CMPSCI 630 Systems **Fall 2019**

Lecture 21

Lecturer: Emery Berger *Scribe: Vincent Pun, Wenjun Huang*

21.1 Property-based Testing

Traditional fuzzing is like testing with an oracle. If the program doesn't crash, then it's good; if the program crashes, then it's bad. But this oracle is imperfect and is not always available, so we would like to perform testing with a spec instead of with an oracle.

A full spec would usually be impractical, so we settle for something that's between an oracle and a spec. In property-based testing, we have some basic spec (e.g. $x + 0 = x$ and $x + y = y + x$) and we do fuzzing based on that.

Another approach, which is arguably not as good, is differential testing. In differential testing, we compare multiple implementations of the same thing (e.g. different browsers and different compilers) and find bugs.

21.2 Distributed Systems

Advantages of distributed systems:

- geographical locality (geo-colocation): reduced latency
- scale: more computational power
- fault-tolerance: no single point of failure (helps scaling)

Distributed systems are usually not fully connected, which requires $n(n-1)$ links and doesn't scale. Achieving scalability should normally be on the order of n, or even $\log n$.

To get fault-tolerance, we need (consistent) replication. When network links fail in a way that there is a complete split between nodes, we get *network partition*. Under this condition, there is no way to keep states in sync.

Add in availability, this leads to the CAP theorem, which states that you can only get two of [consistency, availability, parition tolerance], but not all three of them.

Consistency here means the replicas are in sync with each other. Consistency can be strong or eventual. Eventual consistency means that replication happens in an asynchronous manner. This might sound undesirable, but for some domains (e.g. FB likes) this is fine. There are domains where consistency is crucial (e.g. Google Doc), but there is no generic solution to this problem.

21.3 Consensus

In the early days, before distributed systems became a thing, the consensus problem is phrased as follows. There is a group of nodes in a system, and each of them picks a random binary value. Consensus is reached if (1) the system terminates (2) the nodes agree on a value (3) the final value is valid.

This is easy to do synchronously via taking a majority vote for some number of rounds: you would know which nodes have crashed if they went missing after a round. However, doing it asynchronously is hard, as there's no upper bound on message delivery time, and you don't know which nodes have crashed.

People have come up with various ways to know the unknown:

- timeouts: protocols must support the case where nodes rejoin the system later
- leader: leader can crash and is a single point of failure
- Paxos: hard to understand/implement
- Raft: easier than Paxos, but still not trivial

21.3.1 Raft

Raft separates Paxios into separate components. It regularly elects leaders and detects crashes using timeouts. Only servers with the most up-to-date log are eligible to become a leader.

When a message comes from a client to the leader, the leader appends to the logs and they are replicated to the servers. If the leader crashes during this time, future leaders can replay the logs.

21.4 Byzantine Faults

First, there is the Byzantine Generals problem. Suppose there is a group of generals who must agree upon a time of attack by sending messages back and forth. Additionally, any one of them might be a traitor, so messages might be randomly forged/changed/dropped/delayed. How do you reach consensus in such cases? If servers in a distributed system start acting like Byzantine generals, then you have Byzantine faults.

If all servers are Byzantine, there is no way to reach consensus. This situation is rather bleak, and this is the strongest fault model. Any system resistant to this fault will be able to resist anything, because this is the strongest adversary. If a limited portion of the servers are Byzantine, then it might still be possible for you to get Byzantine fault tolerance (BFT): Lamport shows that $3n + 1$ servers are needed to tolerate n faults. This number can be lowered to $2n + 1$ if the messages are encrypted, so no messages can be forged, or if there are suspended clones of servers such that if a Byzantine fault is detected it can just wake up the clones.

However, BFT isn't really useful in practice: if an adversary is able to take down $\frac{1}{3}$ of the servers in a system, what's stopping the adversary from compromising the other $\frac{2}{3}$?

21.5 Bitcoin

- it doesn't solve any real problem

- it's not anonymous
- thanks to its "proof of work" mechanism, it uses power inefficiently
- it is decentralized
- it runs true transactions in a decentralized setting

Bitcoin works by having all nodes build logs. Only the longest log is accepted as truth. A SHA hash of data to be added to the log is to be computed and added to the log based on the previous block in the log (hence the term *blockchain*). The incentive is there is bitcoin rewarded for each successful addition to the log.

Bitcoin was intended to support micro-transactions. The problem with micro-transactions is that the transaction costs might be higher than the transaction itself (e.g. the bank charges you 1 dollar for sending 1 cent). Obviously, Bitcoin doesn't support this usage very well right now, and the problem remains unsolved.

There are Bitcoin mining groups that split the rewards based on people's share of work, but how do you know that someone has actually made contributions? Bitcoin's difficulty is based on the number of leading zeros in the computed hash. To (probabilistically) measure the compute power that people have contributed, the number of leading zeros in their "best" hashes can serve as a proxy of their contribution.