



WIKIPEDIA
The Free Encyclopedia

[Main page](#)

[Contents](#)

[Featured content](#)

[Current events](#)

[Random article](#)

[Donate to Wikipedia](#)

[Wikipedia store](#)

Interaction

[Help](#)

[About Wikipedia](#)

[Community portal](#)

[Recent changes](#)

[Contact page](#)

Tools

[What links here](#)

[Related changes](#)

[Upload file](#)

[Special pages](#)

[Permanent link](#)

[Page information](#)

[Wikidata item](#)

[Cite this page](#)

Print/export

[Create a book](#)

[Download as PDF](#)

[Printable version](#)

Languages

[Español](#)

[Français](#)

[日本語](#)

[Русский](#)

[中文](#)

 [Edit links](#)

Article

[Talk](#)

[Read](#)

[Edit](#)

[View history](#)



Paxos (computer science)

From Wikipedia, the free encyclopedia

(Redirected from [Paxos algorithm](#))

Paxos is a family of protocols for solving [consensus](#) in a network of unreliable processors. 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]

Consensus protocols are the basis for the [state machine replication](#) approach to distributed computing, as suggested by [Leslie Lamport](#)^[2] and surveyed by [Fred B. Schneider](#).^[3] [State machine replication](#) 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 protocol was first published in 1989 and named after a fictional legislative consensus system used on the [Paxos](#) island in Greece.^[4] It was later published as a journal article in 1998.^[5]

The Paxos family of protocols includes a spectrum of trade-offs 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. Although no deterministic fault-tolerant consensus protocol can guarantee progress in an asynchronous network (a result proven in a paper by Fischer, Lynch and Paterson^[6]), Paxos guarantees safety (consistency), and the conditions that could prevent it from making progress are difficult to provoke.^{[5][7][8][9][10]}

Paxos is usually used where durability is required (for example, to replicate a file or a database), in which the amount of durable state could be large. The protocol attempts to make progress even during periods when some bounded number of replicas are unresponsive. There is also a mechanism to drop a permanently failed replica or to add a new replica.

Contents [hide]

1 History

2 Assumptions

2.1 Processors

2.2 Network

2.3 Number of processors

3 Roles

3.1 Quorums

3.2 Proposal Number & Agreed Value

4 Safety and liveness properties

5 Typical deployment

6 Basic Paxos

6.1 Phase 1a: *Prepare*

6.2 Phase 1b: *Promise*

6.3 Phase 2a: *Accept Request*

6.4 Phase 2b: *Accepted*

6.5 Message flow: Basic Paxos

6.6 Error cases in basic Paxos

6.7 Message flow: Basic Paxos, failure of Acceptor

6.8 Message flow: Basic Paxos, failure of redundant Learner

6.9 Message flow: Basic Paxos, failure of Proposer

6.10 Message flow: Basic Paxos, dueling Proposers

7 Multi-Paxos

7.1 Message flow: Multi-Paxos, start

7.2 Message flow: Multi-Paxos, steady-state

7.3 Typical Multi-Paxos Collapsed Roles deployment

7.4 Message flow: Multi-Paxos Collapsed Roles, start

7.5 Message flow: Multi-Paxos Collapsed Roles, steady state

8 Optimizations

9 Cheap Paxos

9.1 Message flow: Cheap Multi-Paxos
10 Fast Paxos
10.1 Message flow: Fast Paxos, non-conflicting
10.2 Message flow: Fast Paxos, conflicting proposals
10.3 Message flow: Fast Paxos, collapsed roles
11 Generalized Paxos
11.1 Example
11.2 Message flow: Generalized Paxos (example)
11.3 Performance
12 Byzantine Paxos
12.1 Message flow: Byzantine Multi-Paxos, steady state
12.2 Message flow: Fast Byzantine Multi-Paxos, steady state
12.3 Message flow: Fast Byzantine Multi-Paxos, failure
13 Production use of Paxos
14 See also
15 References
16 External links

History [\[edit\]](#)

The topic predates the protocol. In 1988, [Lynch](#), [Dwork](#) and [Stockmeyer](#) had demonstrated ^[11] the solvability of consensus in a broad family of "partially synchronous" systems. Paxos has strong similarities to a protocol used for agreement in [viewstamped replication](#), first published by Oki and [Liskov](#) in 1988, in the context of distributed transactions.^[12] Notwithstanding this prior work, Paxos offered a particularly elegant formalism, and included one of the earliest proofs of safety for a fault-tolerant distributed consensus protocol.

Reconfigurable state machines have strong ties to prior work on reliable group multicast protocols that support dynamic group membership, for example [Birman](#)'s work in 1985 and 1987 on the [virtually synchronous gbcast](#)^[13] protocol. However, it should be noted that gbcast is unusual in supporting durability and addressing partitioning failures. Most reliable multicast protocols lack these properties, which are required for implementations of the state machine replication model. This point is elaborated in a paper by [Lamport](#), [Malkhi](#) and Zhou.^[14]

Assumptions [\[edit\]](#)

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.

Processors [\[edit\]](#)

- Processors operate at arbitrary speed.
- Processors may experience failures.
- Processors with stable storage may re-join the protocol after failures (following a crash-recovery failure model).
- Processors do not collude, lie, or otherwise attempt to subvert the protocol. (That is, [Byzantine failures](#) don't occur. See [Byzantine Paxos](#) for a solution that tolerates failures that arise from arbitrary/malicious behavior of the processes.)

Network [\[edit\]](#)

- 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. (That is, Byzantine failures don't occur. See [Byzantine Paxos](#) for a solution which tolerates corrupted messages that arise from arbitrary/malicious behavior of the messaging channels.)

Number of processors [\[edit\]](#)

In general, a consensus algorithm can make progress using $2F+1$ processors despite the simultaneous failure of any F processors.^[15] However, using reconfiguration, a protocol may be employed which survives any number of total failures as long as no more than F fail simultaneously.

Roles [\[edit\]](#)

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 (Voters)

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 (i.e.: 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, they may stall the protocol by continuously proposing conflicting updates. However, the [safety properties](#) are still preserved in that case.

Quorums [\[edit\]](#)

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

Quorums are defined as subsets of the set of Acceptors such that any two subsets (that is, any two Quorums) share at least one member. Typically, a Quorum is any majority of participating Acceptors. For example, 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}. More generally, arbitrary positive weights can be assigned to Acceptors and a Quorum defined as any subset of Acceptors with the summary weight greater than half of the total weight of all Acceptors.

Proposal Number & Agreed Value [\[edit\]](#)

Each attempt to define an agreed value v is performed with proposals which may or may not be accepted by Acceptors. Each proposal is uniquely numbered for a given Proposer. The value corresponding to a numbered proposal can be computed as part of running the Paxos protocol, but need not be.

Safety and liveness properties [\[edit\]](#)

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

Non-triviality

Only proposed values can be learned.^[8]

Safety

At most one value can be learned (i.e., 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 processes remain non-faulty).^[9]

Typical deployment [\[edit\]](#)

In most deployments of Paxos, each participating process acts in three roles; Proposer, Acceptor and Learner.^[16] 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 i th instance of the

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

A typical implementation's message flow is covered in the section [Multi-Paxos](#).

Basic Paxos [\[edit\]](#)

This protocol is the most basic of the Paxos family. 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. A Proposer should not initiate Paxos if it cannot communicate with at least a Quorum of Acceptors:

Phase 1a: *Prepare* [\[edit\]](#)

A [Proposer](#) (the [leader](#)) creates a proposal identified with a number N. This number must be greater than any previous proposal number used by this Proposer. Then, it sends a *Prepare* message containing this proposal to a [Quorum](#) of [Acceptors](#). The Proposer decides who is in the Quorum.

Phase 1b: *Promise* [\[edit\]](#)

If the proposal's number N is higher than any previous proposal number received from any Proposer by the Acceptor, then the Acceptor must return a promise to ignore all future proposals having a number less than N. If the Acceptor accepted a proposal at some point in the past, it must include the previous proposal number and previous value in its response to the Proposer.

Otherwise, the Acceptor can ignore the received proposal. It does not have to answer in this case for Paxos to work. However, for the sake of optimization, sending a denial (*Nack*) response would tell the Proposer that it can stop its attempt to create consensus with proposal N.

Phase 2a: *Accept Request* [\[edit\]](#)

If a Proposer receives enough promises from a Quorum of Acceptors, it needs to set a value to its proposal. If any Acceptors had previously accepted any proposal, then they'll have sent their values to the Proposer, who now must set the value of its proposal to the value associated with the highest proposal number reported by the Acceptors. If none of the Acceptors had accepted a proposal up to this point, then the Proposer may choose any value for its proposal.^[17]

The Proposer sends an *Accept Request* message to a Quorum of Acceptors with the chosen value for its proposal.

Phase 2b: *Accepted* [\[edit\]](#)

If an Acceptor receives an Accept Request message for a proposal N, it must accept it **if and only if** it has not already promised to only consider proposals having an identifier greater than N. In this case, it should register the corresponding value v and send an *Accepted* message to the Proposer and every Learner. Else, it can ignore the Accept Request.

Note that an Acceptor can accept multiple proposals. These proposals may even have different values in the presence of certain failures. However, the Paxos protocol will guarantee that the Acceptors will ultimately agree on a single value.

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.

Notice that when Acceptors accept a request, they also acknowledge the leadership of the Proposer. Hence, Paxos can be used to select a leader in a cluster of nodes.

Here is a graphic representation of the Basic Paxos protocol. Note that the values returned in the *Promise* message are null the first time a proposal is made, since no Acceptor has accepted a value before in this round.

Message flow: Basic Paxos [\[edit\]](#)

(first round is successful)

Client	Proposer	Acceptor	Learner

X----->						Request
	X-----> -> ->					Prepare (1)
	<-----X--X--X					Promise (1, {Va,Vb,Vc})
	X-----> -> ->					Accept! (1,Vn)
	<-----X--X--X-----> ->					Accepted (1,Vn)
<-----X--X						Response

Vn = highest of (Va,Vb,Vc)

Error cases in basic Paxos [\[edit\]](#)

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 [\[edit\]](#)

(Quorum size = 2 Acceptors)

Client	Proposer	Acceptor	Learner
X----->			
	X-----> -> ->		
		!	
	<-----X--X		
	X-----> ->		
	<-----X--X-----> ->		
<-----X--X			

Request
Prepare (1)
!! FAIL !!
Promise (1, {null,null, null})
Accept! (1,V)
Accepted (1,V)
Response

Message flow: Basic Paxos, failure of redundant Learner [\[edit\]](#)

Client	Proposer	Acceptor	Learner
X----->			
	X-----> -> ->		
	<-----X--X--X		
	X-----> -> ->		
	<-----X--X--X-----> ->		
			!
<-----X			

Request
Prepare (1)
Promise (1, {null,null,null})
Accept! (1,V)
Accepted (1,V)
!! FAIL !!
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 [\[edit\]](#)

(re-election not shown, one instance, two rounds)

Client	Proposer	Acceptor	Learner
X----->			
	X-----> -> ->		
	<-----X--X--X		
	X-----> -> ->		
	!		
	X-----> -> ->		
	<-----X--X--X		
	X-----> -> ->		
	<-----X--X--X-----> ->		
<-----X--X			

Request
Prepare (1)
Promise (1, {null, null, null})
!! Leader fails during broadcast !!
Accept! (1,Va)
!! NEW LEADER !!
Prepare (2)
Promise (2, {null, null, null})
Accept! (2,V)
Accepted (2,V)
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 [\[edit\]](#)

(one instance, four unsuccessful rounds)

Client	Proposer	Acceptor	Learner
X----->			Request
	X-----> -> ->		Prepare(1)
	<-----X--X--X		Promise(1, {null,null,null})
	!		!! LEADER FAILS
			!! NEW LEADER (knows last number was 1)
	X-----> -> ->		Prepare(2)
	<-----X--X--X		Promise(2, {null,null,null})
			!! OLD LEADER recovers
			!! OLD LEADER tries 2, denied
	X-----> -> ->		Prepare(2)
	<-----X--X--X		Nack(2)
			!! OLD LEADER tries 3
	X-----> -> ->		Prepare(3)
	<-----X--X--X		Promise(3, {null,null,null})
			!! NEW LEADER proposes, denied
	X-----> -> ->		Accept!(2, Va)
	<-----X--X--X		Nack(3)
			!! NEW LEADER tries 4
	X-----> -> ->		Prepare(4)
	<-----X--X--X		Promise(4, {null,null,null})
			!! OLD LEADER proposes, denied
	X-----> -> ->		Accept!(3, Vb)
	<-----X--X--X		Nack(4)
			... and so on ...

Multi-Paxos [\[edit\]](#)

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 I is included along with each value. Multi-Paxos reduces the failure-free message delay (proposal to learning) from 4 delays to 2 delays.

Message flow: Multi-Paxos, start [\[edit\]](#)

(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, Vm)
	<-----X--X--X-----> ->		Accepted(N, I, Vm)
<-----X--X			Response

V_m = highest of (Va, Vb, Vc)

Message flow: Multi-Paxos, steady-state [\[edit\]](#)

(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+1, W)
<-----X--X				Response

Typical Multi-Paxos Collapsed Roles deployment [\[edit\]](#)

The most common deployment of the Paxos family is Multi-Paxos,^[16] specialized for participating processors to each be Proposers, Acceptors and Learners. The message flow with roles collapsed may be optimized as depicted here:

Message flow: Multi-Paxos Collapsed Roles, start [\[edit\]](#)

(first instance with new leader)

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

Message flow: Multi-Paxos Collapsed Roles, steady state [\[edit\]](#)

(subsequent instances with same leader)

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

Optimizations [\[edit\]](#)

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 [\[edit\]](#)

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]

Message flow: Cheap Multi-Paxos [\[edit\]](#)

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 [\[edit\]](#)

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).

Message flow: Fast Paxos, non-conflicting [\[edit\]](#)

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


Commutativity Table

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

Note that X in this table indicates operations which are non-commutative.

A possible sequence of operations :

```
<1:Read(A), 2:Read(B), 3:Write(B), 4:Read(B), 5:Read(A), 6:Write(A)>
```

Since **5:Read(A)** commutes with both **3:Write(B)** and **4:Read(B)**, one possible permutation equivalent to the previous order is the following:

```
<1:Read(A), 2:Read(B), 5:Read(A), 3:Write(B), 4:Read(B), 6:Write(A)>
```

In practice, a commute occurs only when operations are proposed concurrently.

Message flow: Generalized Paxos (example) [\[edit\]](#)

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
| |         | |    | |    | |
| |         X----->|->|->| |    !! New Leader Begins Round
| |         |<-----X--X--X | |    Prepare (N)
| |         X----->|->|->| |    Promise (N,null)
| |         | |    | |    | |    Phase2Start (N,null)
| |         | |    | |    | |
| |         | |    | |    | |    !! Concurrent commuting proposals
| X-----?|------?|-?|-?| |    Propose (ReadA)
X-----?|------?|-?|-?| |    Propose (ReadB)
| |         |<-----X-->----->|->| |    Accepted (N,<ReadA,ReadB>)
| |         |<-----X--X----->|->| |    Accepted (N,<ReadB,ReadA>)
| |         | |    | |    | |
| |         | |    | |    | |    !! No Conflict, both stable
| |         | |    | |    | |    V = <ReadA, ReadB>
| |         | |    | |    | |
| |         | |    | |    | |    !! Concurrent conflicting proposals
X-----?|------?|-?|-?| |    Propose (WriteB)
| X-----?|------?|-?|-?| |    Propose (ReadB)
| |         | |    | |    | |
| |         |<-----X----- | |    Accepted (N,V.<WriteB,ReadB>)
| |         |<-----X----- | |    Accepted (N,V.<ReadB,WriteB>)
| |         | |    | |    | |
| |         | |    | |    | |    !! Conflict detected at the leader.
| |         | |    | |    | |
| |         X----->|->|->| |    Prepare (N+1)
| |         |<-----X----- | |    Promise (N+1, N, V.<WriteB,ReadB>)
| |         |<-----X----- | |    Promise (N+1, N, V.<ReadB,WriteB>)
| |         |<-----X----- | |    Promise (N+1, N, V)
| |         | |    | |    | |
| |         X----->|->|->| |    Phase2Start (N+1,V.<WriteB,ReadB>)
| |         |<-----X--X--X----->|->| |    Accepted (N+1,V.<WriteB,ReadB>)
| |         | |    | |    | |
| |         | |    | |    | |    !! New stable sequence
| |         | |    | |    | |    U = <ReadA, ReadB>, WriteB, ReadB>
| |         | |    | |    | |
| |         | |    | |    | |    !! More conflicting proposals
X-----?|------?|-?|-?| |    Propose (WriteA)
| X-----?|------?|-?|-?| |    Propose (ReadA)
| |         | |    | |    | |
```

							!! This time spontaneously ordered by the
network							
			[<-----X--X-----> ->				Accepted (N+1, U.<WriteA, ReadA>)

Performance [\[edit\]](#)

The above message flow shows us that Generalized Paxos can leverage operation semantics to avoid collisions when the spontaneous ordering of the network fails. This allows the protocol to be in practice quicker than Fast Paxos. However, when a collision occurs, Generalized Paxos needs two additional round trips to recover. This situation is illustrated with operations WriteB and ReadB in the above schema.

In the general case, such round trips are unavoidable and comes from the fact that multiple commands might be accepted during a round. This makes the protocol more expensive than Paxos when conflicts are frequent. Hopefully two possible refinements of Generalized Paxos are possible to improve recovery time.^[18]

- First, if the coordinator is part of every quorum of acceptors (round N is said *centered*), then to recover at round N+1 from a collision at round N, the coordinator skip phase 1 and proposes at phase 2 the sequence it accepted last during round N. This reduces the cost of recovery to a single round trip.
- Second, if both rounds N and N+1 are centered around the same coordinator, when an acceptor detects a collision at round N, it proposes at round N+1 a sequence suffixing both (i) the sequence accepted at round N by the coordinator and (ii) the greatest non-conflicting prefix it accepted at round N. For instance, if the coordinator and the acceptor accepted respectively at round N <WriteB, ReadB> and <ReadB, ReadA> , the acceptor will spontaneously accept <WriteB, ReadB, ReadA> at round N+1. With this variation, the cost of recovery is a single message delay which is obviously optimal.

Byzantine Paxos [\[edit\]](#)

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.^[19]

Byzantine Paxos^{[8][10]} 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 [\[edit\]](#)

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-----> ->			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 [\[edit\]](#)

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 [\[edit\]](#)

Client	Acceptor	Learner
	!	
X-----> -> ->!		!! One Acceptor is faulty
	X<>X<>X-----> ->	Accept! (N, I, V)
	!	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)
	!	

Production use of Paxos [\[edit\]](#)













- Google uses the Paxos algorithm in their Chubby [distributed lock service](#) in order to keep replicas consistent in case of failure. Chubby is used by [BigTable](#) which is now in production in Google Analytics and other products.
- [Google Spanner](#) and Megastore use the Paxos algorithm internally.
- The [OpenReplica replication service](#) [\[1\]](#) uses Paxos to maintain replicas for an open access system that enables users to create fault-tolerant objects. It provides high performance through concurrent rounds and flexibility through dynamic membership changes.
- IBM supposedly uses the Paxos algorithm in their [IBM SAN Volume Controller](#) product to implement a general purpose fault-tolerant virtual machine used to run the configuration and control components of the [storage virtualization](#) services offered by the cluster.^{[\[citation needed\]](#)}
- Microsoft uses Paxos in the [Autopilot cluster management service](#) [\[2\]](#) from Bing.
- [WANDisco](#) have implemented Paxos within their DConE active-active replication technology.^{[\[20\]](#)}
- [XtreemFS](#) uses a Paxos-based [lease](#) negotiation algorithm for fault-tolerant and consistent replication of file data and metadata.^{[\[21\]](#)}
- Heroku uses [Doozerd](#) [\[3\]](#) which implements Paxos for its consistent distributed data store.
- [Ceph](#) uses Paxos as part of the monitor processes to agree which OSDs are up and in the cluster.
- The [Clustrix](#) distributed SQL database uses Paxos for [distributed transaction resolution](#) [\[4\]](#).
- [Neo4j](#) HA graph database implements Paxos, replacing [Apache ZooKeeper](#) from v1.9
- VMware NSX Controller uses Paxos-based algorithm within NSX Controller cluster.
- [Amazon Web Services](#) uses the Paxos algorithm extensively to power its platform.^{[\[22\]](#)}
- [Nutanix](#) implements the Paxos algorithm in Cassandra for [metadata](#) [\[5\]](#).

See also [\[edit\]](#)

- [Chandra–Toueg consensus algorithm](#)
- [Raft consensus algorithm](#)
- [State machine replication](#)
- [Isis2](#), a library implementing Paxos in a [virtually synchronous](#) group model.
- [Blockchain Database](#), a distributed consensus model used in [Bitcoin](#)

References [\[edit\]](#)

- ^{[^](#)} Pease, Marshall; Shostak, Robert; Lamport, Leslie (April 1980). "Reaching Agreement in the Presence of Faults" [\[6\]](#). *Journal of the Association for Computing Machinery* **27** (2). Retrieved 2007-02-02.
- ^{[^](#)} Lamport, Leslie (July 1978). "Time, Clocks and the Ordering of Events in a Distributed System" [\[7\]](#). *Communications of the ACM* **21** (7): 558–565. doi:10.1145/359545.359563 [\[8\]](#). Retrieved 2007-02-02.
- ^{[^](#)} Schneider, Fred (1990). "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial" [\[9\]](#) (PDF). *ACM Computing Surveys* **22**: 299. doi:10.1145/98163.98167 [\[10\]](#).
- ^{[^](#)} [Leslie Lamport's history of the paper](#) [\[11\]](#)
- ^{[^](#)} ^{[a](#)} ^{[b](#)} Lamport, Leslie (May 1998). "The Part-Time Parliament" [\[12\]](#). *ACM Transactions on Computer Systems* **16** (2): 133–169. doi:10.1145/279227.279229 [\[13\]](#). Retrieved 2007-02-02.
- ^{[^](#)} Fischer, M. (April 1985). "Impossibility of distributed consensus with one faulty process" [\[14\]](#). *Journal of the ACM* **32** (2): 374–382. doi:10.1145/3149.214121 [\[15\]](#).
- ^{[^](#)} ^{[a](#)} ^{[b](#)} ^{[c](#)} Lamport, Leslie; Massa, Mike (2004). *Cheap Paxos* [\[16\]](#). Proceedings of the [International Conference on Dependable Systems and Networks](#) (DSN 2004).
- ^{[^](#)} ^{[a](#)} ^{[b](#)} ^{[c](#)} ^{[d](#)} ^{[e](#)} ^{[f](#)} Lamport, Leslie (2005). "Fast Paxos" [\[17\]](#).
- ^{[^](#)} ^{[a](#)} ^{[b](#)} ^{[c](#)} ^{[d](#)} ^{[e](#)} Lamport, Leslie (2005). "Generalized Consensus and Paxos" [\[18\]](#).

10. ^a ^b Castro, Miguel (2001). "Practical Byzantine Fault Tolerance"  (PDF).
11. ^a Dwork, Cynthia; Lynch, Nancy; Stockmeyer, Larry (April 1988). "Consensus in the Presence of Partial Synchrony"  (PDF). *Journal of the ACM* **35** (2): 288–323. doi:10.1145/42282.42283 .
12. ^a Oki, Brian; Liskov, Barbara (1988). *Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems* . PODC '88: Proceedings of the seventh annual ACM Symposium on Principles of Distributed Computing. pp. 8–17. doi:10.1145/62546.62549 .
13. ^a Birman, Kenneth; Joseph, Thomas (February 1987). "Reliable Communication in the Presence of Failures". *ACM Transactions on Computer Systems*: 47–76.
14. ^a Lamport, Leslie; Malkhi, Dahlia; Zhou, Lidong (March 2010). "Reconfiguring a State Machine". *SIGACT News* **41** (1): 63–73. doi:10.1145/1753171.1753191 .
15. ^a Lamport, Leslie (2004). "Lower Bounds for Asynchronous Consensus" .
16. ^a ^b Chandra, Tushar; Griesemer, Robert; Redstone, Joshua (2007). "Paxos Made Live – An Engineering Perspective" . *PODC '07: 26th ACM Symposium on Principles of Distributed Computing*.
17. ^a Lamport, Leslie (2001). *Paxos Made Simple*  *ACM SIGACT News (Distributed Computing Column)* 32, 4 (Whole Number 121, December 2001) 51-58.
18. ^a Pierre, Sutra; Marc, Shapiro (2011). *Fast Genuine Generalized Consensus*  (PDF). SRDS'11: 30th IEEE Symposium on Reliable Distributed Systems.
19. ^a Lamport, Leslie; Shostak, Robert; Pease, Marshall (July 1982). "The Byzantine Generals Problem" . *ACM Transactions on Programming Languages and Systems* **4** (3): 382–401. doi:10.1145/357172.357176 . Retrieved 2007-02-02.
20. ^a Aahlad et al.(2011). "The Distributed Coordination Engine (DConE)" . WANdisco white paper.
21. ^a Kolbeck, Björn; Höggqvist, Mikael; Stender, Jan; Hupfeld, Felix (2011). "Flease - Lease Coordination without a Lock Server" . 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2011).
22. ^a Vermeulen, Al; Sivasubramanian, Swami (2014). "Under the Covers of AWS: Core Distributed Systems Primitives That Power Our Platform" . AWS re:invent 2014.

External links

- Leslie Lamport's home page 
- Paxos Made Simple 
- Revisiting the Paxos Algorithm 
- Paxos Commit 
- Google Whitepaper: Chubby Distributed Lock Service 
- Google Whitepaper: Bigtable A Distributed Storage System for Structured Data 
- Survey of Paxos Algorithms (2007) 
- OpenReplica Open Replication Service 
- FTFile: Fault Tolerant File library 
- Isis2 library (the SafeSend primitive is a free, open source implementation of Paxos) 
- Mencius - Circular rotating Paxos for geo-distributed systems 
- WANdisco - Active-Active Replication solutions for Hadoop, Subversion & GIT 
- libpaxos, a collection of open source implementations of the Paxos algorithm 
- libpaxos-cpp, a C++ implementation of the paxos distributed consensus algorithm 
- JBP - Java Byzantine Paxos 
- erlpaxos, Paxos by Erlang 
- paxos - Straight-forward paxos implementation in Python & Java 
- Manhattan Paxos (mpaxos), Paxos in C, supporting multiple paxos groups and efficient transactions across them. 
- Clustering with Neo4j 
- HT-Paxos 

Categories: Distributed algorithms | Fault-tolerant computer systems

This page was last modified on 28 August 2015, at 14:09.

Text is available under the [Creative Commons Attribution-ShareAlike License](#); additional terms may apply. By using this site, you agree to the [Terms of Use](#) and [Privacy Policy](#). Wikipedia® is a registered trademark of the [Wikimedia Foundation, Inc.](#), a non-profit organization.

[Privacy policy](#) [About Wikipedia](#) [Disclaimers](#) [Contact Wikipedia](#) [Developers](#) [Mobile view](#)

