

Christopher Jones

DRAFT

**Achieving distributed consensus
with Paxos**

Part II Project

Trinity Hall

April 5, 2018

Contents

1	Introduction	1
1.1	Background	1
1.2	Aims	2
2	Preparation	3
2.1	Theoretical background	3
2.1.1	Assumptions of the environment	3
2.1.2	Aim of consensus	4
2.1.3	State machine replication	4
2.1.4	Single-decree Paxos	5
2.1.5	Multi-decree Paxos	6
3	Implementation	11
3.1	Data structures	11
3.1.1	Ballots	11
	Bibliography	13

Chapter 1

Introduction

1.1 Background

Distributed systems suffer from a number of possible errors and failure modes. Unreliability is present in the network where messages can be delayed, re-ordered and dropped and processes can exhibit faulty behaviour such as stalling and crashing. The result of this is that distributed systems can end up inconsistent states and even unable to make progress.

Consensus is the reaching of agreement in the face of such unreliable conditions. Transaction systems, distributed databases and leadership elections are all applications that require consensus in order to remain consistent. Consensus algorithms provide a means by which to reach agreement across in a distributed system in the face of such unreliability; this is crucial in the design of distributed systems.

Paxos is a consensus algorithm first described by Lamport [4] that allows for consensus to be reached under the typical unreliable conditions present in a distributed system. It is a three-phase commit system that relies on processes participating in the *Synod* voting protocol in order to tolerate the failure of a minority of processes.

Paxos is used internally in large-scale production systems such as Google's Chubby [1] distributed lock service, where it is used to maintain consistency between replicas. Microsoft's Autopilot [3] system for data centre management also uses Paxos, again to replicate data across machines. The extreme generality of Paxos allows it to be used as an underlying primitive for various distributed systems techniques. State Machine Replication [10] is a technique whereby any application that behaves like a state machine can be replicated across a number of machines participating in the Paxos protocol. Likewise, atomic broadcast [9] can be implemented with Paxos as an underlying primitive.

Over time Paxos has been extended and modified to emphasise different performance trade-offs. Multi-Paxos is the most typically deployed variant which allows for explicit agreement over a sequence of values. Another example, Fast Paxos [6], is a variant that reduces the number of message delays between proposing a value and it being chosen. More recently, Flexible Paxos [2] is a variant that **relaxes the requirement on same-phase quorums intersecting** in order to improve performance.

There are a number of alternative means of reaching consensus. Viewstamped replication [7] is primarily a replication protocol but can be used as a consensus algorithm. Raft [8] is a modern alternative to Paxos that attempts to reduce the complexity of implementing Paxos. **Why didn't we use these though?**

1.2 Aims

The aim of this project was to produce an implementation of the Multi-Paxos variant of the Paxos algorithm to replicate a **toy** distributed application. This is the variant that is used most widely in production systems and provides a foundation upon which to use state machine replication to replicate an application.

OCaml will be used as the primary development language. **Why OCaml?**

Evaluation and simulator explanation.

Chapter 2

Preparation

2.1 Theoretical background

2.1.1 Assumptions of the environment

When considering developing a system with distributed consensus, it is necessary to consider the assumptions made in the environment in which such a system will operate. This is to ensure there is enough functionality embedded in the consensus system to ensure that consensus is reached under a given set of assumptions.

A *process* (or networked process) is an instance of the program running on a networked machine in a distributed system. Assumptions of these processes that participate in the system:

- Processes can undergo *crash failures*. A crash failure is defined as a process terminating but not entering an invalid state.
- Processes may recover from crash failures. They can rejoin the system in some valid state.
- Processes operate at arbitrary speeds. This cannot be distinguished by other processes from arbitrarily long delays in the network.

Assumptions of the network in which these processes communicate:

- All processes can communicate with one another.
- The network environment is *asynchronous*. That is, messages may take an arbitrarily long time to be delivered.
- Messages may be re-ordered upon delivery.
- Messages may be duplicated in the network.
- Messages may be dropped from the network.

- Messages are not corrupted or modified in the network.
- A message that is received by one process was, at some point in the past, sent by another process.

The last two assumptions assume a system that does not tolerate what are known generally as *Byzantine failures*.

2.1.2 Aim of consensus

With distributed consensus, we wish for a network of processes to agree on some value. In consensus algorithms it is assumed that processes can somehow propose values to one participating processes. The goal of distributed consensus is, given a number of processes that can each propose some value v , that one of the proposed values is chosen. This is the *single-decree* case, that is only one value is proposed by each process and only one is chosen.

In the *multi-decree* case, agreement is reached over a sequence of values. That is, each process will propose a sequence of values v_1, v_2, \dots, v_n and the role of the consensus protocol is to have the system choose one such sequence from all those proposed. This multi-decree case allows for the state machine replication technique to be employed to replicate an application across a number of machines in a distributed system.

2.1.3 State machine replication

A desirable goal of distributed computing is to replicate an application across a number of machines so that each *replica* has the same strongly-consistent view of the application's state. This technique is referred to as State machine replication (SMR); it leads to both for increased fault tolerance and higher availability. Multi-decree consensus protocols provide a primitive by which an application (that behaves like a state machine) can be replicated.

Each process participating in the consensus protocol runs the replicated state machine application, with each process starting in the same state. Then by treating the values proposed in the consensus protocol as *commands* to perform a state transition, then by running a consensus protocol each process will receive the same serialized sequence of commands c_1, c_2, \dots, c_n . These commands are treated as commands to perform a state transition and as such each process perform the same sequence of transitions from the same starting state and thus

Role	Purpose	Number required
Proposer	Propose values to acceptors. Send prepare requests with proposal numbers.	$f + 1$
Acceptor	Decide whether to <i>adopt</i> a proposal based on its proposal number Decide whether to <i>accept</i> a proposal based on a higher numbered proposal having arriving.	$2f + 1$
Learner	Learn value chosen by majority of acceptors	$f + 1$

Table 2.1: Summary of the roles in single-decree Paxos. In this description the system can tolerate the failure of up to f of each given role.

be a replica of the the state machine application.

Before considering how to implement SMR in the multi-decree case, it is useful to examine how Paxos operates in the simpler single-decree case.

2.1.4 Single-decree Paxos

Single-decree Paxos is the variant of the algorithm that allows for a single value to be chosen from a set of proposals and provides a foundation for the multi-decree case that will be considered next. The terminology used here follows Lamport's paper [5] describing the single-decree protocol in simple terms. Processes take the roles of *proposers*, *acceptors* and *learners*, each of which has a designated task in the Paxos algorithm. In reality these roles are often co-located within a single process but it is simply to consider each separately. The prupose of each role and the number of each role required to tolerate f failures is summarised in Table 2.1.

Proposers that wish to propose a value v submit proposals of the form (n, v) , where $n \in \mathbb{N}$ is called a proposal number. Each proposer may propose one proposal at a time and may only use strictly increasing proposal numbers for each proposal. Furthermore, each proposer must use a disjoint set of proposal numbers. The Paxos algorithm is divided into a number of stages described below.

Phase 1a (Prepare phase) A proposer wishing to propose a value first sends a **prepare**(n) message to a majority of the set of acceptors, where n is the highest proposal number it has used so far.

Phase 1b (Promise phase) In this phase an acceptor receiving a `prepare(n)` message must decide whether or not to *adopt* this proposal number. Adopting a proposal number is the act of promising not to accept a future proposal number n' such that $n' < n$. The acceptor will adopt n if it is the highest proposal number it has received thus far, in which case it will reply to the proposer with a `promise(n'', v)` message, where n'' is the highest proposal number it has previously accepted and v is the corresponding proposal's value. Otherwise, it can simply ignore the proposer or send a `NACK` message so the proposer can abandon the proposal.

Phase 2a (Accept phase) Upon receipt of a `promise(n, v)` message from a majority of the set of acceptors, the proposer replies to each with an `accept(n', v')`, where n' is the highest proposal number returned by the acceptors in the promise phase and v' is its corresponding value.

Phase 2b (Commit phase) An acceptor receiving a `accept(n, v)` message from a proposer will decide whether to commit the proposal for v . If the acceptor hasn't made a promise to adopt a proposal number higher than n , then it will commit v , otherwise it will ignore this message or send a `NACK` to the proposer.

Once this process is completed, a majority of the acceptors will have chosen the same proposed value. Learners are required to learn what value was chosen by the majority. A number of different methods can be employed to deliver this information. Acceptors can, on choosing a value to accept, broadcast their decision to the set of learners. An alternative method is to have a distinguished learner (or small subset of) that are sent all decisions which then forward onto the set of learners when they have learned the majority.

Talk about some simple examples with corresponding timing diagrams.

2.1.5 Multi-decree Paxos

Single-decree Paxos can be naively extended by allowing proposers to propose values one at a time. However, this is wasteful as it requires that proposers send `prepare` messages for each proposal they wish to make. A number of optimisations and extensions can be put in place to increase the efficiency of the system. The system here primarily follows **PAXOS MADE MODERATELY COMPLEX** and introduces different types of nodes. **Also discussed here is how to extend the system to use state machine replication.** The new roles and their

Role	Purpose	Number required
Client	Send commands to replicas and receive responses	N/A
Replica	Receive requests from clients. Serialize proposals and send to leaders. Receive decisions and apply to the replicated application state. Handle reconfiguration of set of leaders	$f + 1$
Leader	Request acceptors adopt ballots.	$f + 1$
Acceptor	Fault tolerant distributed memory. Voting protocol.	$2f + 1$

Table 2.2: Summary of the roles in Multi Paxos. In this description the system can tolerate the failure of up to f of each given role. Note that clients do not explicitly participate in the protocol and so there is no requirement on any number being live at any given time.

correspondence to the single-decree roles are summarised in Table 2.2.

Diagram showing the communication pattern of nodes in Mutli-Paxos. Contrast with the single-decree case.

Clients and replicas are introduced to provide a means of implementing the replicated state machine.

Clients

The purpose of clients is to allow for commands to be sent externally to the system which can then be formed into proposals internally. This allows the system to behave in a manner more like that of a typically deployed distributed system (with a client / server architecture) and provides a degree of failure transparency.

A command c takes the form (κ, cid, op) , where κ is a unique identifier for the client, cid is a unique identifier for the client's sent commands and op is the operation the command should perform. Clients broadcast a **request**(c) message

to the replicas and each is issued a **response**(*cid*, *result*) when consensus is reached and it has been applied to each replica's application state.

Replicas

Replicas receive commands and attempt to serialize them by converting each command c into a proposal (s, c) , where $s \in \mathbb{N}$ is a slot number. The slot number describes ordering of the sequence in which the commands should be committed; this is not to be confused with the proposal number n in the single-decree protocol.

Different replicas may form different sequences of proposals and so broadcasts a **propose**(s, c) message to the set of leaders and awaits a **decision**(s', c') message. The resulting decision may differ in its slot number and so the replica may have to re-propose a command it has proposed for the decided slot. Upon receipt of decisions the replica will applying the associated operation to the application state, maintaining the replicated application.

(Also reconfigurations)

Diagram showing message flow between clients and replicas to clarify the last points.

Ballots and pvalues

Explain ballots.

- A ballot may map a command to multiple slots.
- A slot may be mapped to multiple ballots.
- Leaders can attempt to secure adoption of multiple ballots concurrently.

Ballot numbers are either pairs (r, λ) (where $r \in \mathbb{N}$ is called a round number and λ is a leader's unique identifier) or \perp , a specially designated least ballot number.

A *pvalue* is a triple (b, s, c) consisting of a ballot number, a slot number and a command. These are analogous to the (n, v) pairs used in the single-decree case. In the single-decree case, we required that each proposer used a disjoint subset of \mathbb{N} for their proposal numbers. We can avoid this requirement as each ballot number encodes the identifier of the leader directly in its ballot number

(i.e. no two leaders can generate equal ballot numbers).

We require ballot numbers to be totally ordered so that acceptors can compare which ballot number is less than another when choosing whether to adopt or accept. Letting \mathcal{B} denote the set of all ballot numbers, we define the relation $\leq \in \mathcal{B} \times \mathcal{B}$ which satisfies the following two conditions:

$$\forall (n, \lambda), (n', \lambda') \in \mathcal{B}. (n, \lambda) \leq (n', \lambda') \iff (n \leq n') \vee (n = n' \wedge \lambda \leq \lambda') \quad (2.1)$$

$$\forall b \in \mathcal{B}. \perp \leq b \quad (2.2)$$

Note this implies that we require leader identifiers be equipped with a total order relation as well.

Quorums

Explanation of quorum systems.

Let \mathcal{Q} be the set of all quorums of acceptors, that is $\mathcal{Q} = \mathcal{P}(\mathcal{A})$. Quorums are *valid* if they share at least one common member, that is

$$\forall Q_1, Q_2 \in \mathcal{Q}. Q_1 \cap Q_2 \neq \emptyset$$

Hence we can use a majority quorum by requiring that $|Q_1| = |\mathcal{A}| / 2$.

Need to really think about these quorum systems.

The Synod Protocol

The synod protocol is the protocol undertaken by the set of leaders and acceptors in order to decide which command is committed to which slot. The protocol proceeds in two phases similarly to the single-decree case except now leaders and acceptors operate over pvalues.

In the absence of receiving any **propose**(s, c) messages from replicas, leaders attempt to secure an initial ballot with ballot number $(0, \lambda_i)$, where λ_i is the identifier of the **ith** leader. They do this by broadcasting a **phase1a**(b) message in the same format as that below.

Phase 1a Leaders attempt to secure an initial ballot by broadcasting **phase1a**((n, λ)) message.

Phase 1b Acceptors receiving a `phase1a` $((n, \lambda))$ compare (n, λ) to the highest ballot number they have adopted thus far. This is initially \perp so acceptors will adopt the first ballot number they receive automatically. If $b \leq (n, \lambda)$, then the acceptor will adopt this new ballot number. In either case, the acceptor will reply with a

Chapter 3

Implementation

3.1 Data structures

3.1.1 Ballots

```
type t = Bottom
      | Number of int * leader_id
```

Figure 3.1: Types of ballot numbers.

```
type t
val bottom : unit -> t
val init : leader_id -> t
val succ_exn : t -> t
```

Figure 3.2: Excerpt of the type definitions in the Ballot module. These are the types of functions that can be used to generate ballot numbers.

Talk about the correspondence between the definition and the types of ballots. Talk about how, with the interface we've given for ballots, the concrete type is obscured so the compiler will reject a whole host of errors to do with generating invalid ballots.

Bibliography

- [1] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.
- [2] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible paxos: Quorum intersection revisited. *CoRR*, abs/1608.06696, 2016.
- [3] Michael Isard. Autopilot: Automatic data center management. Technical report, April 2007.
- [4] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.
- [5] Leslie Lamport. Paxos made simple. pages 51–58, December 2001.
- [6] Leslie Lamport. Fast paxos. *Distributed Computing*, 19:79–103, October 2006.
- [7] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM.
- [8] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.
- [9] Luís Rodrigues and Michel Raynal. Atomic broadcast in asynchronous crash-recovery distributed systems and its use in quorum-based replication. *IEEE Trans. on Knowl. and Data Eng.*, 15(5):1206–1217, September 2003.
- [10] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.