Paxos (computer science)

From Wikipedia, the free encyclopedia

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

History

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] Not withstanding 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

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

- 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

- 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

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

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 can form Quorums (cf. the definition of Quorum below). 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 (e.g., Promises sent to a Proposer, or Accepted messages sent to a Learner).

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

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

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

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

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 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 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

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*

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*

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*

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*

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 any prepare 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.

A Learner learns a value only if it receives a Quorum of *Accepted* messages with the same value.

Message flow: Basic Paxos

(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

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(1)
   |         |          |  |  !       |  |  !! FAIL !!
   |         |<---------X--X          |  |  Promise(1,{null,null})
   |         X--------->|->|          |  |  Accept!(1,V)
   |         |<---------X--X--------->|->|  Accepted(1,V)
   |<---------------------------------X--X  Response
   |         |          |  |          |  |
```

Message flow: Basic Paxos, failure of redundant Learner

```
Client   Proposer      Acceptor     Learner
   |         |          |  |  |       |  |
   X-------->|          |  |  |       |  |  Request
   |         X--------->|->|->|       |  |  Prepare(1)
   |         |<---------X--X--X       |  |  Promise(1,{null,null,null})
   |         X--------->|->|->|       |  |  Accept!(1,V)
   |         |<---------X--X--X------>|->|  Accepted(1,V)
   |         |          |  |  |       |  !  !! 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  Proposer        Acceptor     Learner
   |      |             |  |  |       |  |
   X----->|             |  |  |       |  |  Request
   |      X------------>|->|->|       |  |  Prepare(1)
   |      |<------------X--X--X       |  |  Promise(1,{null, null, null})
   |      |             |  |  |       |  |
   |      |             |  |  |       |  |  !! Leader fails during broadcast !!
   |      X------------>|  |  |       |  |  Accept!(1,Va)
   |      !             |  |  |       |  |
   |         |          |  |  |       |  |  !! NEW LEADER !!
   |         X--------->|->|->|       |  |  Prepare(2)
   |         |<---------X--X--X       |  |  Promise(2,{null, null, null})
   |         X--------->|->|->|       |  |  Accept!(2,V)
   |         |<---------X--X--X------>|->|  Accepted(2,V)
   |<---------------------------------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(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 ...
```

Note that Basic Paxos is a stripped down model of the core of every version of Paxos used to prove the correctness of family of Paxos algorithms, as shown in the paper Paxos Made Simple.[17]

Multi-Paxos

If each command is the result of a single instance of the Basic Paxos protocol a significant amount of overhead would result. The paper Paxos Made Simple[17] defines Paxos to be what is commonly called "Multi-Paxos" which in steady state uses a distinguished leader to coordinate an infinite stream of commands. A typical deployment of Paxos uses a continuous stream of agreed values acting as commands to update a distributed state machine.

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

(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
   |         |          |  |  |       |  |
```

Vm = highest of (Va, Vb, Vc)

Message flow: Multi-Paxos, steady-state

(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

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

(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

(subsequent instances with same leader)

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

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).

Message flow: Fast Paxos, non-conflicting

```
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)
   |         |          |  |  |  |       |  |
```

Message flow: Fast Paxos, conflicting proposals

Conflicting proposals 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)
 |  |      |        |  |  |  |      |  |
```

Message flow: Fast Paxos, collapsed roles

(merged Acceptor/Learner roles)

```
Client         Servers
 |  |         |  |  |  |
 |  |         X->|->|->|  Any(N,I,Recovery)
 |  |         |  |  |  |
 |  |         |  |  |  |  !! 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 the replicated state machine and the consensus protocol that implements it. The main discovery involves optimizations of Paxos when conflicting proposals could be applied in any order. i.e., when the proposed operations are commutative operations for 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 operations.

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

Example

In order to illustrate Generalized Paxos, the example below shows a message flow between two concurrently executing clients and a replicated state machine implementing read/write operations over two distinct registers A and B.

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)

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-->-->-------->|->|  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

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

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

```
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

```
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

```
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)
   |      |  |  !       |  |
```

Production use of Paxos

- The Petal project from DEC SRC was likely the first system to use Paxos, in this case for widely replicated global information (e.g., which machines are in the system).[20]
- 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.
- The Infinit peer-to-peer file system relies on Paxos to maintain consistency among replicas while allowing for quorums to evolve in size.
- Google Spanner and Megastore use the Paxos algorithm internally.
- The OpenReplica replication service (http://openreplica.org) 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.

- Microsoft uses Paxos in the Autopilot cluster management service (http://research.microsoft.com/pubs/64604/osr2007.pdf) from Bing.
- WANdisco have implemented Paxos within their DConE active-active replication technology.[21]
- XtreemFS uses a Paxos-based lease negotiation algorithm for fault-tolerant and consistent replication of file data and metadata.[22]
- Heroku uses Doozerd (https://github.com/ha/doozerd) 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 (http://docs.clustrix.com/display/CLXDOC/Consistency,+Fault+Tolerance,+and+Availability).
- 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.[23]
- Nutanix implements the Paxos algorithm in Cassandra for metadata (http://stevenpoitras.com/the-nutanix-bible/#scalable_metadata).
- Apache Mesos uses Paxos algorithm for its replicated log (http://mesos.apache.org/blog/mesos-0-17-0-released-featuring-autorecovery/) coordination.
- Windows Fabric used by many of the Azure services make use of the paxos algorithm for replication between nodes in a cluster
- Oracle NoSQL Database leverages Paxos-based automated fail-over election process in the event of a master replica node failure to minimize downtime.[24]

See also

- Blockchain database, a distributed consensus model used in Bitcoin
- Chandra–Toueg consensus algorithm
- Distributed algorithm
- Vsync, a library implementing Paxos in a virtually synchronous group model
- Raft consensus algorithm
- State machine replication

References

1. Pease, Marshall; Shostak, Robert; Lamport, Leslie (April 1980). "Reaching Agreement in the Presence of Faults". *Journal of the Association for Computing Machinery*. **27** (2): 228–234. doi:10.1145/322186.322188. Retrieved 2007-02-02.
2. Lamport, Leslie (July 1978). "Time, Clocks and the Ordering of Events in a Distributed System". *Communications of the ACM*. **21** (7): 558–565. doi:10.1145/359545.359563. Retrieved 2007-02-02.
3. Schneider, Fred (1990). "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial" (PDF). *ACM Computing Surveys*. **22**: 299–319. doi:10.1145/98163.98167.
4. Leslie Lamport's history of the paper (http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html#lamport-paxos)
5. Lamport, Leslie (May 1998). "The Part-Time Parliament". *ACM Transactions on Computer Systems*. **16** (2): 133–169. doi:10.1145/279227.279229. Retrieved 2007-02-02.
6. Fischer, M.; Lynch, N.; Paterson, M. (April 1985). "Impossibility of distributed consensus with one faulty process". *Journal of the ACM*. **32** (2): 374–382. doi:10.1145/3149.214121.
7. Lamport, Leslie; Massa, Mike (2004). *Cheap Paxos*. Proceedings of the International Conference on Dependable Systems and Networks (DSN 2004).

8. Lamport, Leslie (2005). "Fast Paxos".
9. Lamport, Leslie (2005). "Generalized Consensus and Paxos".
10. Castro, Miguel (2001). "Practical Byzantine Fault Tolerance" (PDF).
11. 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. 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. Birman, Kenneth; Joseph, Thomas (February 1987). "Reliable Communication in the Presence of Failures". *ACM Transactions on Computer Systems*. **5**: 47–76. doi:10.1145/7351.7478.
14. Lamport, Leslie; Malkhi, Dahlia; Zhou, Lidong (March 2010). "Reconfiguring a State Machine". *SIGACT News*. **41** (1): 63–73. doi:10.1145/1753171.1753191.
15. Lamport, Leslie (2004). "Lower Bounds for Asynchronous Consensus".
16. Chandra, Tushar; Griesemer, Robert; Redstone, Joshua (2007). "Paxos Made Live – An Engineering Perspective". *PODC '07: 26th ACM Symposium on Principles of Distributed Computing*.
17. Lamport, Leslie (2001). Paxos Made Simple (http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html#paxos-simple) *ACM SIGACT News (Distributed Computing Column) 32*, 4 (Whole Number 121, December 2001) 51-58.
18. Pierre, Sutra; Marc, Shapiro (2011). *Fast Genuine Generalized Consensus* (PDF). SRDS'11: 30th IEEE Symposium on Reliable Distributed Systems.
19. 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. Edward Lee and Chandramohan Thekkath. "Petal: Distributed Virtual Disks" (http://dl.acm.org/citation.cfm?id=237157). ASPLOS VII, 1996.
21. Aahlad et al. (2011). "The Distributed Coordination Engine (DConE)" (https://www.wandisco.com/get?f=documentation%2Fwhitepapers%2FWANdisco_DConE_White_Paper.pdf). WANdisco white paper.
22. Kolbeck, Björn; Högqvist, Mikael; Stender, Jan; Hupfeld, Felix (2011). "Flease - Lease Coordination without a Lock Server" (http://www.xtreemfs.org/publications/flease_paper_ipdps.pdf). 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2011).
23. Vermeulen, Al; Sivasubramanian, Swami (2014). "Under the Covers of AWS: Core Distributed Systems Primitives That Power Our Platform" (http://www.slideshare.net/AmazonWebServices/spot302-under-the-covers-of-aws-core-distributed-systems-primitives-that-power-our-platform-aws-reinvent-2014). AWS re:invent 2014.
24. "Oracle NoSQL Database 12.1.4.0.9, Community Edition Data Sheet" (http://www.oracle.com/technetwork/database/database-technologies/nosqldb/documentation/datasheet-oracle-nosql-db-ce-1876735.pdf). Oracle Data Sheet (2016).

External links

- erlpaxos, Paxos by Erlang (http://code.google.com/p/erlpaxos/)
- FTFile: Fault Tolerant File library (http://ftfile.rodia.pl/)
- Google Whitepaper: Bigtable A Distributed Storage System for Structured Data (http://research.google.com/archive/bigtable.html)
- Google Whitepaper: Chubby Distributed Lock Service (http://research.google.com/archive/chubby.html)
- HA Clustering with Neo4j (http://de.slideshare.net/jexp/new-neo4j-auto-ha-cluster)
- HT-Paxos (http://arxiv.org/abs/1407.1237)
- Vsync library (the SafeSend primitive is a free, open source implementation of Paxos) (http://vsync.codeplex.com)
- JBP - Java Byzantine Paxos (http://www.navigators.di.fc.ul.pt/software/jitt/jbp.html)
- Leslie Lamport's home page (http://www.lamport.org/)
- libpaxos, a collection of open source implementations of the Paxos algorithm (http://libpaxos.sourceforge.net/)
- libpaxos-cpp, a C++ implementation of the paxos distributed consensus algorithm (http://www.leonmergen.com/libpaxos-cpp/)
- Mencius - Circular rotating Paxos for geo-distributed systems (http://www.usenix.org/event/osdi08/tech/f

ull_papers/mao/mao.pdf)
- OpenReplica Open Replication Service (http://openreplica.org)
- paxos - Straight-forward paxos implementation in Python & Java (https://github.com/cocagne/paxos/)
- *Consensus on Transaction Commit (https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2003-96.pdf)*, Jim Gray and Leslie Lamport, January 1, 2004.
- *Paxos Made Simple (http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf)*, Leslie Lamport, November 1, 2001.
- Revisiting the Paxos Algorithm (http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.44.5607)
- Survey of Paxos Algorithms (2007) (http://www.fractalscape.org/2007/10/01/paxos-family.html)
- WANdisco - Active-Active Replication solutions for Hadoop, Subversion & GIT (http://www.wandisco.com/)
- PhxPaxos - C++ Paxos library that has been used in Wechat production environment (https://github.com/tencent-wechat/phxpaxos)