The Chubby Lock Service for Loosely-Coupled Distributed Systems

...and a little bit about Paxos too

Presenter: Xiao Shi

Yale University

xiao.shi@yale.edu

Oct 2015

Distributed Consensus and the Paxos Protocol

...and a little bit about Chubby Lock Service too

Presenter: Xiao Shi

Yale University

xiao.shi@yale.edu

Oct 2015

Overview

1. Consensus Problem
2. Paxos Algorithm
3. Chubby

Consensus Problem

Definition 1

Assume a collection of processes under a message passing framework that can propose initial values. Up to f processes may **fail**. Devise an algorithm or a protocol which lets each process decide on a final value and guarantees the following properties:

- **Agreement**: all non-faulty processes decide the same value;
- **Validity**: if all processes proposed the same initial value, that is the value eventually decided;
- **Termination**: all non-faulty processes eventually decide;
- **Non-triviality**: there exists executions A and B that produce different outputs.[a]

[a] Prevents trivial protocols that always returns 0 for example.

Variants of the Consensus Problem

Failure modes

- Crash failures (stopping failures): never restart
- Crash failures, but processes may restart
- Byzantine failures: all kinds of mischief...
- ...

Byzantine adversary

Frameworks

- Message passing
  - Synchronous (blocking): divided nicely into rounds
  - Asynchronous (non-blocking): messages may be delayed, lost, out of order, duplicated; but may **not** be corrupt
- Distributed Shared memory[a]
- ...

[a] never really worked well...

Proven Bounds and Results

Theorem 2 (Lower bound on the number of rounds)

In synchronous message passing frameworks, it requires at least $f+1$ rounds if f processes can fail (either crash failure or Byzantine failure).

Theorem 3 (Minimum number of processes for Byzantine agreement)

The total number of processes n must be at least $3f+1$ to tolerate a system with f Byzantine processes.

Theorem 4

Asynchronous agreement cannot be achieved with even one crash failure (without augmenting the model with clocks or randomization etc.).

Theorem 4 is known as the famous FLP (Impossibility) Result[1].

[1] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. "Impossibility of Distributed Consensus with One Faulty Process". In: *J. ACM* 32.2 (Apr. 1985), pp. 374–382. ISSN: 0004-5411. DOI: 10.1145/3149.214121. URL: http://doi.acm.org/10.1145/3149.214121.

Paxos[5]: A bit of History

- Previous work on consensus: after the FLP result in 1985, Dwork et al.[2] showed in 1988 that consensus is achievable under the assumption of partial synchrony[3].
- Viewtimestamped Replication[4] (1998) was a related paper addressing replication of DB transactions and maintenance of consistent views. The *view change protocol* bears a striking resemblance to Paxos. However, these two papers were developed completely independently.
- Lamport wrote a tech report describing Paxos in 1990, which did not get published until 1998.

[2] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. "Consensus in the presence of partial synchrony". In: *Journal of the ACM (JACM)* 35.2 (1988), pp. 288–323.

[3] Partial synchrony:
○ Of communication: let Δ be the upper bound message delivery time, assume processors clock rates are the same: either (1) Δ exists but unknown; or (2) Δ holds from some unknown time onward.
○ Of processes: let Φ be the upper bound on relative processor speed: either (3) Φ exists but unknown; or (4) Φ holds from some unknown time onward.

[4] Brian M Oki and Barbara H Liskov. "Viewstamped replication: A new primary copy method to support highly-available distributed systems". In: *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*. ACM. 1988, pp. 8–17.

[5] Leslie Lamport. "The Part-time Parliament". In: *ACM Trans. Comput. Syst.* 16.2 (May 1998), pp. 133–169. ISSN: 0734-2071. DOI: 10.1145/279227.279229. URL: http://doi.acm.org/10.1145/279227.279229.

Paxos: Model and Assumptions

- Processes
  - crash failures only; no Byzantine failures
  - (the processes with durable storage) may restart/re-join
- Network
  - asynchronous message passing framework
  - each pair of processes can ping/message each other
- Result: all non-faulty processes decide on **one** of the proposed values.
- Paxos has grown to a large family of protocols, and is understood to be one of the most efficient practical algorithms for distributed consensus.

Paxos: Development and Future

- Single decree vs. multi decree (multi-Paxos)
- Deployments and usage:
  - Google: Chubby, Spanner
  - Microsoft: Autopilot (data center management)
  - Amazon: AWS
  - Apache Zookeeper, VMWare, ...
- Alternatives:
  - Phase King algorithm[6] (synchronous message passing with Byzantine tolerance)
  - Chandra-Toueg consensus algorithm[7]
    - uses *eventually strong* failure detectors
    - structurally similar to Paxos
  - Raft[8]

[6] Piotr Berman and Juan A Garay. "Cloture Votes: n/4-resilient Distributed Consensus int+ 1 rounds". In: *Mathematical Systems Theory* 26.1 (1993), pp. 3–19.

[7] Tushar Deepak Chandra and Sam Toueg. "Unreliable Failure Detectors for Reliable Distributed Systems". In: *J. ACM* 43.2 (Mar. 1996), pp. 225–267. ISSN: 0004-5411. DOI: 10.1145/226643.226647. URL: http://doi.acm.org/10.1145/226643.226647.

[8] Diego Ongaro and John Ousterhout. "In search of an understandable consensus algorithm". In: *Proc. USENIX Annual Technical Conference*. 2014, pp. 305–320.

Paxos Protocol

- Processes are divided into three categories: **proposers**, **accepters**, and **learners**. In practice, a process can take on all three roles.
- High-level description:
  - **Proposers** attempt to ratify their proposed value by collecting *acceptances* from a *majority* of the **accepters**;
  - **Learners** "observe" whether any proposal has gained majority by asking **accepters** for their accepted values, and output the eventual decided value.
- **Accepters** accept values *locally*, i.e., the value that an accepter accepts may not be the value decided by the entire system, thus separating "individual choice" from "group decision."

Paxos Protocol: cont'd (1)

Requirement P1

An acceptor must accept the *first* proposal that it receives.

- In the absence of failure or message loss, we want a value to be chosen even if only one value is proposed by a single proposer.
- Possible "deadlock" scenarios:
  - multiple proposals, none gained majority;
  - one proposal won against another by one vote, then one process crashed;
- Therefore, in order to make progress, an accepter must be able to accept more than one proposal.
- $\Longrightarrow$ Impose total order on proposals using proposal numbers (tuples of process id or IP address AND timestamp).
- Each proposal (n, v) then consists of a proposal number n and a proposed value v.

Paxos Protocol: cont'd (2)

Paxos Phase 1

1. The proposer sends a message **prepare**(n) to all accepters, where n is the proposal number.
2. Each accepter compares n to the highest-numbered proposal for which it has responded to a **prepare** message. If n is greater, the accepter responds with **ack**(n, v, n_v), where v is the highest-numbered proposal it has accepted and n_v is the number of that proposal (or **ack**($n, null, null$) if the accepter has not accepted any prior proposals).[a]

[a] Think of the **ack** messages as a promise never to accept any proposal numbered less than n.

Paxos Protocol: cont'd (3)

Paxos Phase 2

3. The proposer waits (possibly forever) to receive **ack** from a majority of accepters. Among all the **ack**'s with a value, the proposer picks the value v_h with the highest proposal number. If none contained a value, the proposer picks its own proposed value. It then sends **accept**(n, v_h)[a] to all accepters.
4. Upon receiving **accept**(n, v), an accepter accepts v unless it has already received **prepare**(n') for some $n' > n$.

[a]Think of this as imperative–"Please accept!"–instead of acquiesce. Although, the accepters will still need to decide whether to accept.

Whenever an accepter accepts a value, it sends a message **accepted**(n, v) to all the learners. As soon as any learner sees that a proposal has been accepted by a majority of accepters, it outputs that value as the final decision of the system.

Much of the above can be optimized, e.g., by sending **nack** messages.

Paxos Protocol: cont'd (4) Safety Properties

Definition 5

A value is **chosen** if a single proposal with that value *has been* accepted by a majority of accepters. We also say that proposal has been **chosen**.

Can multiple proposals be chosen? Yes. Can they have different values? **No.**

Invariant $P2^b$

If a proposal (m, v) is chosen, then every higher-numbered proposal issued by any proposer has value v.

Invariant $P2^b$ implies that the Paxos protocol the first chosen value will be the decision value of the entire system and that no two distinct values are both chosen, which gives **agreement**.
Validity follows from that no value is chosen unless it is previously proposed.

Paxos Protocol: cont'd (5) Proof of Invariant $P2^b$

Invariant $P2^b$

If a proposal (m, v) is chosen, then every higher-numbered proposal issued by any proposer has value v.

Proof

This follows by induction on proposal numbers:

- Assume that proposal n is chosen, where $n > m$. Induction hypothesis: $\forall m \leq i < n$, the issued[a] proposal i has value v.
- Hence, there is a majority of accepters (call this set S) such that each accepter in S have accepted at least one proposal numbered $[m, n-1]$ with value v.
- For proposal n to be chosen, there must be some set C consisting of a majority of accepters such that each accepter in C sent back an **ack** message. Since S and C has at least one process in common (because of overlapping majority), proposal n will have value v.

Paxos Protocol: cont'd (6) Liveness Properties

- With a single proposer which survives long enough to receive **ack** and send out **accept** messages, the protocol will terminate in a few message delays.
- However, with multiple proposers, it is easy to imagine different proposers step on each other. The solution is to ensure that there is eventually some interval during which there is exactly one proposer who does not fail. The FLP result tells us that this is impossible unless we use randomization or real time (but **safety** is always guaranteed).
- Some solutions include exponential random backoff (as used by the Ethernet for congestion avoidance), the Ω failure detector, etc.

Paxos in Chubby[9]

- Chubby is a lock service with an API similar to that of a file system. It allows clients to perform *whole-file* reads and writes, augmented with advisory locks and notification of various events such as file modification.
- Expected usage:
  - coarse-grained synchronization within client systems, in particular, leader election (electing a primary among a set of otherwise equivalent servers)
  - well-known and available storage of a small amount of metadata
  - work partition between multiple servers
- Chubby needs leader election itself (among the 5 servers in a Chubby cell), and it uses Paxos for *master election*. The *master lease* is essentially a mechanism to designate one single proposer for a period of time, and thus guaranteeing progress.
- Chubby also relies on Paxos to propagate the writes among the replicas.

[9] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. "Paxos made live: an engineering perspective". In: *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. ACM. 2007, pp. 398–407.

Highlights of Chubby Design Choices

Lock service over Paxos library or consensus service

- Low programming complexity (for the clients...)
  - Easy to maintain existing program structures
  - Familiar API to developers
  - A single client can make progress safely using locks
- Mechanism to advertise election results, hence serving small files

Coarse-grained over fine-grained

- Weaker latency requirements (especially considering fail-overs)
- Far less load
- Enough for the clients, since they can easily implement their own fine-grained locks

Advisory locks over mandatory locks

Chubby Engineering Techniques

"Building Chubby was an engineering effort; ... [I]t was not research. ... The purpose of this paper is to describe what we did and why, rather than to advocate it." –Mike Burrows

Hiding complexity from client applications

In Chubby's case, the Chubby servers and its client libraries collaborate to provide the illusion to the application that no failure has occurred.

- *Jeopardy/safe/expired* events as well as grace periods during master fail-overs

Sequence numbering to handle delayed or reordered requests

- Chubby: sequencer (and lock-delay) in place of sequence numbers in *virtual synchrony*; epoch number for masters
- Paxos proposal number; also extensively used in network protocols such as TCP/IP

Chubby Engineering Techniques: cont'd

"We have found that the key to scaling Chubby is not server performance; reducing communication to the server can have far greater impact."

Scaling by reducing traffic and overhead

- Sessions and KeepAlives (with piggybacked events)
- Caching (including negative caching, i.e., caching failure results)
- Proxies for reads and KeepAlives, resembles a write-through cache
- Chubby permitting clients to cache locks–holding locks longer than strictly necessary

Other scaling techniques

- Balance client-server ratio: one Chubby cell per data center, etc.
- Partitioning: hierarchical namespace in a filesystem-like API comes in handy

Chubby: Lessons Learned

Unexpected use as a name server

- Chubby provides explicit invalidation for caching vs. time-based caching in DNS (TTL refreshes)
- Name resolution requires only timely notification rather than full consistency: to utilize this property, Chubby had additional implementations

Abuses

- Repeated open/close calls on files; not understanding RPC costs
- Some used Chubby as storage for large amounts of data
- Some attempted to use Chubby's event notification mechanism as publish/subscribe

Chubby: Lessons Learned (cont'd)

Use of UDP as transport protocol for RPCs

TCP (in particular, its back off policy) ignores the high-level timeouts such as the Chubby leases, so TCP based KeepAlives led to many lost sessions at times of high network congestion.

Distribution of engineering efforts

- Fail-over code, executed far less often than other parts of the system, has been a rich source of interesting bugs.
- Chubby rewrote a simpler version of Berkeley DB and used Paxos for replication because Berkeley DB's replication code is less frequently used and thus less well-maintained

Chubby: Rants

"Despite attempts at education, our developers regularly write loops that retry indefinitely when a file is not present, or poll a file by opening it and closing it repeatedly when one might expect they would open the file just once."

Chubby: Rants (cont'd)

On over-optimism of Chubby's availability:

"We find that our developers rarely think about failure probabilities, and are inclined to treat a service like Chubby as though it were always available."

"Developers also fail to appreciate the difference between a service being up, and that service being available to their applications."

Resources

- James Aspnes, Notes on Theory of Distributed Systems. http://www.cs.yale.edu/homes/aspnes/classes/465/notes.pdf
- Apache Zookeeper. https://zookeeper.apache.org/

The End