Lecture 33 - Byzantine Fault Tolerance

Videos

https://youtu.be/wItXpoClurc - Lect-33-4010-BFT-pt1.mp4
https://youtu.be/_bhabm9Iy5k - Lect-33-4010-BFT-pt2.mp4
https://youtu.be/hbnVxTb8zoo - Lect-33-4010-pt3-BFT.mp4
https://youtu.be/7_2yrIOFm-Q - Lect-33-4010-pt4.mp4

From Amazon S3 - for download (same as youtube videos)

http://uw-s20-2015.s3.amazonaws.com/Lect-33-4010-BFT-pt1.mp4
http://uw-s20-2015.s3.amazonaws.com/Lect-33-4010-BFT-pt2.mp4
http://uw-s20-2015.s3.amazonaws.com/Lect-33-4010-pt3-BFT.mp4
http://uw-s20-2015.s3.amazonaws.com/Lect-33-4010-pt4.mp4

Papers

Original paper by Leslie Lamport, Robert Shostak and Marshall Pease, 1982:
https://people.eecs.berkeley.edu/~luca/cs174/byzantine.pdf

Practical Byzantine Fault Tolerance and Proactive Recovery" by Miguel Castro and Barbara Liskov:
http://www.pmg.csail.mit.edu/papers/bft-tocs.pdf

Honney Badger BFT, 2016: https://eprint.iacr.org/2016/199.pdf

Go Implementation: https://github.com/anthdm/hbbft

What is Byzantine Fault Tollerance

A system for recovery and data distribution with worst possible failures and communication failure.

1. Nodes can fail.
2. Nodes can be malicious.
3. Communication can fail.
4. Communication can be malicious.

What can go wrong?

1. Crash - or worst - Crash - recover - Crash - recover ....

2. Corrupted data - data rot - disk failure - database error.
3. Service is gone - You depend on a google service that was deprecated.
4. User - query of death - A user - legitimate runs a query that is bad, really really bad.
5. Denial of Service Attack - malicious.
6. Lots of people like you - crushed by load (Flash Crowd, Viral etc)
7. A server went down - cascade failure
8. Fails for a customer but not you
9. Lost data - it's gone
10. Security Breach
11. Network Failures - Google reports that 99% of database failures are really network failure.
12. The Back Hoe!
13. The Squirl!
14. Fire / Flood / Earthquake / Volcano.
15. Your system has caused a fire
16. Data center raided by FBI - "Hackers Game".
17. Hacked node - Yep -had that.
18. Non-Hack - then - system administrator intended to run `rm –rf ./dev` instead ran `rm –rf /dev` .
19. Runaway automation – Example $400 million error
20. This message `Got server HTTP error: HTTP Error 404: Not Found. Retrying fragment 7 (attempt 1 of 10)...`
21. Certificate Expire - They only last for a limited amount of time! Let's Encrypt and ACME protocol.
22. Power is off - network is off - etc.
23. Shift in workload....
24. Index in database got dropped (ran out of space etc.)
25. Personal change (person quit, died etc.)
26. System commits suicide - License Key Timeout.
27. Missing source - You think all is well but you find out that the version of the code you have is out of date and you DON'T have the current version.
28. Windows leeks memory - and it crashes - Expedia.
29. Random variation in ... Power, Load, etc.
30. Library you are using - has a flaw!
31. Network is just slow.
32. Email crashes entire network - England's NHS - and "Reply All".

Things that are in-frequent are not worth automation. Things that happen often are. Look at a ROI model - and how much real costs are. Take labor time 1.53 for burdened then look at amount of down time.

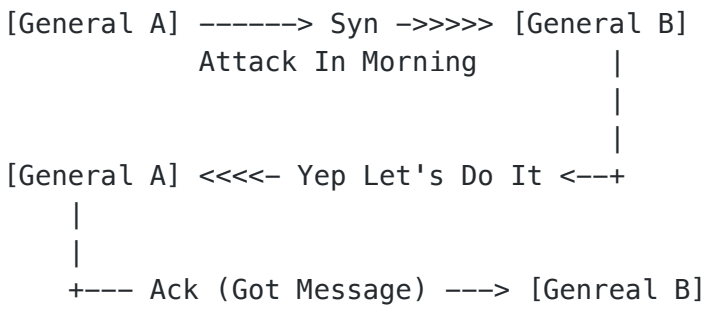
Generally 2 different kind of failures:

1. Stop Failure: When a node fails - it might have a redundant node and recover but the point is that the node stooped for a while. This could also be just a communication failure. You really can't tell the difference between a node failing and the nodes communication failing.
2. Now all the other failures - nodes doing what is not correct (traitor) - sends message A to one peer, B to another peer. This may not be a malicious failure. It could be just an error in transmission. It could also be that you have a malicious node operator - or - a hacked node.

Two Generals Problem

1. 2 generals with armies.
2. Opponent is larger than each of the 2 armies but smaller than the combined 2 armies.
3. Incentive to cheat.

Come up with a solution (Like TCP/IP Syn / SynAck / Ack)



With the network as the "opponent" there is no way to solve this. *General B* never knows until a successful `Ack` what state he should be in.

There is no way to get consensus.

Let's reverse this and let's assume that the nodes are the corrupting element.

Lamport's '82 paper has a solution. Very Ugly - Very Slow. Speed is `O(n^(f+1))`. Puts proven bounds on number of bad nodes and what can be done to solve this.

Is this valuable:

1. Boeing 767, 777 both use a solution for fly by wire.
2. Blockchains/Cryptocurrency is using this.
3. Data distribution over a faulty network.
4. IBM Hyperledger (Merk, IBM Food Trust etc).
5. Zilliqa uses an optimized version pBFT with a PoW round every ~100 blocks.
6. Ripple (XRP) uses a modified pBFT solution.
7. MasterCard's internal blockchain (5000 TPS) uses a pBFT

One of the big keys to this is getting the Assumptions correct. Clearly not solvable with just 2 nodes. What about with more nodes? What failure cases are you going to ignore and what cases are you going to program for?

In pBFT - it turns out that half+1 of the nodes agreeing is a "consensus" - in the generals case it is either attack-or-wait. This will not mean that "attack" is the true/correct result or that wait is the true/correct result. What it meas is that the system has reached a "consensus" decisions. To the half+1 you need more than 2 nodes. Let's talk about 5 nodes, A..F.

Multi Generals and Consensus

After Consensus:

```
General A         - A D D A A
General B         - A D D A A
General C         - D A A A A
General D         - A D D A A
General E         - A A D A A
                    ^ <<<<<<<<<< This one!
```

Communicate 1 decision from 1 general to all the all the other generals. Solves 1 communication.

How many traitors can we deal with?

Trivial Case - 1 general.

2 - general. If 1 traitor, then no conses.

Leader general says Attack, then how if you as a non-leader know. You could ask your piers. Walk through cases. Ask Piers

1. Case with peer as a traitor.
2. Case with leader as a traitor.

No Solution w/ 3 generals and 1 traitor case.

How many generals to tolerate 1 traitor. Generals = 3, 1 traitor. 3m+1. – Or look at it as you have to have less than $1/3$ traitors.

Proof: Assume that a solution exists. 12 Generals 4 Traitors. Simulate each one. Determine that you don't have a distinct answer. The lack of a unique solution means that you can't have this many. Simulate with 12 and 5 ... same result. Simulate with all cases with 12 and 3 - you can get a unique solution.

Algorithm that works:

1. 0 Traitors - then everybody shares - all agree - all good
2. M Traitors - less than $1/3$ - sends order. Record order. Add up all the total number of orders - and when you have a set of answers. Use majority.

Demo of this with a traitor.

2 traitors means you will need 7 generals.

m = number of traitors.

Traitors	Order
m = 0	0
m = 1	n^2
m = 2	n^3
m = 3	n^4

messages.

In a practical sense then Ethereum with 11401 nodes as of today, you have to a lot of traitors (11401 devided by 3) to effect the system. Since each node costs thousands (average of 18204.00) of dollars and effective attack on the system would cost `3800 * 18k` or about `$70` million in hardware to attack the system.

Practical Byzantine Fault Tolerance

For pBFT to work, we assume that the amount of malicious (lying) nodes in the network cannot simultaneously equal or exceed $1/3$ of the total number of nodes during any one time of vulnerability. The larger the number of nodes in the system the more robust the system. With `n` nodes as long as

at most (n-($\frac{1}{3}$)) are malicious or faulty at the same time the pBFT algorithm provides both speed and safety. Eventual consensus of the replies is achieved due to linearizability.

Each round of pBFT consensus (called views) is in 4 distinct phases. This model follows more of a "Commander and Lieutenant" system than a pure Byzantine Generals' Problem, where all generals are equal, due to the presence of a leader node. The phases are below.

1. A client sends a request to the leader node to invoke a service operation.
2. The leader node uses "hear-say" to multicasts the request to the other nodes.
3. The nodes execute the request and then send a reply to the client.
4. The client awaits f + 1 (f represents the maximum number of nodes that may be faulty) replies from different nodes with the same result. This result is the result of the operation.

The requirements for the nodes are that they are deterministic and start in the same state. The final result is that all honest nodes come to an agreement on the order of the record and they either accept it or reject it.

The leader node can be changed in a round robin type format during every view and can even be replaced with a protocol called view change if a specific amount of time has passed without the leader node multicasting the request. A majority of honest nodes can also decide whether a leader is faulty and remove them with the next leader in line as the replacement.

The primary advantages of the pBFT model is its ability to provide transaction finality without the need for confirmations like in Proof-of-Work models. If a proposed block is agreed upon by the nodes in a pBFT system, then that block is final. This addresses a lot of the energy usage concerns of Eth/BTC.

The model only works well with small consensus group sizes and has a large amount of communication overhead. The paper mentions using digital signatures and MACs (Method Authentication Codes) for authenticating messages. This is extremely inefficient and will not scale. Also you can't prove the authenticity of the messages. Digital signatures are a better solution.

Copyright