CMPSCI 377 Operating Systems **Fall 2009**

Lecture 17

Lecturer: Emery Berger *Scribe: Alexandre Passos*

17.1 Byzantine faults, Paxos, and consensus

Today's papers are on the byzantine generals problem and Paxos.

17.1.1 Byzantine faults

Some of you didn't have that much context, but some did. The byzantine generals problem is an old issue about what to do when you have malicious actors (which are not exactly simple failures). Byzantine faults are the most worst-case assumption possible in fault-tolerance. Think of RAID, where you never assume that a disk drive can be evil. If you design a system that is resistant to byzantine failure it's resistant to anything. There's no situation worse than having everybody conspiring against you.

In practice you need an awful lot of good guys to compensate for bad guys. There's another paper, FLP, on the impossibility of consensus, that argues that if you have bad actors around you can't even agree on the value of a single bit. The consensus problem is when you have a bunch of distributed peers that need to agree on a single bit.

17.1.2 Issues with the byzantine fault model

One practical point is that you need $3m + 1$ servers to handle m byzantine failures. However, you never know what the value of m can be. You can never predict how many of your generals can go south, or east, or whatever. There are some efficiency issues. Charlie argues that because everybody has to talk to everybody else you can't really scale. Charlie also argues that you need to constrain the threat model, maybe to handle evil actors who cannot collude.

Regarding the completely adversarial BFT, randomness is really the nicest possible fault model. It's always easy to get rid of random noise or errors. Evil can mean absolutely anything. One of the most evil things which can be done is arbitrarily dropping messages.

Alexandre argues that the adversarial model is too strong, and leads to tinfoil hat-style thinking.

17.1.3 Tolerating BFTs in practice

There's been a lot of recent work in byzantine fault tolerance. There are all sorts of fault models. If you can make a system byzantine fault tolerant, you can make it anything fault tolerant. Emery knows some people who work in this area, and it's a quite small area. Barbara Liskov, Miguel Castro, Lorzenzl Alvizi, and Mike Dalin. At every SOSP or ¡other conference¿ there's always a session on byzantine fault tolerance. Some people in this department discovered a way of handling byzantine fault tolerance with less good guys, which are only activated when something bad is suspected to have happened. Then you can outnumber the bad

guys, and handles byzantine fault with much less overhead. This changes the necessary number of systems from $3m + 1$ to $2m + 1$. They use content-addressed pages, sharing pages across VMs, to make these extra replicas really cheap.

The two challenges are (1) replicas (the number is very high, with a lot of good guys doing a lot of useless work, as the common case is not failure. You pay a huge cost mostly for nothing), (2) communication overhead. These protocols have a lot of communication.

17.1.4 Paxos

Paxos, oddly enough, has become fairly popular. People have invested a lot of engineering effort reducing the overhead of paxos by mostly piggybacking, and amortizing the cost of paxos' messages by sending these messages on top of other messages. A bunch of big-name companies are using Paxos, but they actually use Fast-Paxos, which eliminate most of the overhead. You no longer have to go through rounds, and piggybacking more than one round and role at once.

There is publicly available software called Zookeeper, from Yahoo!, which a Paxos-based scheme to provide consensus.

There's a question: if we can reduce the overhead, why not use it all the time? Even if you could get it down to twice the replicas, it's still a factor of two, which no one is willing to pay on the common case. Some tried to sell Amazon the idea of byzantine fault tolerance. Amazon has a big firewall, and all things which you ship to them sit on a virtual machine. When you shut down VMs things are gone, and you can't have any effects, modulo crazy VM leakage bugs. In google there's also a very private system which lets you host code on their servers.

17.1.5 Peer-to-peer distribution and faults

So how can byzantine faults happen? What are these things used for? Peer-to-peer? I'm sure you're all familiar with peer to peer networks, and most have probably used something like bittorrent to download Linux or something similar (as far as I know).

So what's the value of BFT in p2p? Charlie: anyone can connect and everybody acts as a server, so adversarial people can have a big impact. Bittorrent is an interesting system. Bittorrent is from a 30,000ft level a distributed hash table. It's a huge set of blobs indexed by a hash of their contents. I then go look for all the blobs that make up a file. With bittorrent I try to obtain all these blobs at the same time, to take advantage of extra bandwidth. If everybody's good then things are really nice. It's absolutely trivial to set up an evil tracker. You go into a website searching for these trackers. Anybody could make up their own trackers to replace true files with bogus things, but hashes provide some form of protection against this. The most common attack is clicking on a file and after downloading it discovering that it only links you to an evil website to download some software. This is called poisoning.

What's the difference between Napster and bittorrent? Napster is totally centralized. Everybody connects to the server, posts a list of their files, which would be hosted on the server. When someone searches for a file the server just gives them these tokens. Speed is much worse in Napster, as you only connect to a single peer at a time, and also you have no fault tolerance. We can easily come up with ways of engineering ways to fix Napster.

17.1.6 Peer-to-peer versus centralized architectures

Why don't people do this? Because it's illegal. There's no technical reason why Napster is inferior to bittorrent, and from a discovery point of view it's way better. In bittorrent you can look for, say, Ubuntu, and you have to go over loads of separate servers. In Napster, after searching for ubuntu, you get a big list "instantly" of files, while in bittorrent you have to find some out-of-system way to search.

There are peer-to-peer search engines, which are slow enough to be unusable. A centralized service such as Napster to download music would be much better.

iTunes is actually horrible. Charlie says: it's Safari. Emery: iTunes is a frontend over safari, but what do they have in their web servers which is so slow? He downloaded a map, a gigabyte, and it took hours. Amazon seems to know how to build infrastructure, and their music download service is almost instantaneous. Amazon has a cloud service (as has google now), and the way it works is that anything you buy now gets automatically stored in their servers. Their latency is about the same as itunes, which keeps everything in your computer.

Engineering solutions to make reliable high-bandwidth high-availability services exist. The motivation for these distributed systems was a legal one, not a technical one.

In this peer-to-peer system poisoning happens because of the $3m + 1$ problem. What is m? You don't know. So you can look for, say, the next version of Ubuntu, and you find that all the copies you can find are fake.

17.1.7 Paxos's transient faults versus Byzantine faults

If your adversary is a transient fault simple replication and voting solves your problem. Let's go to paxos. Everybody gave pretty comprehensive reviews.

Nobody is using BFT. It's hard to see where it's going to be useful.

One situation where you might want to tolerate byzantine faults is with the routers on the internet and BGP. The internet is sadly insecure, and works on the honor system. BFT however is not a generic solution. It's too powerful and too costly. It'd be easier to reengineer BGP to be trust-aware than to use a generic BFT solution.

All datacenters are designed to withstand transient faults. Everything has timeouts. It's impossible to tell whether someone is slow or dead. It's often the case where things will fail to respond to a timeout and then reappear. Most datacenters use majority voting; as long as the servers have data, voting is fine. BFT is overcome.

17.1.8 Paxos in practice

Paxos is in a sense sort of like RSA. RSA is expensive, so you use it as a wrapper to a cheaper protocol. With Paxos you elect a leader to be in charge of a large-scale event. The coordinator is a single point of failure, so you can use paxos to come up with a new coordinator. Most of the time you don't run paxos, you only run it when something bad happens to prevent any one thing to be a single point of failure.

John Osterhout came up with this thing called RAM Cloud. YOu have a cloud system but you don't want to store data on disk. All your data is only replicated in memory and it never needs to be written to disk (as long as you always deal with power failure).

Many datacenters have a hierarchical structure, using batteries, flywheels, and then finally generators. It

was used in formula one cars. Failures of flywheels are scary, which is why you should aim your datacenters at you competitors (joke).

RAMCloud is a log-structured filesystem. These are file systems which never do syncs for writing and can keep the most recent updates in the log as well. You also need to do garbage-collection to get rid of stale blocks. RAMCloud does the same thing in RAM. It's very clever. However, all communication has to go through one coordinator to find the peers. Single-point-of-failure is not usually a good thing, but you can use Zookeeper to elect new coordinators.