Good Math/Bad Math

The beauty of math; the humor of stupidity.

Paxos, a really beautiful protocol for distributed consensus

The work that I do in real life is all focused on tools for other developers. In todays environment, that means that I've spent a lot of time working on tools that, in one way or another, help other developers deal with distributed systems. In that work, I've noticed that there are some really key things that straddle the line between pure math and pure engineering. That's really interesting to someone like me!

A good example of that is something called *paxos*. My first exposure to paxos was very interesting. I'd just been hired by Google, and was working on their build tool. At the time, engineers in Google had a problem. Google's codebase was contained in one massive version control repository. Doing things that way has a ton of really neat advantages – most importantly, the fact that it makes it really convenient to reuse code written by anyone else at the company. The problem was that code reuse can become very confusing. Project A reuses a bit of code written by people from project B. B's code reused some stuff from C, and C from D, and D from E. So now project A is using code from project E, and they don't know why!

In this case, I had someone from a storage project coming to me trying to figure out just why his system had a dependency on a plan9 database system called paxos. I had to built a tool that would allow people to ask questions like "Why does A depend on E?".

As it turned out, paxos was a *really* important thing, and it was widely reused through the Google codebase. Once I learned about it, I started seeing it *everywhere*. Since I left Google nearly four years ago, I didn't stop seeing it. It's ubiquitous in distributed systems. Outside of Google, we weren't using that friendly old plan9 paxos implementation – but the paxos model has been reimplemented dozens of times, because it's so darned useful!

paxos is a system for managing *consensus*.

In distributed systems, there a collection of hard problems that you constantly need to deal with.

1. Things fail. You can *never* count on anything being reliable. Even if you have perfectly bug-free software, and hardware that never breaks, you've still got to deal with the fact that network connections

can break, or messages within a network can get lost, or that some bozo might sever your network connection with a bulldozer. (That really happened while I was at Google!)
2. Given (1), you can never rely on one copy of anything, because that copy might become unavailable due to a failure. So you need to keep multiple copies, and those copies need to be *consistent* – meaning that at any time, all of the copies agree about their contents.
3. There's no way to maintain a single completely consistent view of time between multiple computers. Due to inconsistencies in individual machine performance, and variable network delays, variable storage latency, and several other factors, there's no canonical way of saying that for two events X and Y, "X happened before Y". What that means is that when you try to maintain a consistent set of data, you can't just say "Run all of the events in order", because while one server maintaining one copy might "know" that X happened before Y, another server maintaining another copy might be just as certain that Y happened before X.

In a world where you can't count on different agents seeing events in the same order, and where you expect things to be constantly failing, how can you make sure that any distributed system you build ends up with a consistent view of reality?

The answer is a *consensus protocol*. You need to create a mechanism based on communication between the copies of your system that allows them to mantain a consistent *consensus* of what the current state of the world is, even in the presence of failures of machines, storage systems, and communications.

paxos is a very elegant, reasonably simple consensus protocol.

Let's get a bit more precise. Paxos is built on a model of storage. The original application of it was a consistent database, so it's built around the idea of keeping data consistent. In paxos, the state of the storage is modelled as a sequence of *transactions*. Each transaction is a pair *(t, v)*, where t is a numeric transaction identifier, and a v is a transaction value.

The state of the system being modelled is a sequence of transaction pairs, $[(t_i, v_i), (t_j, v_j), ..]$, where the *t* values are increasing as you progress through the sequence. As time passes, new transaction pairs can be added to the state.

The focus of the paxos protocol is ensuring that in a collection of $2n+1$ participants, all surviving participants will *agree* on the current value of the state, even if up to *n* participants fail, and even if messages can be delivered in arbitrary order.

Before I go further into the description of paxos, we need to look at the basic assumptions that underlie it. Like any formal model, it's not defined in terms of real computers. It's defined in terms of an abstraction that approximates reality. In this case, the approximation is quite good, but we still need to go through the basic assumptions that make up its model of the universe.

1. Processors (aka participants, servers, computers):
    1. operate at any speed. No two processors necessarily operate at the same speed.
    2. may fail without warning.
    3. may rejoin after recovering from a failure.
    4. are cooperative (in the sense that they do not attempt to cause failures).
2. Network:
    1. Delivers messages between any pair of processors.
    2. Transmits messages asynchronously.
    3. Delivers messages at arbitrary speeds.
    4. Does *not* guarantee that messages will be delivered in the order in which they were transmitted.
    5. Does guarantee that a message, if delivered, will be delivered correctly, without any changes.
    6. May fail to deliver a message.
    7. May deliver multiple copies of the same message.

In short, everything can fail at any time; after failure, participants can recover and rejoin the system; any no part of the system acts in an actively adversarial way.

The protocol describes the behavior of the system in terms of a collection of *roles*. A participant can have more than one role in the system – in fact, in most implementations of paxos, all partipants do have multiple roles. The roles are:

Client
The client is *not* part of the paxos cluster. It's an external entity whose actions trigger state changes by making *requests* to the paxos system. Each state update in paxos is initiated by a client request, and completed by a reply to the client.

Acceptor
An acceptor (also called a *voter*) is a participant in the maintanence of distributed storage. A state change in a paxos cluster does not occur until a majority (quorum) of acceptors agree upon it.

Proposer
A proposer recieves a request from the client, and attempts to get a quorum of acceptors to agree on it.

Leader
One of the proposers is special. It is the single proposer who most recently had a proposal accepted. In many paxos implementations, there is only one active proposer serving client requests: the only time the other proposers send proposals is when the current leader fails, and a new one needs to be selected.

Learner

The learner is the real service provided by the paxos cluster. Once a proposal is accepted, a *learner* processes the request from the client, and sends it the result.

In a typical paxos cluster, the client sends requests to a proposer. The proposer sends a proposal to update the state with the new client request, and attempts to convince a majority of the acceptors to accept it. Once a majority accepts it, the client request is processed by the learner, and a result is returned to the client.

The meat of paxos the protocol that the proposer gets a majority of acceptors to agree on a proposal, and how that protocol process ensures that the collection of acceptors maintains a consistent state.

The protocol itself is pretty simple. Each *round* is effectively independent, and consists of a process of attempting to reach consensus. Within each round, finding consensus is a two-phase process, where each phase consists of a message sent from a proposer to a group of acceptors, and a reply from the acceptors to the proposer.

1. **Phase One: Prepare/Promise**
    - **Proposer:** A proposer attempts to start setting a new consensus by sending a *Prepare(N)* message to a *quorum* of acceptors. It can send to *any* group of acceptors, so long as that group forms a majority of the acceptors. The prepare message specifies a numeric identifier N for its proposal, which is larger than any proposal that's been sent by this proposer.
    - **Acceptors:**
      Each acceptor, upon receiving the proposal, checks if the N-value from the prepare message is greater than any proposal from the current round that it has accepted. If so, it sends a reply called a *Promise* to the proposer, promising that it will never accept any proposal with a number *less that* N. If the acceptor has accepted a proposal with number less than N in the current round, then it includes the pair (v, n_v) consisting of the proposed consensus value v and the number n_v of the accepted proposal that proposed v.
      The acceptor thus sends a message *Promise(N, (v, n_v))* (if it has accepted a proposal this round) or *Promise(N, null)* (if it has not yet accepted a proposal with number less than N).

      Once it's sent a promise message, it *must not* accept any request for a proposal with number less that N. Note though that this does *not* mean that the acceptor promises to accept the proposal: all it's doing is promising *not* to accept any proposal with number less than N! If in receives a message *Prepare(N+1)*, it's free to promise that – but if it does, it will no longer be able to accept the proposal for N.

      (If *N* is smaller that the number of any proposal promised or accepted by the acceptor, then in the original version of paxos, the acceptor does nothing; in some optimizations of the protocol, it replies *Reject(n_v)*.)

What this phase does is allow a proposer to determine whether or not a new proposal is even worth

considering. If a quorum (majority) of acceptors send promises, then it can move on to phase 2.

2. **Phase Two: Accept!/Accepted**
   When a proposer recieves promises from a quorum of acceptors, then it moves forward to try to actually commit the proposal. In order to do this, it needs to choose a *value* for the proposal. If any of the *Promise* messages contained a value, then the value of this proposal must be set to the value of the *highest* proposal number in any of the promises. If all of the promises were empty, then the proposer can choose any value that it wants for the proposal.

   Once the proposer has chosen a value, then it sends a message *Accept!(N, V)* to a quorum of acceptors. This is typically written with the exclamation point, because it's really a *command* to the acceptors: they're being told to accept the proposal, if they can.

   When an acceptor receives an *Accept!(N, v)* message, if it has not issued a promise for a proposal with number greater than N, then it *must* accept the message. It accepts the proposal by sending a message *Accepted(N, V)* to both the original proposer, and all of the learners.

   When *Accepted* messages have been received from a quorum of acceptors, the new value *V* becomes the *consensus* value for the paxos cluster, and the new proposal number N is fully committed.

As with so many things, this is easier to understand when you think about an example. One use of paxos that I've worked with is in a cluster scheduling service. In that system:

- a *client* is a user attempting to run a new job on the cluster. It sends a request to the scheduler detailing the set of resources that it wants to request.
- Each duplicate of the scheduler is a *proposer*, an acceptor, *and* a learner. There's one active instance of the scheduler, which is the *leader*. When a client wants to schedule a job, its request gets sent to the leading scheduler.
- In the normal non-error case, this works as follows:
  1. When a scheduling request is received, the leader proposes scheduling the job, by sending a message to all of the other schedulers saying that it wants to schedule job N.
  2. The other schedulers, if they haven't seen a proposal for a job with number greater than i, make promises to accept that proposal.
  3. The leading scheduler chooses resources for the job, and then sends an *Accept!* message to the other schedulers.
  4. The other schedulers reply accepting the scheduling. The non-leader schedulers, acting as learnings, record the scheduling information, and the leader actually starts the job.
- Errors occur when there was some kind of failure. In that case, we don't necessarily know who the leader is – so we get multiple schedulers trying to act as if they're the leader. So they each send proposals. Whichever proposal had the largest proposal number will eventually get accepted, and its proposer becomes the new leader.

It's a pretty simple thing – the core concept is simply that no consensus proposal is considered "committed" until it's been accepted by a majority of the participants. And if it's been accepted by a majority of the participants, that means that no conflicting proposal can ever reach consensus – because that would require at least one participant to accept 2 conflicting proposals.

But there's still a bit of formality that's working looking at. Exactly what guarantees does paxos give? What properties does paxos-style consensus have?

Even the formal properties of paxos are easy to understand. Paxos provides two key properties: validity, and agreement.

Validity
No value ever reaches consensus without first being proposed, and having its proposal accepted.

Agreement
No two distinct values ever reach consensus at the same time.

You an easily prove those two properties. In fact, the proof is completely obvious once you recognize that the paxos protocol has two invariants (and those invariants are themselves clear from the definition of the protocol!):

1. An acceptor can only accept a proposal p if and only if it has not yet made a promise for a proposal n where $n >= p$ is the consensus value of the highest numbered proposal that has been accepted before this proposal.

Getting back to the beginning: the point of all of this is to have a system in which we can be sure that things work correctly even in the presence of failures. In paxos, as long as at some point there was a quorum of machines that come to agreement, then any failure that leaves a surviving quorum of machines must have overlapped with the previous quorum – which means that the previous consensus still remains in effect, and will be propagated to the remaining participants. If you've got 5 machines, then two can fail, and you won't lose consistency among the remaining ones.

This entry was posted in Distributed Systems, Programming on January 30, 2015 [http://www.goodmath.org/blog/2015/01/30/paxos-a-really-beautiful-protocol-for-distributed-consensus/] .

---

4 thoughts on "Paxos, a really beautiful protocol for distributed consensus"