sec (Fig 14d). At this level, the network is close to saturation, with an utilization of over 7Mbits. The response time is on



(a) Client latency (b) Instance latency (c) Avg batch size (bsz)

(d) Requests/sec (e) Instances/sec (f) Avg window size (w)

Figure 14: Emulab, experimental results with $S_{req} = 128$.

the order of seconds (Fig 14a), which is justified by the low capacity of the system.

Contrary to the cluster experiments, the maximum throughput is reached for a variety of combinations of the *BSZ* and *WND*. For small values of *WND*, the *BSZ* must be increases up to 70KB to achieve the maximum throughput, but as *WND* is increased, the maximum is reached with smaller and smaller values for *BSZ*. This is explained by a different bottleneck in the Emulab experiments. The CPU is mostly idle, while the limited bandwidth and high latency limit the performance. Under these conditions, executing multiple instances in parallel pays off, by filling up the waiting time between instances due to the latency. But if the batch size is large enough, then sending the batch is already enough to fully occupy the bandwidth, in which case there is no gain in executing parallel instances. This is clear in Fig 14f, which shows that for small batch sizes the system can run many parallel instances, but as the batch size decreases the number of parallel instances drops to around 3.

10 Related work

The first works describing replication protocols incorporating the algorithmic ideas of Paxos were published in the end of the 1980s. Lamport presented Paxos and MultiPaxos in [26], while in [17] Oki and Liskov presented the Viewstamped Replication protocol for state machine replication, which is based on the same ideas as MultiPaxos. For the first two decades after the initial publication, the work on Paxos was mostly theoretical, consisting mainly of analytical analysis and extensions of the basic algorithm. In [7], DePrisco describes and analyzes Paxos using timed-automata, and in [27], Lampson describes several variants of Paxos and compares them in terms of safety, liveness and performance. Among the protocols that extend Paxos, some of the most important are the following: Disk Paxos [28] uses remotely accessible disks to decrease the number of processes needed for liveness, PBFT [29] extends the algorithmic ideas of Paxos into the Byzantine model, Cheap Paxos [2] shows how to use only f+1 acceptors, Fast Paxos [3] decreases the fault-free case latency by one message delay, and Mencious [5] uses a multi-coordinator schema to decrease the average number of message delays in high-latency networks.

Although Paxos is well understood from a theoretical point of view, it was only recently that there has been a growing interests in the practical aspects of using Paxos for state machine replication. This is especially important because, as shown in this report, the abstract description of Paxos leaves open many important details that must all be correctly addressed in any practical implementation. One of the first works addressing these practical aspects was published in 2007 in [10]). The article describes several implementation issues concerning the Paxos algorithm in the Chubby distributed locking server developed at Google. A year later, [11] provides a detailed algorithmic description of an implementation of state machine replication using MultiPaxos. This work was the primary inspiration for JPaxos. Other works have focused in improving the throughput of Paxos. LCR [30] distributes the processes along a ring to maximize network utilization and RingPaxos [31] combines several optimizations, including ring topology, f+1 acceptors and usage of IP multicast.

The crash-recovery model has also been the subject of many research papers. Both of the original descriptions of Paxos ([1, 17]) mention that by logging to stable storage one message per consensus instance, Paxos can tolerate catastrophic failures. Because stable storage access adds significant overhead to the protocol, there has been some effort to reduce the usage of stable storage. In [32], Liskov describes a variant of viewstamped replication that reduces the number of stable storage accesses to one per view change.

Completely avoiding stable storage is not possible in the general case, because if all processes crash at the same time, the system looses all its state. In [14] the authors provide a tight bound on the number of processes that must be always-up for consensus to be solved. In practice, it is possible to circumvent this impossibility and design a system where an arbitrary number of processes can fail, as long as not "too many" fail at the "same time". This is because as long as a majority of processes are up and have a correct state at any given time, they can help processes that crashed and recovered in rebuilding their state. Therefore, sometime after the crash and recovery of a process, the system is repaired, allowing other processes to crash. This idea is used in [32], in another variant of viewstamped replication that tolerates an arbitrary number of crashes and recoveries over the lifetime of the system.

In our work, we improve upon these algorithms, by further reducing the amount of stable storage accesses from one per view change to one per recovery. Further details, including the description of the recovery algorithms implemented in JPaxos, can be found in [23].

11 Conclusions

In this report, we have presented JPaxos, a fully functional state-machine replication library. JPaxos has at its core the MultiPaxos algorithm described originally in [1], providing the fundamental theoretical basis of our work. But, as described in this report, this is just a small part of building a complete system; a fully functional system must tackle many engineering challenges. For instance, it must handle unreliable network with message loss and delays, so we presented the catch-up mechanism, which guarantees that eventually every correct replica will be up to date. The snapshot mechanism is used to bound the size of the replicated log and to limit the amount of data that must be copied during state transfer. To improve the efficiency of JPaxos, we implemented several performance improvements like skipping redundant messages, pipelining and batching. A companion report [12] studies in detail how to tune pipelining and batching. To tolerate crash-recovery faults, JPaxos includes four different recovery algorithms, which we have described above. We have also devoted considerable attention to the threading architecture of JPaxos, with the goals of providing a good scalability with the number of cores in a node while at the same time keeping the architecture simple enough to minimize the risk of race-conditions. For this, we adopted an hybrid architecture, combining event-driven with thread-based modules, where the decision for each module is based on its potential for parallelism, and the relative implementation complexity of the two options.

Although some of these challenges were already described in the literature, others, to our knowledge, have received little attention in the context of state machine replication, like tuning batching and pipelining, the recovery algorithms with little or no stable storage, and a scalable threading architecture.

We have released JPaxos as an open source project[3]. Additionaly, several research projects are using JPaxos, including PaxosSTM [33], where it is used as communication layer.

Our future work includes introducing further performance improvements to JPaxos and adding support for crash-recovery models without the stable storage at all. It would be also interesting to evaluate all recovery algorithms in a network with message loss and environment when failure of replica is a common scenario.

[3]`https://github.com/JPaxos/JPaxos`

Acknowledgments

We would like to thank Tadeusz Kobus and Maciej Kokociński for using and testing our library and discussions about JPaxos API.

This work has been partially supported by the Polish Ministry of Science and Higher Education within the European Regional Development Fund, Grant No. POIG.01.03.01-00-008/08. The visit of Kończak and Żurkowski at EPFL in July and August 2009 was supported by EPFL within the Summer@EPFL internship program.

References

[1] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, May 1998.

[2] L. Lamport and M. Massa, "Cheap Paxos," in *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*. Washington, DC, USA: IEEE Computer Society, 2004. [Online]. Available: http://portal.acm.org/citation.cfm?id=1009382.1009745

[3] L. Lamport, "Fast Paxos," *Distributed Computing*, vol. 19, no. 2, pp. 79–102, Oct. 2006.

[4] E. Gafni and L. Lamport, "Disk Paxos," *Distributed Computing*, vol. 16, no. 1, pp. 1–20, 2003.

[5] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: building efficient replicated state machines for wans," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 369–384. [Online]. Available: http://portal.acm.org/citation.cfm?id=1855741.1855767

[6] L. Lamport, D. Malkhi, and L. Zhou, "Vertical Paxos and primary-backup replication," in *Proceedings of the 28th ACM symposium on Principles of distributed computing*, ser. PODC '09. New York, NY, USA: ACM, 2009, pp. 312–313. [Online]. Available: http://doi.acm.org/10.1145/1582716.1582783

[7] R. De Prisco, B. Lampson, and N. Lynch, "Revisiting the Paxos algorithm," *Theoretical Computer Science*, vol. 243, no. 1–2, pp. 35–91, 2000.

[8] N. Hayashibara, P. Urbán, A. Schiper, and T. Katayama, "Performance comparison between the Paxos and Chandra-Toueg consensus algorithms," in *Proc. Int'l Arab Conference on Information Technology (ACIT 2002)*, 2002, pp. 526–533.

[9] L. Lamport, "Lower bounds for asynchronous consensus," *Distributed Computing*, vol. 19, pp. 104–125, 2006, 10.1007/s00446-006-0155-x. [Online]. Available: http://dx.doi.org/10.1007/s00446-006-0155-x

[10] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM Press, 2007, pp. 398–407. [Online]. Available: http://dx.doi.org/10.1145/1281100.1281103

[11] Y. Amir and J. Kirsch, "Paxos for system builders," Johns Hopkins University, Tech. Rep. CNDS-2008-2, 2008.

[12] N. Santos and A. Schiper, "Tuning Paxos for high-throughput with batching and pipelining," EPFL, Tech. Rep. EPFL-REPORT-165372, Jul. 2011.

[13] M. Welsh, D. Culler, and E. Brewer, "Seda: an architecture for well-conditioned, scalable internet services," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, ser. SOSP. New York, NY, USA: ACM, 2001, pp. 230–243. [Online]. Available: http://doi.acm.org/10.1145/502034.502057

[14] M. K. Aguilera, W. Chen, and S. Toueg, "Failure detection and consensus in the crash-recovery model," *Distributed Computing*, vol. 13, no. 2, pp. 99–125, Apr. 2000. [Online]. Available: http://www.cs.cornell.edu/home/sam/FDpapers/crash-recovery-finaldcversion.ps

[15] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Comput. Surv.*, vol. 36, Dec. 2004.

[16] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, Mar. 1996.

[17] B. M. Oki and B. H. Liskov, "Viewstamped replication: A new primary copy method to support highly-available distributed systems," in *PODC '88: Proceedings of the seventh annual ACM Symposium on Principles of distributed computing.* New York, NY, USA: ACM, 1988, pp. 8–17.

[18] R. Ekwall and A. Schiper, "Solving atomic broadcast with indirect consensus," in *Proceedings of the International Conference on Dependable Systems and Networks.* Washington, DC, USA: IEEE Computer Society, 2006, pp. 156–165. [Online]. Available: http://portal.acm.org/citation.cfm?id=1135532.1135696

[19] R. Friedman and R. Renesse, "Packing messages as a tool for boosting the performance of total ordering protocols," Department of Computer Science, Cornell University, Tech. Rep. TR95-1527, 1995.

[20] B. Carmeli, G. Gershinsky, A. Harpaz, N. Naaman, H. Nelken, J. Satran, and P. Vortman, "High throughput reliable message dissemination," in *Proceedings of the 2004 ACM Symposium on Applied Computing*, NY, USA, 2004.

[21] A. Bartoli, C. Calabrese, M. Prica, E. Di Muro, and A. Montresor, "Adaptive message packing for group communication systems," in *OTM 2003 Workshops*, ser. LNCS. Springer, 2003.

[22] R. Friedman and E. Hadad, "Adaptive batching for replicated servers," in *Symposium on Reliable Distributed Systems, SRDS'06*, Oct. 2006.

[23] N. Santos, "Recovery on the Paxos protocol," EPFL, Tech. Rep., May 2010.

[24] N. Santos and A. Schiper, "Scaling Paxos to multicore systems," EPFL, Tech. Rep. to appear, 2011.

[25] B. White and J. L. et al, "An integrated experimental environment for distributed systems and networks," in *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

[26] L. Lamport, "The part-time parliament," SRC Research, Tech. Rep., Sep. 1989.

[27] B. Lampson, "The ABCD's of Paxos," in *Proceeding of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC'01).* ACM Press, Aug. 2001, p. 13.

[28] E. Gafni and L. Lamport, "Disk Paxos," in *Proceedings of the 14th International Conference on Distributed Computing (DISC'00)*, 2000, pp. 330–344.

[29] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems*, vol. 20, no. 4, 2002.

[30] R. Guerraoui, R. R. Levy, B. Pochon, and V. Quéma, "Throughput optimal total order broadcast for cluster environments," *ACM Trans. Comput. Syst.*, vol. 28, no. 2, 2010.

[31] P. Marandi, M. Primi, N. Schiper, and F. Pedone, "Ring Paxos: A high-throughput atomic broadcast protocol," in *Dependable Systems and Networks (DSN'10)*, Jun. 2010.

[32] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams, "Replication in the Harp file system," *SIGOPS Oper. Syst. Rev.*, vol. 25, no. 5, pp. 226–238, 1991.

[33] T. Kobus and M. Kokociński, "Design and implementation of Distributed Software Transactional Memory (DSTM)," Master's thesis, Poznan University of Technology, Poznan, Sep 2010.