The Paxos Family of Consensus Protocols

Bryan Turner (bryan.turner@pobox.com)

Oct, 2007

Submitted to Wikipedia.org as an expansion to the article "Paxos_algorithm".

Introduction

Paxos is a family of protocols for solving consensus in a network of unreliable processors. Consensus protocols are the basis for the State Machine Approach to distributed computing, as suggested by Leslie Lamport in [3] and surveyed by Fred Schneider in [14].

Consensus is the process of agreeing on one result among a group of participants. This problem becomes difficult when the participants or their communication medium may experience failures [1].

The State Machine Approach is a technique for converting an algorithm into a fault-tolerant, distributed implementation. Ad-hoc techniques may leave important cases of failures unresolved. The principled approach proposed by Lamport et. al. ensures all cases are handled safely.

The Paxos family of protocols includes a spectrum of tradeoffs between the number of processors, number of message delays before learning the agreed value, the activity level of individual participants, number of messages sent, and types of failures. The convergent property of the Paxos family is their safety from inconsistency [4][7][8][10][9].

Safety Properties

In order to guarantee safety, Paxos defines three safety properties and ensures they are always held, regardless of the pattern of failures:

Nontriviality:
Only proposed values can be learned. [8]

Consistency:
At most one value can be learned (Two different learners cannot learn different values). [8][9]

Liveness(C;L):
If value C has been proposed, then eventually learner L will learn some value (if sufficient processors remain non-faulty). [9]

Preliminaries

In order to simplify the presentation of Paxos, the following assumptions and definitions are made explicit. Techniques to broaden the applicability are known in the literature, and are not covered in this article, see references for further reading.

Processors

- Processors operate at arbitrary speed.
- Processors experience independent, random failures.
- Processors with stable storage may re-join the protocol after failures.
- Processors do not collude, lie, or otherwise attempt to divert the protocol.

(See **Byzantine Paxos** for a solution which tolerates arbitrary failures)

Network

- Processors can send messages to any other processor.
- Messages are sent asynchronously and may take arbitrarily long to deliver.
- Messages may be lost, reordered, or duplicated.
- Messages are delivered without corruption.

(See **Byzantine Paxos** for a solution which tolerates arbitrary failures)

Number of Processors

In general, a consensus algorithm can make progress using 2F+1 processors despite the simultaneous failure of any F processors [2]. However, using reconfiguration, a protocol may be employed which survives more total failures if they do not occur too rapidly:

> "For example, it takes seven servers to tolerate three [simultaneous] failures. In many systems, the best way to achieve the desired degree of fault tolerance is to reconfigure the system to replace failed servers by spares. With reconfiguration, a system that uses three active servers and two spares can tolerate a total of three failures, if a failed server can be replaced by a spare before another failure occurs. Reconfiguration therefore allows fewer processors to tolerate the same total number of failures, though not the same number of simultaneous failures. (In most systems, simultaneous failures are much less likely than successive ones.)" [7]

Roles

Paxos describes the actions of the processes by their roles in the protocol; Client, Acceptor, Proposer, Learner, and Leader. In typical implementations, a single processor may play one or more roles at the same time. This does not affect the correctness of the protocol – it is usual to coalesce roles to improve the latency and/or number of messages in the protocol.

Client

The Client issues a *request* to the distributed system, and waits for a *response*. For instance a “write” request on a file in a distributed file server.

Acceptor

The Acceptors act as the fault-tolerant “memory” of the protocol. Acceptors are collected into groups called Quorums. Any message sent to an Acceptor must be sent to a Quorum of Acceptors, any message received from an Acceptor is ignored unless a copy is received from each Acceptor in a Quorum.

Proposer

A Proposer advocates a client request, attempting to convince the Acceptors to agree on it, and acting as a coordinator to move the protocol forward when conflicts occur.

Learner

Learners act as the replication factor for the protocol. Once a Client request has been agreed on by the Acceptors, the Learner may take action (ie: execute the request and send a response to the client). To improve availability of processing, additional Learners can be added.

Leader

Paxos requires a distinguished Proposer (called the leader) to make progress. Many processes may believe they are leaders, but the protocol only guarantees progress if one of them is eventually chosen. If two processes believe they are leaders, it is possible to stall the protocol by continuously proposing conflicting updates. The safety properties are preserved regardless.

Quorums

Quorums express the safety properties of Paxos by ensuring at least some surviving processor retains knowledge of the results.

Typically, a Quorum is any majority of participating Acceptors. ie: given the set of Acceptors {A,B,C,D}, a majority Quorum would be any three Acceptors: {A,B,C}, {A,C,D}, {A,B,D}, {B,C,D}, etc..

Choice

In Paxos, the leader sometimes has to choose among a set of conflicting values. If a set of values is in conflict, the leader must choose one of the values from the most recent round. The protocol does not specify which value must be chosen, and correctness is guaranteed regardless of the choice. However, it is possible for the choice to impede progress.

A typical Choice function is to select the majority value from the highest round.

Typical Deployment

In most deployments of Paxos, each participating process acts in three roles; Proposer, Acceptor and Learner [11]. This reduces the message complexity significantly, without sacrificing correctness:

> “In Paxos, clients send commands to a leader. During normal operation, the leader receives a client's command, assigns it a new command number i, and then begins the ith instance of the consensus algorithm by sending messages to a set of acceptor processes.” [9]

By merging roles the protocol “collapses” into an efficient client-master-replica style deployment typical of the database community. The benefit of the Paxos family (including implementations with merged roles) is the guarantee of its [safety properties].

A typical implementation's message flow is covered in **Typical Multi-Paxos Deployment**.

Basic Paxos

This protocol is the most basic of the Paxos family; it is **not** the protocol which is typically implemented in a deployment (see **Multi-Paxos**).

Each instance of the Basic Paxos protocol decides on a single output value. The protocol proceeds over several rounds, a successful round has two phases:

Phase 1a: *Prepare*

A **Proposer** (the **leader**) selects a proposal number N and sends a *Prepare* message to a **Quorum** of **Acceptors**.

Phase 1b: *Promise*

If the proposal number N is larger than any previous proposal, then each Acceptor promises not to accept proposals less than N, and sends the value it last accepted for this instance to the Proposer (the leader).

Otherwise a denial is sent.

Phase 2a: *Accept!*

If the Proposer receives responses from a Quorum of Acceptors, it may now **Choose** a value to be agreed upon. If any of the Acceptors have already accepted a value, the leader must Choose a value from this set. Otherwise, the Proposer is free to propose any value.

The Proposer sends an *Accept!* message to a Quorum of Acceptors with the Chosen value.

Phase 2b: *Accepted*

If the Acceptor receives an Accept! message for a proposal it has promised, then it Accepts the value.

Each Acceptor sends an *Accepted* message to the Proposer and every Learner.

Rounds fail when multiple Proposers send conflicting *Prepare* messages, or when the Proposer does not receive a Quorum of responses (*Promise* or *Accepted*). In these cases, another round must be started with a higher proposal number.

Here is a graphic representation of the Basic Paxos protocol. Note that the values returned in the *Promise* message (Va, Vb, Vc) are typically null for the first round of each instance, they are shown below for completeness.

[Message Flow : Basic Paxos (one instance, one successful round)]

```
Client   Proposer      Acceptor     Learner
  |         |          |  |  |       |  |
  X-------->|          |  |  |       |  |  Request
  |         X--------->|->|->|       |  |  Prepare(N)
  |         |<---------X--X--X       |  |  Promise(N,{Va,Vb,Vc})
  |         X--------->|->|->|       |  |  Accept!(N,Vn)
  |         |<---------X--X--X------>|->|  Accepted(N,Vn)
  |<---------------------------------X--X  Response
  |         |          |  |  |       |  |
```

Error Cases in Basic Paxos

The simplest error cases are the failure of a redundant Learner, or failure of an Acceptor when a Quorum of Acceptors remains live. In these cases, the protocol requires no recovery. No additional rounds or messages are required, as shown below:

[Message Flow : Basic Paxos, failure of Acceptor (Quorum size = 2 Acceptors)]

```
Client   Proposer      Acceptor     Learner
  |         |          |  |  |       |  |
  X-------->|          |  |  |       |  |  Request
  |         X--------->|->|->|       |  |  Prepare(N)
  |         |          |  |  !       |  |  !! FAIL !!
  |         |<---------X--X          |  |  Promise(N,{Va,Vb,Vc})
  |         X--------->|->|          |  |  Accept!(N,Vn)
  |         |<---------X--X--------->|->|  Accepted(N,Vn)
  |<---------------------------------X--X  Response
  |         |          |  |          |  |
```

[Message Flow : Basic Paxos, failure of redundant Learner]
```
 Client   Proposer      Acceptor     Learner
   |         |          |  |  |       |  |
   X-------->|          |  |  |       |  |  Request
   |         X--------->|->|->|       |  |  Prepare(N)
   |         |<---------X--X--X       |  |  Promise(N,{Va,Vb,Vc})
   |         X--------->|->|->|       |  |  Accept!(N,Vn)
   |         |<---------X--X--X------>|->|  Accepted(N,Vn)
   |         |          |  |  |       |  !  !! FAIL !!
   |<---------------------------------X     Response
   |         |          |  |  |       |
```

The next failure case is when a Proposer fails after proposing a value, but before agreement is reached. Ignoring Leader election, an example message flow is as follows:

[Message Flow : Basic Paxos, Failure of Proposer (re-election not shown, one instance, two rounds)]
```
Client  Leader         Acceptor     Learner
   |      |             |  |  |       |  |
   X----->|             |  |  |       |  |  Request
   |      X------------>|->|->|       |  |  Prepare(N)
   |      |<------------X--X--X       |  |  Promise(N,{Va,Vb,Vc})
   |      |             |  |  |       |  |
   |      |             |  |  |       |  |  !! Leader fails during broadcast !!
   |      X------------>|  |  |       |  |  Accept!(N,Vn)
   |      !             |  |  |       |  |
   |         |          |  |  |       |  |  !! NEW LEADER !!
   |         X--------->|->|->|       |  |  Prepare(N+1)
   |         |<---------X--X--X       |  |  Promise(N+1,{Vn})
   |         X--------->|->|->|       |  |  Accept!(N+1,Vn)
   |         |<---------X--X--X------>|->|  Accepted(N+1,Vn)
   |<---------------------------------X--X  Response
   |         |          |  |  |       |  |
```

The most complex case is when multiple Proposers believe themselves to be Leaders. For instance the current leader may fail and later recover, but the other Proposers have already re-elected a new leader. The recovered leader has not learned this yet and attempts to begin a round in conflict with the current leader.

[Message Flow : Basic Paxos, Dueling Proposers (one instance, four unsuccessful rounds)]
```
Client   Proposer      Acceptor     Learner
   |      |             |  |  |       |  |
   X----->|             |  |  |       |  |  Request
   |      X------------>|->|->|       |  |  Prepare(N)
   |      |<------------X--X--X       |  |  Promise(N,{Va,Vb,Vc})
   |      !             |  |  |       |  |  !! LEADER FAILS
```

```
  |         |          |  |  |       |  |  !! NEW LEADER (knows N)
  |         X--------->|->|->|       |  |  Prepare(N+1)
  |         |<---------X--X--X       |  |  Promise(N+1,{Va,Vb,Vc})
  |      |  |          |  |  |       |  |  !! OLD LEADER recovers
  |      |  |          |  |  |       |  |  !! OLD LEADER tries N+1, denied
  |      X------------>|->|->|       |  |  Prepare(N+1)
  |      |<------------X--X--X       |  |  Nak(N+1)
  |      |  |          |  |  |       |  |  !! OLD LEADER tries N+2
  |      X------------>|->|->|       |  |  Prepare(N+2)
  |      |<------------X--X--X       |  |  Promise(N+2,{Va,Vb,Vc})
  |      |  |          |  |  |       |  |  !! NEW LEADER proposes, denied
  |      |  X--------->|->|->|       |  |  Accept!(N+1,Vn)
  |      |  |<---------X--X--X       |  |  Nak(N+2)
  |      |  |          |  |  |       |  |  !! NEW LEADER tries N+3
  |      |  X--------->|->|->|       |  |  Prepare(N+3)
  |      |  |<---------X--X--X       |  |  Promise(N+3,{Va,Vb,Vc})
  |      |  |          |  |  |       |  |  !! OLD LEADER proposes, denied
  |      X------------>|->|->|       |  |  Accept!(N+2,Vn)
  |      |<------------X--X--X       |  |  Nak(N+3)
  |      |  |          |  |  |       |  |  ... and so on ...
```

Multi-Paxos

A typical deployment of Paxos requires a continuous stream of agreed values acting as commands to a distributed state machine. If each command is the result of a single instance of the **Basic Paxos** protocol, a significant amount of overhead would result.

If the leader is relatively stable, phase 1 becomes unnecessary. Thus, it is possible to skip phase 1 for future instances of the protocol with the same leader.

To achieve this, the instance number is included along with each value. Multi-Paxos reduces the failure-free message delay (proposal to learning) from 4 delays to 2 delays.

The message flow looks like this:

[Message Flow : Multi-Paxos, first instance with new leader]

```
Client   Proposer      Acceptor     Learner
  |         |          |  |  |       |  | --- First Request ---
  X-------->|          |  |  |       |  |  Request
  |         X--------->|->|->|       |  |  Prepare(N)
  |         |<---------X--X--X       |  |  Promise(N,I,{Va,Vb,Vc})
  |         X--------->|->|->|       |  |  Accept!(N,I,Vn)
  |         |<---------X--X--X------>|->|  Accepted(N,I,Vn)
  |<---------------------------------X--X  Response
  |         |          |  |  |       |  |
```

[Message Flow : Multi-Paxos, subsequent instances with same leader]

```
Client   Proposer      Acceptor     Learner
   |         |          |  |  |       |  |  --- Following Requests ---
   X-------->|          |  |  |       |  |  Request
   |         X--------->|->|->|       |  |  Accept!(N,I+1,W)
   |         |<---------X--X--X------>|->|  Accepted(N,I,W)
   |<---------------------------------X--X  Response
   |         |          |  |  |       |  |
```

Typical Multi-Paxos Deployment

The most common deployment of the Paxos family is Multi-Paxos [11], specialized for participating processors to each be Proposers, Acceptors and Learners. The message flow may be optimized as depicted here:

[Message Flow : Three Server Multi-Paxos, First Instance with New Leader]

```
 Client      Servers
   |         |  |  | --- First Request ---
   X-------->|  |  |  Request
   |         X->|->|  Prepare(N)
   |         |<-X--X  Promise(N,I,{Va,Vb,Vc})
   |         X->|->|  Accept!(N,I,Vn)
   |         |<-X--X  Accepted(N,I)
   |<--------X  |  |  Response
   |         |  |  |
```

[Message Flow : Three Server Multi-Paxos, Subsequent Instances with Same Leader]

```
 Client      Servers
   X-------->|  |  |  Request
   |         X->|->|  Accept!(N,I+1,W)
   |         |<-X--X  Accepted(N)
   |<--------X  |  |  Response
   |         |  |  |
```

Optimizations

A number of optimizations reduce message complexity and size. These optimizations are summarized below:

> "We can save messages at the cost of an extra message delay by having a single distinguished learner that informs the other learners when it finds out that a value has been chosen. Acceptors then send *Accepted* messages only to the distinguished learner. In most applications, the roles of leader and distinguished learner are performed by the same processor.
>
> "A leader can send its *Prepare* and *Accept!* messages just to a quorum of acceptors. As long as all acceptors in that quorum are working and can communicate with the leader and the learners, there is no need for acceptors not in the quorum to do anything.

“Acceptors do not care what value is chosen. They simply respond to *Prepare* and *Accept!* messages to ensure that, despite failures, only a single value can be chosen. However, if an acceptor does learn what value has been chosen, it can store the value in stable storage and erase any other information it has saved there. If the acceptor later receives a *Prepare* or *Accept!* message, instead of performing its Phase1b or Phase2b action, it can simply inform the leader of the chosen value.

“Instead of sending the value v, the leader can send a hash of v to some acceptors in its *Accept!* messages. A learner will learn that v is chosen if it receives *Accepted* messages for either v or its hash from a quorum of acceptors, and at least one of those messages contains v rather than its hash. However, a leader could receive *Promise* messages that tell it the hash of a value v that it must use in its Phase2a action without telling it the actual value of v. If that happens, the leader cannot execute its Phase2a action until it communicates with some process that knows v.” [7]

“A proposer can send its proposal only to the leader rather than to all coordinators. However, this requires that the result of the leader-selection algorithm be broadcast to the proposers, which might be expensive. So, it might be better to let the proposer send its proposal to all coordinators. (In that case, only the coordinators themselves need to know who the leader is.)

“Instead of each acceptor sending *Accepted* messages to each learner, acceptors can send their *Accepted* messages to the leader and the leader can inform the learners when a value has been chosen. However, this adds an extra message delay.

“Finally, observe that phase 1 is unnecessary for round 1 .. The leader of round 1 can begin the round by sending an *Accept!* message with any proposed value.” [8]

Cheap Paxos

Cheap Paxos extends **Basic Paxos** to tolerate F failures with F+1 main processors and F auxiliary processors by dynamically reconfiguring after each failure.

This reduction in processor requirements comes at the expense of liveness; if too many main processors fail in a short time, the system must halt until the auxiliary processors can reconfigure the system. During stable periods, the auxiliary processors take no part in the protocol.

“With only two processors p and q, one processor cannot distinguish failure of the other processor from failure of the communication medium. A third processor is needed. However, that third processor does not have to participate in choosing the sequence of commands. It must take action only in case p or q fails, after which it does nothing while either p or q continues to operate the system by itself. The third processor can therefore be a small/slow/cheap one, or a processor primarily devoted to other tasks.” [7]

A graphic representation of Cheap Paxos is as follows:

[Message Flow : Cheap Multi-Paxos; 3 main Acceptors, 1 Auxiliary Acceptor, Quorum size = 3, showing failure of one main processor and subsequent reconfiguration]

```
            {  Acceptors   }
Proposer     Main       Aux    Learner
|            |  |  |     |       |  -- Phase 2 --
X----------->|->|->|     |       |  Accept!(N,I,V)
|            |  |  !     |       |  --- FAIL! ---
|<-----------X--X--------------->|  Accepted(N,I,V)
|            |  |        |       |  -- Failure detected (only 2 accepted) --
X----------->|->|------->|       |  Accept!(N,I,V)  (re-transmit, include Aux)
|<-----------X--X--------X------>|  Accepted(N,I,V)
|            |  |        |       |  -- Reconfigure : Quorum = 2 --
X----------->|->|        |       |  Accept!(N,I+1,W) (Aux not participating)
|<-----------X--X--------------->|  Accepted(N,I+1,W)
|            |  |        |       |
```

Fast Paxos

Fast Paxos generalizes **Basic Paxos** to reduce end-to-end message delays. In Basic Paxos, the message delay from client request to learning is 3 message delays. Fast Paxos allows 2 message delays, but requires the Client to send its request to multiple destinations.

Intuitively, if the leader has no value to propose, then a client could send an *Accept!* message to the Acceptors directly. The Acceptors would respond as in Basic Paxos, sending *Accepted* messages to the leader and every Learner achieving two message delays from Client to Learner.

If the leader detects a collision, it resolves the collision by sending *Accept!* messages for a new round which are *Accepted* as usual. This coordinated recovery technique requires four message delays from Client to Learner.

The final optimization occurs when the leader specifies a recovery technique in advance, allowing the Acceptors to perform the collision recovery themselves. Thus, uncoordinated collision recovery can occur in three message delays (and only two message delays if all Learners are also Acceptors).

Non-Conflicting Message Flow:

```
Client    Leader         Acceptor      Learner
   |         |          |  |  |  |       |  |
   |         X--------->|->|->|->|       |  |  Any(N,I,Recovery)
   |         |          |  |  |  |       |  |
   X------------------->|->|->|->|       |  |  Accept!(N,I,W)
   |         |<---------X--X--X--X------>|->|  Accepted(N,I,W)
   |<------------------------------------X--X  Response(W)
   |         |          |  |  |  |       |  |
```

Conflicting Message Flow (with uncoordinated recovery). Note: the protocol does not specify how to handle the dropped client request.

```
Client   Leader      Acceptor     Learner
 |  |      |        |  |  |  |      |  |
 |  |      X------->|->|->|->|      |  |  Any(N,I,Recovery)
 |  |      |        |  |  |  |      |  |
 |  |      |        |  |  |  |      |  |  !! Concurrent conflicting proposals
 |  |      |        |  |  |  |      |  |  !!   received in different order
 |  |      |        |  |  |  |      |  |  !!   by the Acceptors
 |  X--------------?|-?|-?|-?|      |  |  Accept!(N,I,V)
 X-----------------?|-?|-?|-?|      |  |  Accept!(N,I,W)
 |  |      |        |  |  |  |      |  |
 |  |      |        |  |  |  |      |  |  !! Acceptors disagree on value
 |  |      |<-------X--X->|->|----->|->|  Accepted(N,I,V)
 |  |      |<-------|<-|<-X--X----->|->|  Accepted(N,I,W)
 |  |      |        |  |  |  |      |  |
 |  |      |        |  |  |  |      |  |  !! Detect collision & recover
 |  |      |<-------X--X--X--X----->|->|  Accepted(N+1,I,W)
 |<---------------------------------X--X  Response(W)
 |  |      |        |  |  |  |      |  |
```

Conflicting Message Flow (merged Acceptor/Learner roles):

```
Client         Servers
 |  |         |  |  |  |
 |  |         X->|->|->|  Any(N,I,Recover)
 |  |         |  |  |  |
 |  |         |  |  |  |  !! Concurrent conflicting proposals
 |  |         |  |  |  |  !!   received in different order
 |  |         |  |  |  |  !!   by the Servers
 |  X--------?|-?|-?|-?|  Accept!(N,I,V)
 X-----------?|-?|-?|-?|  Accept!(N,I,W)
 |  |         |  |  |  |
 |  |         |  |  |  |  !! Servers disagree on value
 |  |         X--X->|->|  Accepted(N,I,V)
 |  |         |<-|<-X--X  Accepted(N,I,W)
 |  |         |  |  |  |
 |  |         |  |  |  |  !! Detect collision & recover
 |<-----------X--X--X--X  Response(W)
 |  |         |  |  |  |
```

Generalized Paxos

Generalized consensus explores the relationship between the operations of a distributed state machine and the consensus protocol used to maintain consistency of that state machine. The main discovery involves optimizations of the consensus protocol when conflicting proposals could be applied to the state machine in any order. ie: The operations proposed by the conflicting proposals are commutative operations of the state machine.

In such cases, the conflicting operations can both be accepted, avoiding the delays required for resolving conflicts and re-proposing the rejected operation.

This concept is further generalized into ever-growing sets of commutative operations, some of which are known to be stable (and thus may be executed). The protocol tracks these sets of operations, ensuring that all proposed commutative operations of one set are stabilized before allowing any non-commuting operation to become stable.

Generalized Paxos Example:

In order to illustrate Generalized Paxos, this example shows a message flow between two concurrently executing clients and a distributed state machine performing the operations of a read/write register with 2 independent register addresses (A and B).

Commutativity Table; marked cells denote interference:

	Read(A)	Write(A)	Read(B)	Write(B)
Read(A)		X		
Write(A)	X	X		
Read(B)				X
Write(B)			X	X

Proposed Series of operations (global order):

```
1:Read(A)
2:Read(B)
3:Write(B)
4:Read(B)
5:Read(A)
6:Write(A)
7:Read(A)
```

Example commutative permutation:

```
{ 1:Read(A),  2:Read(B), 5:Read(A) }
```

```
        { 3:Write(B), 6:Write(A) }
        { 4:Read(B),  7:Read(A)  }
```

Observations:
* `5:Read(A)` may commute in front of `3:Write(B)/4:Read(B)` pair.
* `4:Read(B)` may commute behind the `3:Write(B)/6:Write(A)` pair.
* In practice, a commute occurs only when operations are proposed concurrently.

Generalized Paxos vs. Fast Multi-Paxos

The message flow shows Generalized Paxos performing agreement on seven values in (nominally) 10 message delays. Fast Multi-Paxos would require 15-17 delays for the same sequence (3 delays for each of the three concurrent proposals with uncoordinated recovery, plus at least 2 delays for the eventual re-submission of the three rejected proposals, concurrent re-proposals may add two additional delays).

[Message Flow, responses not shown]
(Note: message abbreviations differ from previous message flows due to specifics of the protocol, see [9] for a full discussion):

```
           {    Acceptors   }
Client      Leader  Acceptor     Learner
 |  |         |      |  |  |         |  |  !! New Leader Begins Round
 |  |         X----->|->|->|         |  |  Prepare(N)
 |  |         |<-----X--X--X         |  |  Promise(N,null)
 |  |         X----->|->|->|         |  |  Phase2Start(N,null)
 |  |         |      |  |  |         |  |
 |  |         |      |  |  |         |  |  !! Concurrent commuting proposals
 |  X--------?|-----?|-?|-?|         |  |  Propose(ReadA)
 X-----------?|-----?|-?|-?|         |  |  Propose(ReadB)
 |  |         X------X-------------->|->|  Accepted(N,<ReadA,ReadB>)
 |  |         |<--------X--X-------->|->|  Accepted(N,<ReadB,ReadA>)
 |  |         |      |  |  |         |  |
 |  |         |      |  |  |         |  |  !! No Conflict, both accepted
 |  |         |      |  |  |         |  |  Stable = <ReadA, ReadB>
 |  |         |      |  |  |         |  |
 |  |         |      |  |  |         |  |  !! Concurrent conflicting proposals
 X-----------?|-----?|-?|-?|         |  |  Propose(<WriteB,ReadA>)
 |  X--------?|-----?|-?|-?|         |  |  Propose(ReadB)
 |  |         |      |  |  |         |  |
 |  |         X------X-------------->|->|  Accepted(N,<WriteB,ReadA> . <ReadB>)
 |  |         |<--------X--X-------->|->|  Accepted(N,<ReadB> . <WriteB,ReadA>)
 |  |         |      |  |  |         |  |
 |  |         |      |  |  |         |  | !! Conflict detected, leader chooses
 |  |         |      |  |  |         |  |    commutative order:
 |  |         |      |  |  |         |  |    V = <ReadA, WriteB, ReadB>
 |  |         |      |  |  |         |  |
 |  |         X----->|->|->|         |  |  Phase2Start(N+1,V)
 |  |         |<-----X--X--X-------->|->|  Accepted(N+1,V)
 |  |         |      |  |  |         |  |  Stable = <ReadA, ReadB> .
 |  |         |      |  |  |         |  |           <ReadA, WriteB, ReadB>
 |  |         |      |  |  |         |  |
 |  |         |      |  |  |         |  |  !! More conflicting proposals
 X-----------?|-----?|-?|-?|         |  |  Propose(WriteA)
 |  X--------?|-----?|-?|-?|         |  |  Propose(ReadA)
 |  |         |      |  |  |         |  |
 |  |         X------X-------------->|->|  Accepted(N+2,<WriteA> . <ReadA>)
 |  |         |<--------X--X-------->|->|  Accepted(N+2,<ReadA> . <WriteA>)
 |  |         |      |  |  |         |  |
 |  |         |      |  |  |         |  |  !! Leader chooses order W
 |  |         X----->|->|->|         |  |  Phase2Start(N+2,W)
 |  |         |<-----X--X--X-------->|->|  Accepted(N+2,W)
 |  |         |      |  |  |         |  |  Stable = <ReadA, ReadB> .
 |  |         |      |  |  |         |  |           <ReadA, WriteB, ReadB> .
 |  |         |      |  |  |         |  |           <WriteA, ReadA>
 |  |         |      |  |  |         |  |
```

Byzantine Paxos

Paxos may also be extended to support arbitrary failures of the participants, including lying, fabrication of messages, collusion with other participants, selective non-participation, etc. These types of failures are called Byzantine Failures, after the solution popularized by Lamport [13].

Byzantine Paxos [10][8] adds an extra message (*Verify*) which acts to distribute knowledge and verify the actions of the other processors:

[Message Flow : Byzantine Multi-Paxos, steady-state operation]
```
Client   Proposer      Acceptor     Learner
   |         |          |  |  |       |  |
   X-------->|          |  |  |       |  |  Request
   |         X--------->|->|->|       |  |  Accept!(N,I,V)
   |         |          X<>X<>X       |  |  Verify(N,I,V) - BROADCAST
   |         |<---------X--X--X------>|->|  Accepted(N,V)
   |<---------------------------------X--X  Response(V)
   |         |          |  |  |       |  |
```

Fast Byzantine Paxos removes this extra delay, since the client sends commands directly to the Acceptors[8]. Note the *Accepted* message in Fast Byzantine Paxos is sent to all Acceptors and all Learners, while Fast Paxos sends *Accepted* messages only to Learners:

[Message Flow : Fast Byzantine Multi-Paxos, steady-state operation]
```
Client    Acceptor     Learner
   |      |  |  |       |  |
   X----->|->|->|       |  |  Accept!(N,I,V)
   |      X<>X<>X------>|->|  Accepted(N,I,V) - BROADCAST
   |<-------------------X--X  Response(V)
   |      |  |  |       |  |
```

The failure scenario is the same for both protocols; Each Learner waits to receive F+1 identical messages from different Acceptors. If this does not occur, the Acceptors themselves will also be aware of it (since they exchanged each other's messages in the broadcast round), and correct Acceptors will re-broadcast the agreed value:

[Message Flow : Fast Byzantine Multi-Paxos, failure scenario]
```
Client    Acceptor     Learner
   |      |  |  !       |  |  !! One Acceptor is faulty
   X----->|->|->!       |  |  Accept!(N,I,V)
   |      X<>X<>X------>|->|  Accepted(N,I,{V,W}) - BROADCAST
   |      |  |  !       |  |  !! Learners receive 2 different commands
   |      |  |  !       |  |  !! Correct Acceptors notice error and choose
   |      X<>X<>X------>|->|  Accepted(N,I,V) - BROADCAST
   |<-------------------X--X  Response(V)
   |      |  |  !       |  |
```

Bibliography

[1] Pease, Marshall, Shostak, Robert; Lamport, Leslie (1980) "Reaching Agreement in the Presence of Faults" Journal of the Association for Computing Machinery 27, 2
http://research.microsoft.com/users/lamport/pubs/pubs.html#reaching

[2] Lamport, Leslie (2004) "Lower Bounds for Asynchronous Consensus" Microsoft Research Technical Report MSR-TR-2004-72
http://research.microsoft.com/users/lamport/pubs/pubs.html#lower-bound

[3] Lamport, Leslie (1978) "Time, Clocks and the Ordering of Events in a Distributed System". Communications of the ACM 21 (7): 558–565
http://research.microsoft.com/users/lamport/pubs/pubs.html#time-clocks

[4] Lamport, Leslie (1998) "The Part-Time Parliament" ACM Transactions on Computer Systems 16 (2): 133–169
http://research.microsoft.com/users/lamport/pubs/pubs.html#lamport-paxos

[5] Lamport, Leslie (2001) "Paxos Made Simple" ACM SIGACT News (Distributed Computing Column) 32, 4
http://research.microsoft.com/users/lamport/pubs/pubs.html#paxos-simple

[6] De Prisco, Roberto; Lampson Butler; Lynch, Nancy (1997) "Revisiting the Paxos Algorithm" Theoretical Computer Science
http://citeseer.ist.psu.edu/deprisco97revisiting.html

[7] Lamport, Leslie; Massa, Mike (2004) "Cheap Paxos" Proceedings of the International Conference on Dependable Systems and Networks (DSN 2004)
http://research.microsoft.com/users/lamport/pubs/pubs.html#web-dsn-submission

[8] Lamport, Leslie (2005) "Fast Paxos" Microsoft Research Technical Report MSR-TR-2005-112
http://research.microsoft.com/users/lamport/pubs/pubs.html#fast-paxos

[9] Lamport, Leslie (2005) "Generalized Consensus and Paxos" Microsoft Research Technical Report MSR-TR-2005-33
http://research.microsoft.com/users/lamport/pubs/pubs.html#generalized

[10] Castro, Miguel (2001) "Practical Byzantine Fault Tolerance"

http://citeseer.ist.psu.edu/castro01practical.html

[11] Chandra, Tushar; Griesemer, Robert; Redstone, Joshua (2007) “Paxos Made Live – An Engineering Perspective” PODC '07: 26th ACM Symposium on Principles of Distributed Computing
http://labs.google.com/papers/paxos_made_live.html

[12] Gray, Jim; Lamport, Leslie “Paxos Commit”
http://research.microsoft.com/users/lamport/pubs/pubs.html#paxos-commit

[13] Lamport, Leslie; Shostak, Robert; Pease, Marshall “The Byzantine Generals Problem” ACM Transactions on Programming Languages and Systems 4, 3, 382-401
http://research.microsoft.com/users/lamport/pubs/pubs.html#byz

[14] Schneider, Fred (1990) “Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial” ACM Computing Surveys 22
http://www.eecs.harvard.edu/cs262/DSbook.c7.pdf