

# Byzantine Fault Tolerant

Andrea Merlina



UiO : Universitetet i Oslo

# What is a Byzantine Fault

A **Byzantine fault** is a condition of a distributed computing system where **components may fail** and there is **imperfect information** on whether a component has failed. [x]

## Considerations:

- It's the worse kind of failure
- A component might behave faulty for somebody and perfectly fine with somebody else
- You can think of a Byzantine component like malicious (and as smart as you want)

[x] [https://en.wikipedia.org/wiki/Byzantine\\_fault](https://en.wikipedia.org/wiki/Byzantine_fault)

# What is the meaning of BFT

Byzantine Fault Tolerance (BFT) is the ability of a network to unmistakably **reach a consensus despite the attempts of malicious nodes** to propagate false data to other peers. [x]

[x] <https://en.bitcoinwiki.org/wiki/PBFT>

# About consensus

Consensus is the problem of **voting and agreeing on a *single* value** by a group of independent entities

Three properties should be respected:

- **Termination:** All non-faulty processes eventually decide on a value
- **Agreement:** All processes that decide do so on the same value
- **Validity:** The value that has been decided must have proposed by some process
  - E.g. always agreeing on “No” is not a valid solution

# Ambiguity about consensus definition



Emin Gün Sirer

@el33th4xor



This is incorrect. PoS is not a consensus protocol. Nor is dPoS.  
There's no such thing as dBFT, but if there was, it wouldn't  
uniquely define a consensus protocol.

[cointelegraph.com/news/from-pos-...](https://cointelegraph.com/news/from-pos-...)

♡ 43 9:51 PM - Sep 13, 2018



From PoS to dBFT: A Brief Review of Consensus Protocols

Pros and cons of every major PoS challenger.

[cointelegraph.com](https://cointelegraph.com)

↪ 18 people are talking about this



Emin Gün Sirer

@el33th4xor



Ok, there is a terribly wrong framework emerging around  
consensus protocols. People think that PoW and PoS are  
consensus protocols, and that they are the only two consensus  
protocols out there.

This is false. Let me explain.

♡ 1,274 5:09 PM - Jun 13, 2018



↪ 527 people are talking about this



D1. Are PoS, PoW, etc. consensus algorithms?  
What do you think?

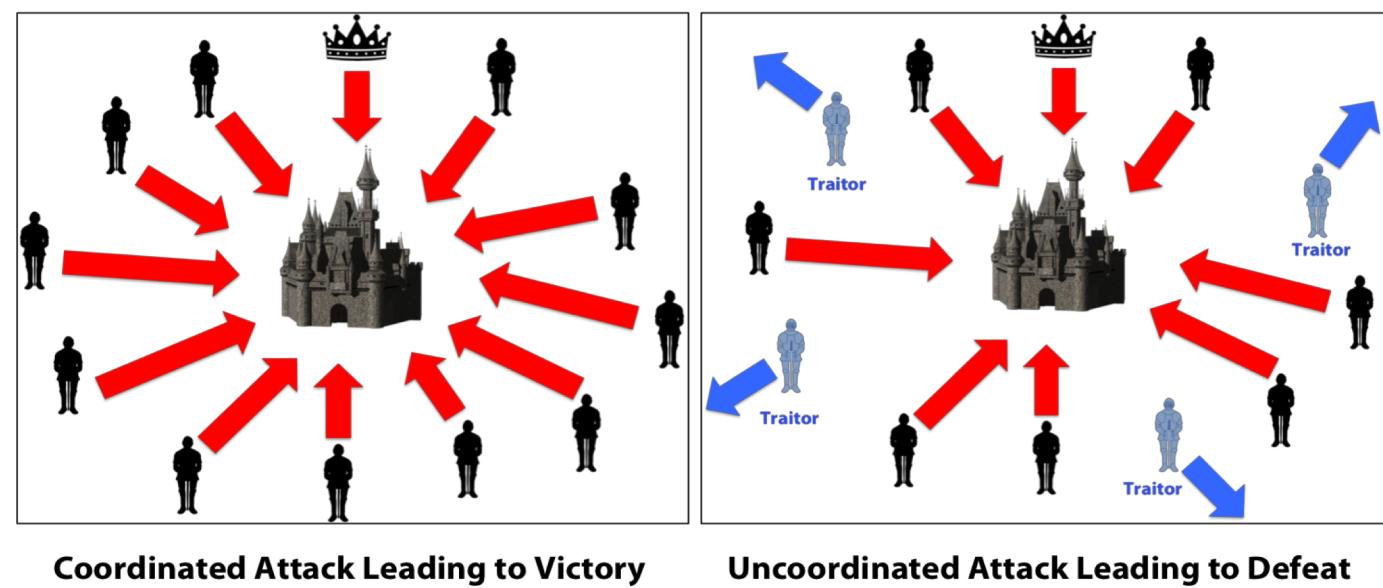
# Applying consensus to war: the general problem

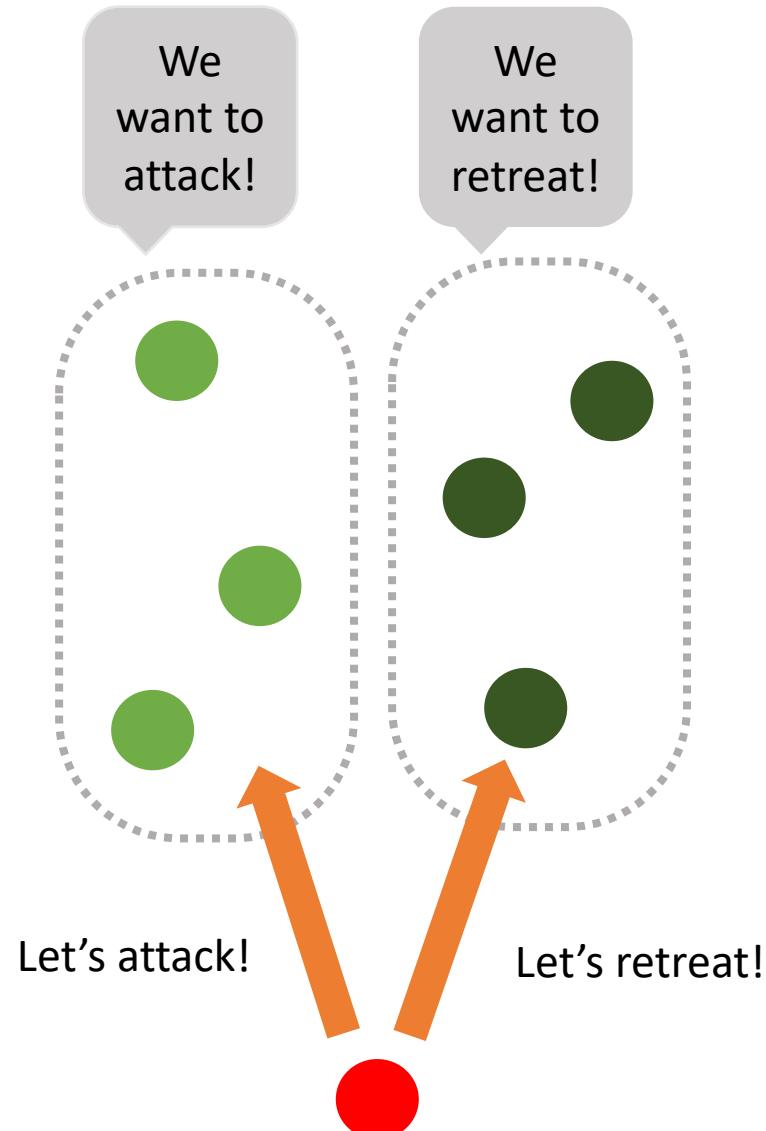
The Byzantine Empire is in decline and any of the generals might be a traitor, interested in the defeat of Byzantine. Two options: attack or retreat

Possible outcomes of the battle:

- If all the faithful generals attack, they will destroy the enemy (favorable outcome)
- If all the faithful generals retreat, they will retain the army (neutral outcome)
- If some loyal generals attack, and some retreat, the enemy will destroy their entire army (unfavorable outcome)

<https://en.bitcoinwiki.org/wiki/PBFT>





## Example

- Let's suppose **there is a tie**  
3 loyal generals want to attack while  
3 others (loyal but with different  
opinion) want to retreat
- The Byzantine node could **fake support**  
to the ones wanting to attack
- He could do the same to the ones  
wanting to retreat
- **6 loyal generals would be tricked into**  
**thinking to have reached a majority**
- **Just one** Byzantine node could lead to  
the worse case scenario

# Approaches

- Naïve approach: we don't listen to any other general because he might lie
  - Each general will act completely independently of the other, for example, making a random selection
  - Binomial distribution  $\binom{n}{k} p^k (1 - p)^{n - k}$  with n = 7 number of generals and p = 0.5
    - < 0.79% probability of all agreeing to attack
    - < 0.79% probability of all agreeing to retreat
- Since the chances is so low, the generals would probably be better off exchanging information among themselves to come to a common solution. But how?

D2. Do you have some idea? How would you do that?

# Why is it important to find a solution for this

Real life uses:

- If you want to attack a city. Very unlikely in XI century, but you never know...
- In computer systems (so pretty much everywhere). To be robust to:
  - Malicious attacks, even coordinated
  - Software errors, buggy code
  - Disks that corrupt, duplicate, lose or generate data
  - Corrupted packets
  - Really anything that doesn't follow our protocol
- Boeing 777 and SpaceX Dragon have been reported to use BFT systems [x]

[x] [https://en.wikipedia.org/wiki/Byzantine\\_fault](https://en.wikipedia.org/wiki/Byzantine_fault)

D3. Can you think about another cool use case?

# And if that's not enough...

stackoverflow

Search...

Home

PUBLIC

Stack Overflow

Tags

Users

Jobs

TEAMS

+ Create Team

I added an example in the answer post. – [yongrae](#) Aug 22 '18 at 23:35

thank you so much yongrae for your effort. However, I dont think this is correct. PBFT only makes sure that the system will tolerate up to  $1f$  byzantine nodes - this means if you get  $2/3$  of the nodes to agree, then at least 1 honest node has executed your request, and this is exactly what we want. There is no implication on the consistency of the internal state of all the nodes. Furthermore, even if we have 2 step commits - during the commit phase, some nodes can also drop out, causing the diagram above to be true as well. – [user2584960](#) Aug 24 '18 at 0:47

I added yet another example to answer your question. It covers specific voting scenarios where Byzantine primary and Byzantine replicas cooperate to destroy the protocol. – [yongrae](#) Aug 24 '18 at 20:21

Thank you again for the answer! I am loving this conversation - however I still think you are incorrect: in your example G3 had  $2f+1$  amount confirmation and they will perform commit. G1 is byzantine so that none of them will actually broadcast the commits to G2. However, G3 will, because they are honest replicas. In this case, G2 will hear at least " $f$ " replicas broadcasting the commit for "1", and per PBFT, after a replica receives " $f$ " number of commits, it will also commit. In addition, G3 can also construct proof that there has been more than  $2f+1$  prepares even if G1 is Byzantine – [user2584960](#) Aug 25 '18 at 21:31

this is a really good conversation yongrae, do you have a contact info I can use? I am looking to hire top talents that understand PBFT. – [user2584960](#) Aug 25 '18 at 21:32

"G1 is byzantine so that none of them will actually broadcast the commits to G2" --> It is wrong statement. Byzantine nodes can do everything in coordinated way. – [yongrae](#) Aug 27 '18 at 10:52

As explained earlier, G3 commit the value, and G2 doesn't. Safety condition states that G2 and G3 must have the same state. (We already have  $f$  number of Byzantine nodes;G1). – [yongrae](#) Aug 27 '18 at 11:00

Anyway, thanks for your interest about me, but I am currently not in job market though ;(. – [yongrae](#) Aug 27 '18 at 11:05

add a comment

# Introducing the paper and its contribution (1/2)

***Practical Byzantine Fault-Tolerance***

*Barbara Liskov and Miguel Castro*

1999

- Written back in 1999 but (as you can see) still studied today!
- Very important paper in distributed system area
- In 20 years, cited 1922 times [x]. That means once every 4 days.

[x] [https://scholar.google.co.uk/scholar?hl=en&as\\_sdt=0%2C5&q=Practical+Byzantine+Fault-Tolerance+by+Barbara+Liskov+and+Miguel+Castro&btnG=](https://scholar.google.co.uk/scholar?hl=en&as_sdt=0%2C5&q=Practical+Byzantine+Fault-Tolerance+by+Barbara+Liskov+and+Miguel+Castro&btnG=)

# Introducing the paper and its contribution (2/2)

***Practical Byzantine Fault-Tolerance***

*Barbara Liskov and Miguel Castro*

1999

- First paper to introduce a Byzantine Fault Tolerant algorithm in an asynchronous system
- Most previous works ignored Byzantine faults or assumed synchronous system model
- We'll learn a way to reach consensus in an asynchronous environment (like the internet) with up to **(n – 1)/3 byzantine nodes**

# Why up to one third?

The algorithm guarantees safety if up to  $\frac{n-1}{3}$  are faulty

This can be converted in saying that at least  $2f + 1$  have to be honest

## Why?

We must be able to proceed after communicating with  $n - f$  replicas.  
Indeed **f replicas could not respond at all because they are faulty.**

But what if the non-responding replicas are the good ones?  
We would like to still be able to outnumber the faulty.

## That means:

$$n - 2f > f \quad \longrightarrow \quad n > 3f$$

D4. Can you come up with a justification for the faulty nodes to be more? Or maybe less?

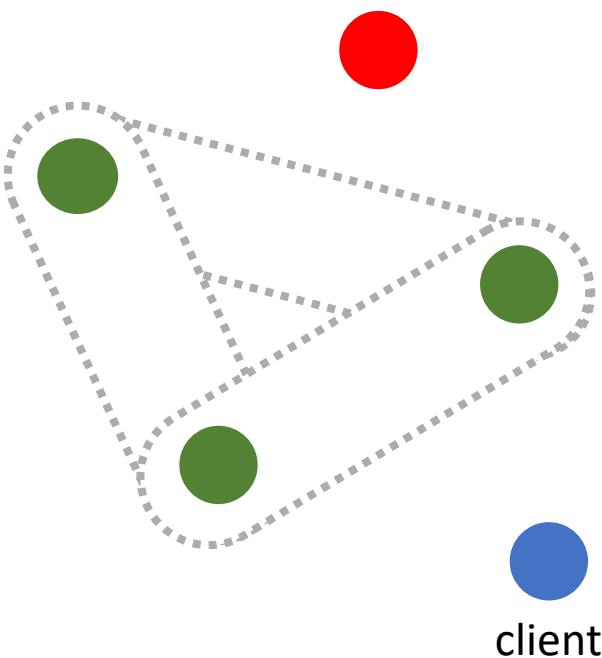
Example	
n	f
4	1
7	2
10	3

# Example with $n=4$

$f = 1$ , have to wait for  $1 + 1 = 2$  replies.

Safety is guaranteed:

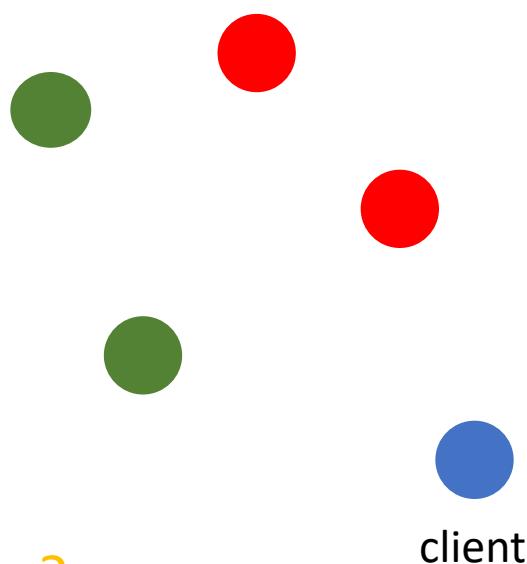
Eventually two honest node will reply



$f = 2$ , have to wait for  $2 + 1 = 3$  replies.

Safety is not guaranteed:

There is a Byzantine node in every possible combination



D5. What if the Client itself is  
Byzantine. What could happen?  
How would you handle it?

# Our System Model

- Certain number of nodes, also called Replicas  
(consideration about how many later)
- **Asynchronous distributed system.** The network could:
  - Fail to deliver messages (up to some point. **Eventually they should be received**)
  - Delay them
  - Duplicate them
  - Deliver them out of order
- Faulty nodes can behave arbitrarily i.e. **Byzantine failure** model
- We assume independent node failure
- The adversary is unable to subvert cryptography

# Terminology for the algorithm (informal definitions)

- **Primary**
- **Backup**
- **Replica**: Primary + Backup
- **View**: kind of the «epoch» of the system
- **Synchronous/asynchronous**: the delay when sending a message is bounded/unbounded respectively
- **Checkpoint**
- **Safety**: all non-faulty nodes agrees/decides on the same value
- **Liveness**: being able to proceed in the protocol without being stuck

# Each replica knows...

- State of the service
- Log with received messages
- View number
  - $p = v \bmod(R)$  –  $R$  is the total number of replicas
  - View number is tied to the Primary. If you know the View then you know the primary and the other way around

One important hypothesis is that the operations are deterministic:

if two different replicas perform the same operations on the same initial state they should end up being in the same final state

# Overview of the algorithm

Composed of four major steps:

1. Client sends a request to the Primary, invoking an operation
2. Primary multicasts the requests to all Backups
3. Replicas (Primary + Backups) execute the Request and send the result directly to the Client
4. Client waits for  $(f + 1)$  Replies from different Replicas with the same result

# Step one

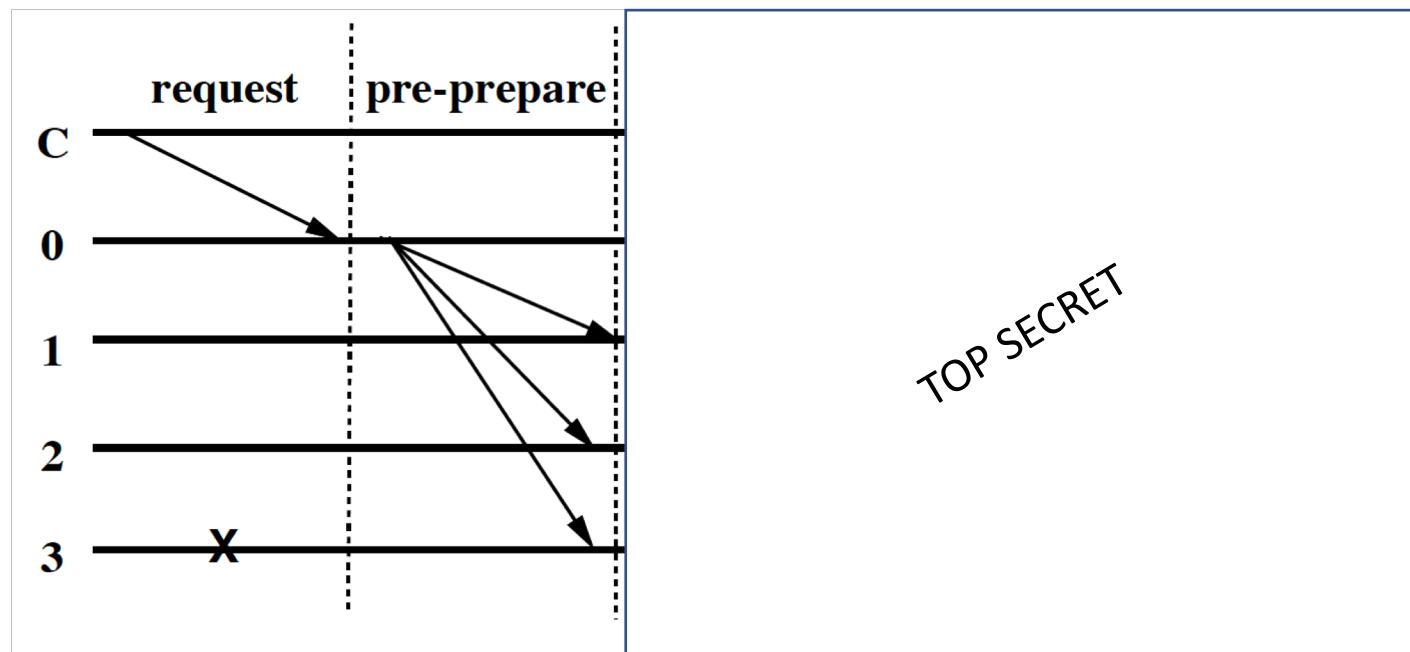


1. Client sends a request to the Primary, invoking an operation

<REQUEST, client ID, timestamp, operation> (signed by client)

# Step two

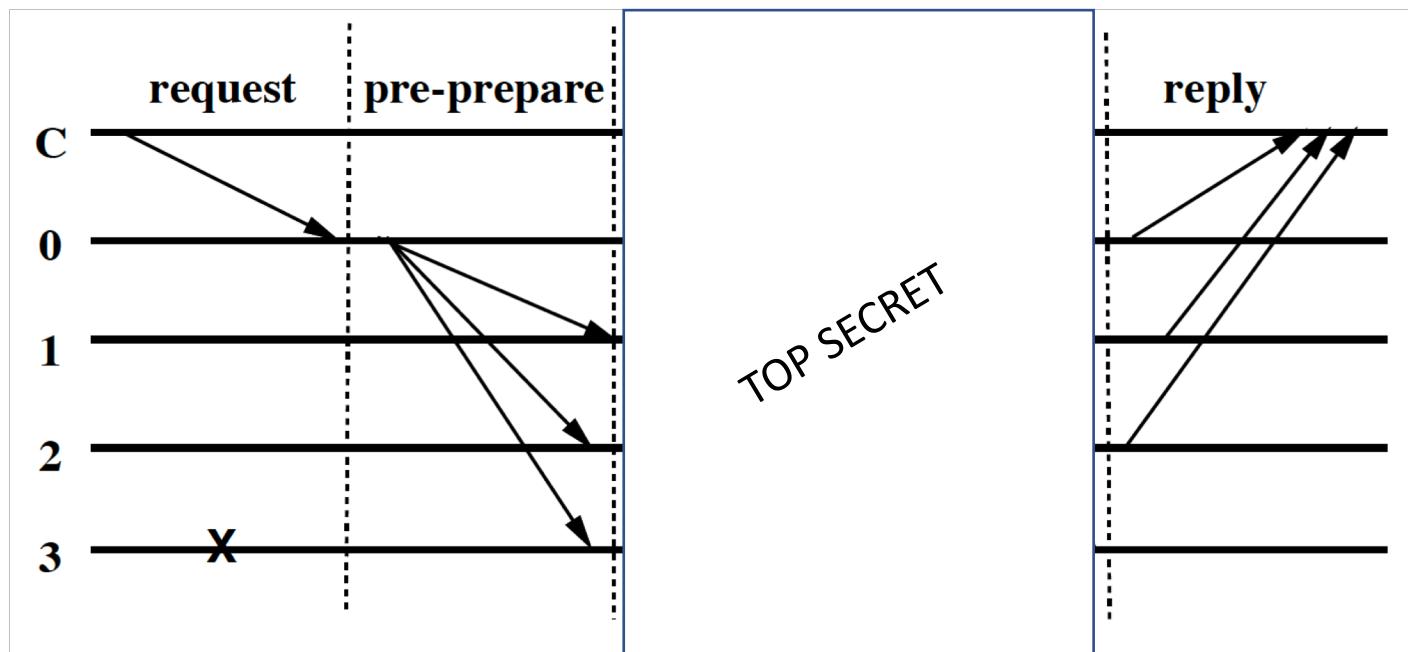
Purpose: link cryptographically the message with the sequence number n



2. Primary multicasts the requests to all Backups

<<PRE-PREPARE, view, n, digest(message)> (signed by primary), message>

# Step three



3. Replicas execute the Request and send the result directly to the Client

D6. Why Replies are sent back directly to the Client? Why don't we use a Primary like for the Request?

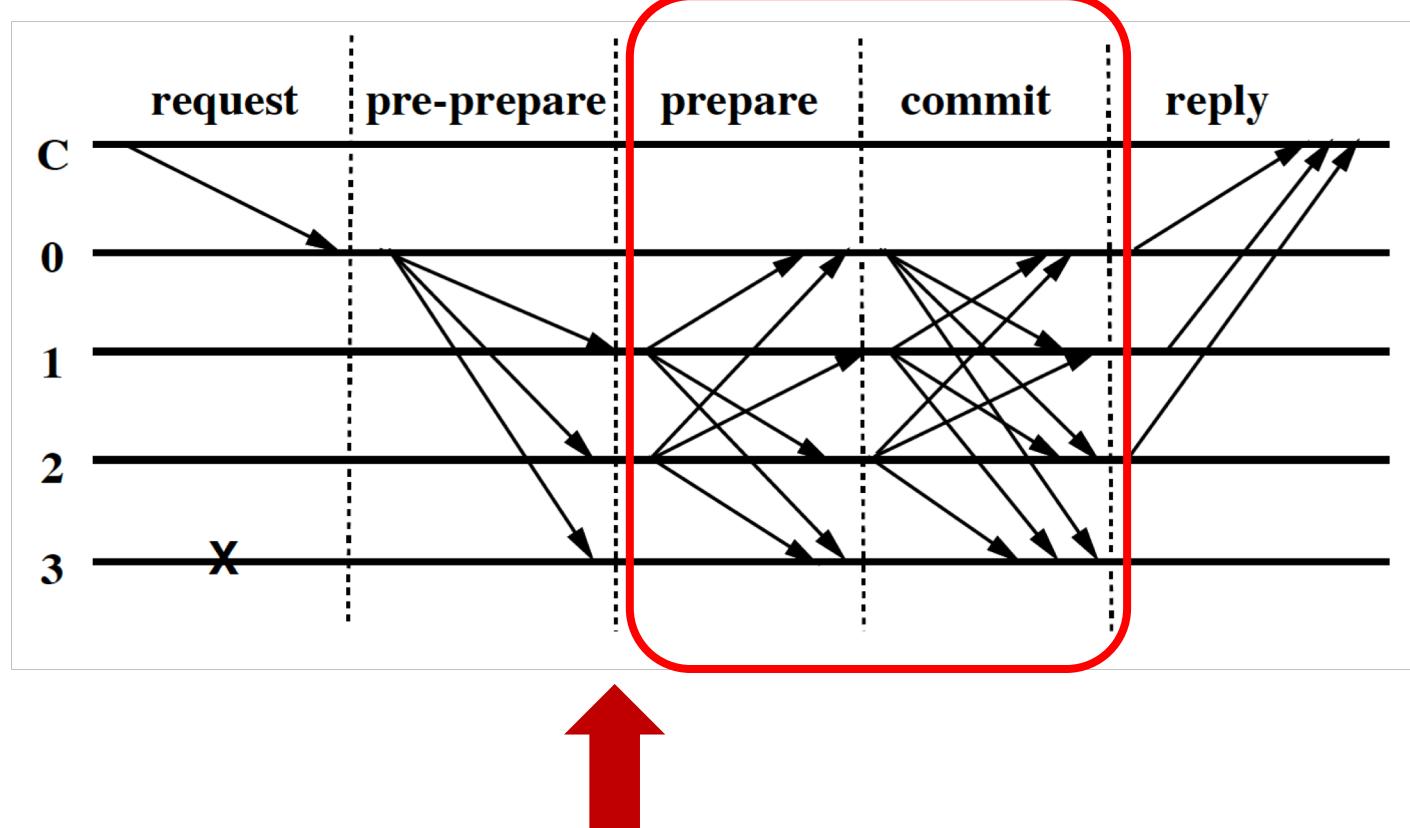
<REPLY, replica ID, client ID, timestamp, view, result> (signed by replica)

# Step four



4. Client waits for  $(f + 1)$  Replies from different Replicas with the same result

# Focus on step three (1/2)

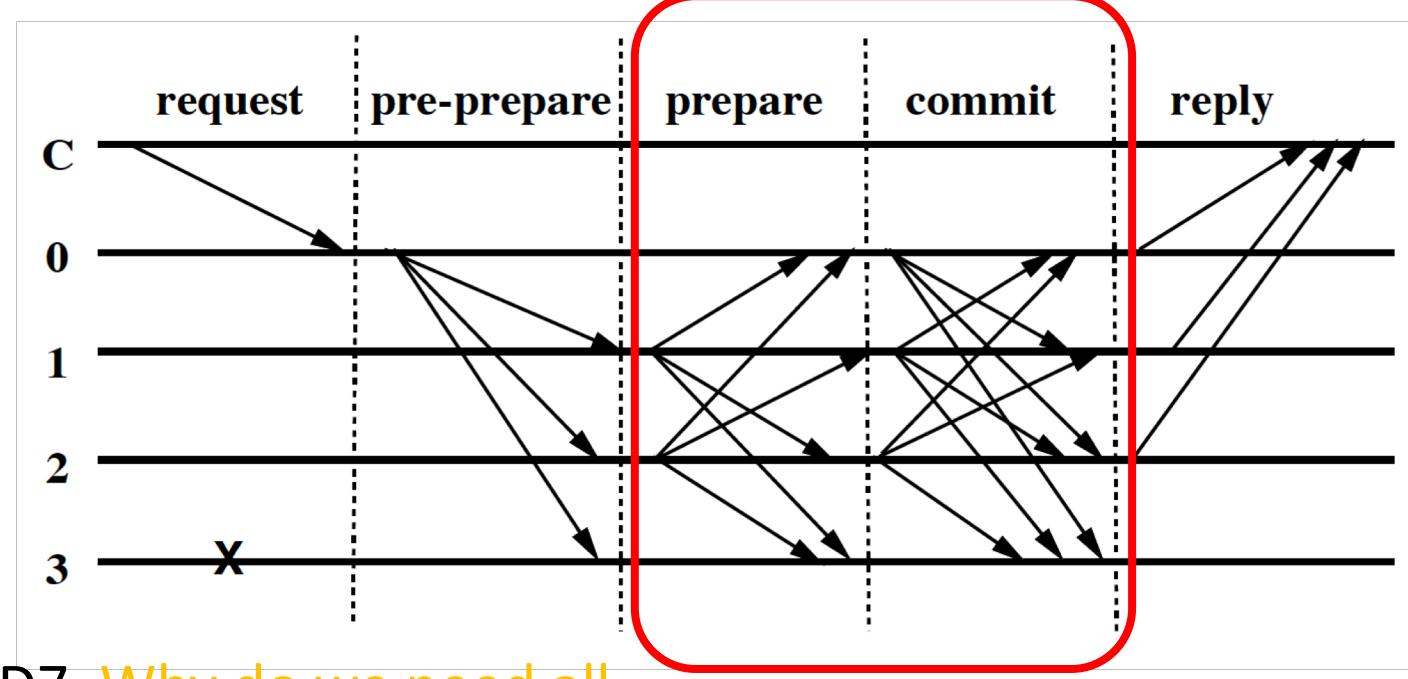


When a backup receives a **<PRE-PREPARE>** message it performs 4 checks:

- Checks the signatures and the digest
- Checks the View
- Inconsistencies
- Sequence number is within valid range

If so it passes into prepare phase and multicasts **<PREPARE>** message

# Focus on step three (2/2)



D7. Why do we need all those phases?  
Couldn't be less?



- When a Replica receives  $2f + 1$  <PREPARE> messages it enters the *committed phase* and multicasts a <COMMIT> message
- If a replica receives  $2f + 1$  <COMMIT> messages, it executes the requested operation and send a Reply to the Client

# Some considerations

- We have now illustrated the flow of one single request but in principle there could be **many requests, from many different Clients**
- It's up to the Primary to order them and multicast them
- The sequence number ensures that they are executed in the proper order, guaranteeing the safety property to the system
- This allows a big number of requests (transactions) to be executed
  - Considerations on how many later on, when comparing it to blockchains

# Garbage collection and Checkpoints

- As we said before each replica has a **log with the messages that has received and sent** up to now (<PRE-PREPARE>, <PREPARE>, <COMMIT>)
- We don't want it to grow indefinitely so we introduce the concept of Checkpoint
- If a Checkpoint is agreed with other Replicas (guess how many? ... **$2f + 1$** ) it then becomes stable and we can delete everything that is older

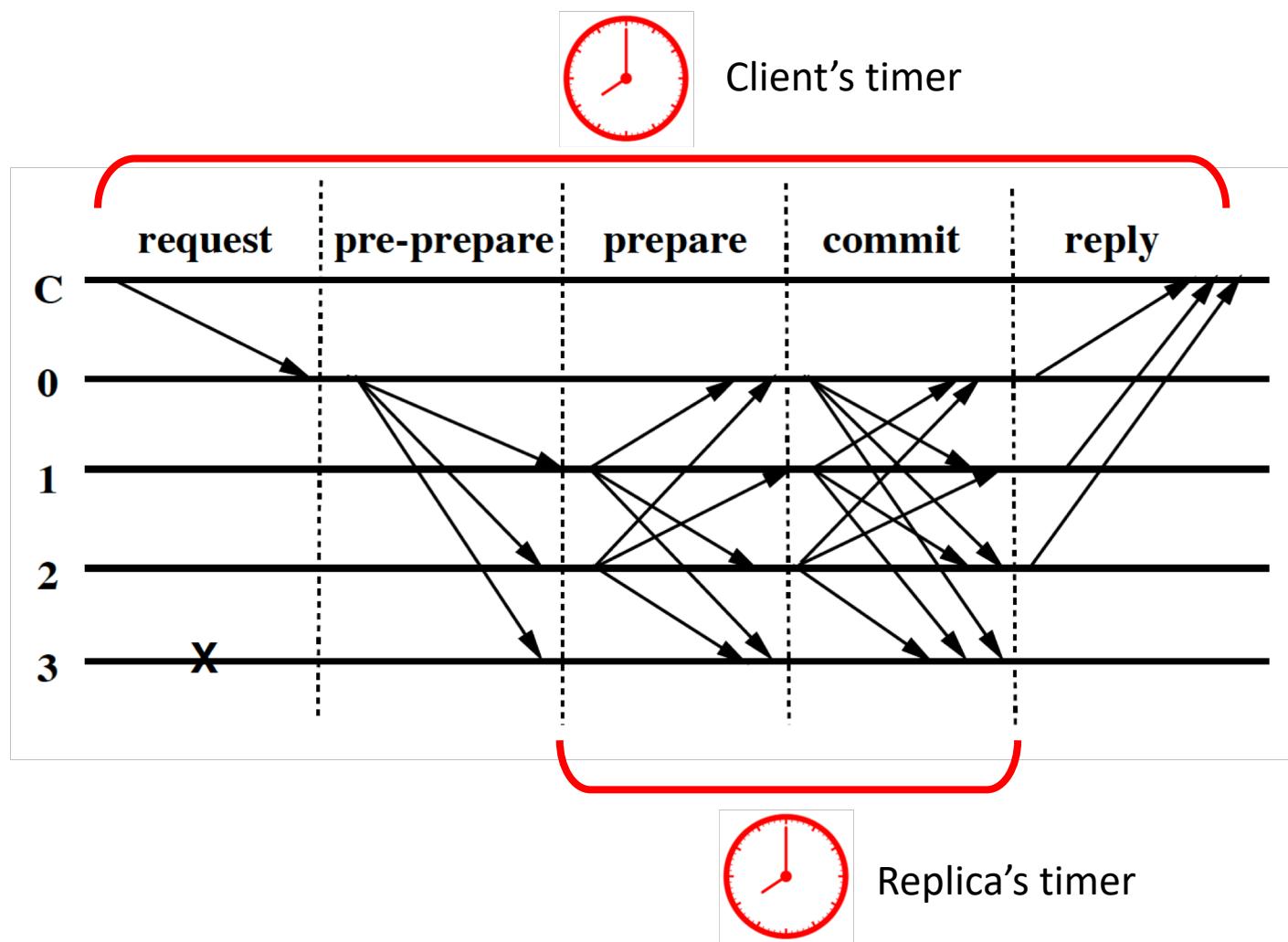
<CHECKPOINT, replica ID, n, digest> (signed by replica)

# View change process (1/2)

- What happens if a **primary** is faulty?  
We don't want to be stuck waiting for him. Do we?
- The Client has a timer. If it doesn't receive a Reply soon enough it broadcasts the Requests to all Replicas
- Also Replicas have timers that trigger a <View-Change> message
- It is necessary to provide liveness to the system i.e. be able to proceed and not be stuck
- It would be good to maximize the View duration for a honest Primary

D8. Why do we need a Primary at all? Can't we do without one?

# View change process (2/2)



D9. How should these timers be set? How long should we wait?

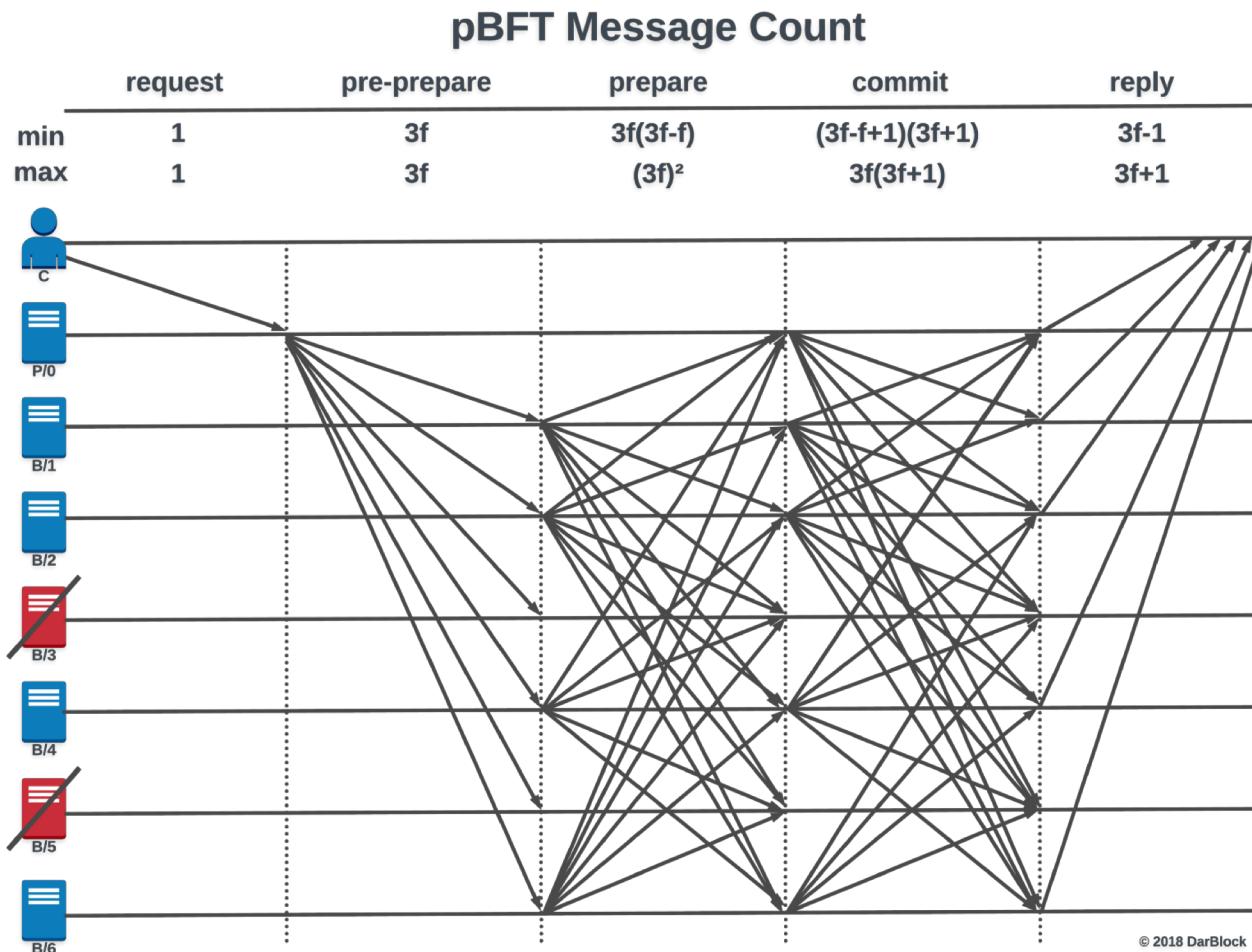
# Optimizations (briefly)

The described algorithm can be optimized in different ways, I will just mention a couple

- **I lied:** messages are not signed with a private keys but are authenticated using **MAC (Message Authentication Codes)**
  - $H(\text{message} + \text{secret key})$
  - Performance gain in the order of 1000
- Every Replica sends the replay back to the Client but **only one contains the result**, all the others only its digest (to save bandwidth)

Those optimizations allowed to implement a BFT service that is **only 3% slower** than a standard unreplicated service

# Consideration on number of messages exchanged



f	messages
2	92
3	191
...	...
30	16.472

- It doesn't scale well

<https://medium.com/coinmonks/pbft-understanding-the-algorithm-b7a7869650ae>

# The Fischer, Lynch and Patterson impossibility (FLP)

- FLP is a very important paper published in 1985
- The authors were awarded the Dijkstra Paper prize for it
- Demonstrates that **deterministic consensus in an asynchronous network is impossible in the presence of node failure**. This is true even for a single crash failure [3]
- Makes PBFT even more important

[1] Impossibility of Distributed Consensus with One Faulty Process - Fischer, Lynch, Paterson

[2] <https://www.the-paper-trail.org/post/2008-08-13-a-brief-tour-of-flp-impossibility/>

[3] <http://ug93tad.github.io/flp/>

# Introducing the second paper

***The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication***

*Marco Vukolich*

2015

- The paper compares PoW-based blockchains to those based on BFT state machine replication
- Good for an overview and to get to know pros and cons of each

# BFT vs PoW. A comparison

## Node identity management

Possibly the most fundamental difference

### Byzantine Fault Tolerant

- Nodes have to be identified uniquely (remember the <replica ID> in messages exchanged)
- A PKI infrastructure should be put in place with every replica knowing the public key (or MAC) of each other

### Proof of work

- Entirely decentralized

# BFT vs PoW. A comparison

## Consensus finality

If a valid block is appended to the blockchain at some point in time, it **should never be removed**

### Byzantine Fault Tolerant

- Satisfied by all BFT and state machine replication protocols

### Proof of work

- Obviously not the case.  
Suggested to wait 6 block i.e.  
60 minutes on average

# BFT vs PoW. A comparison

**Scalability** with respect to number of nodes (i.e. miners/replicas)

## Byzantine Fault Tolerant

- We have showed before how the number increases quadratically

## Proof of work

- Features thousands of mining nodes, demonstrating it in practice

# BFT vs PoW. A comparison

## Performance with focus on number of transactions

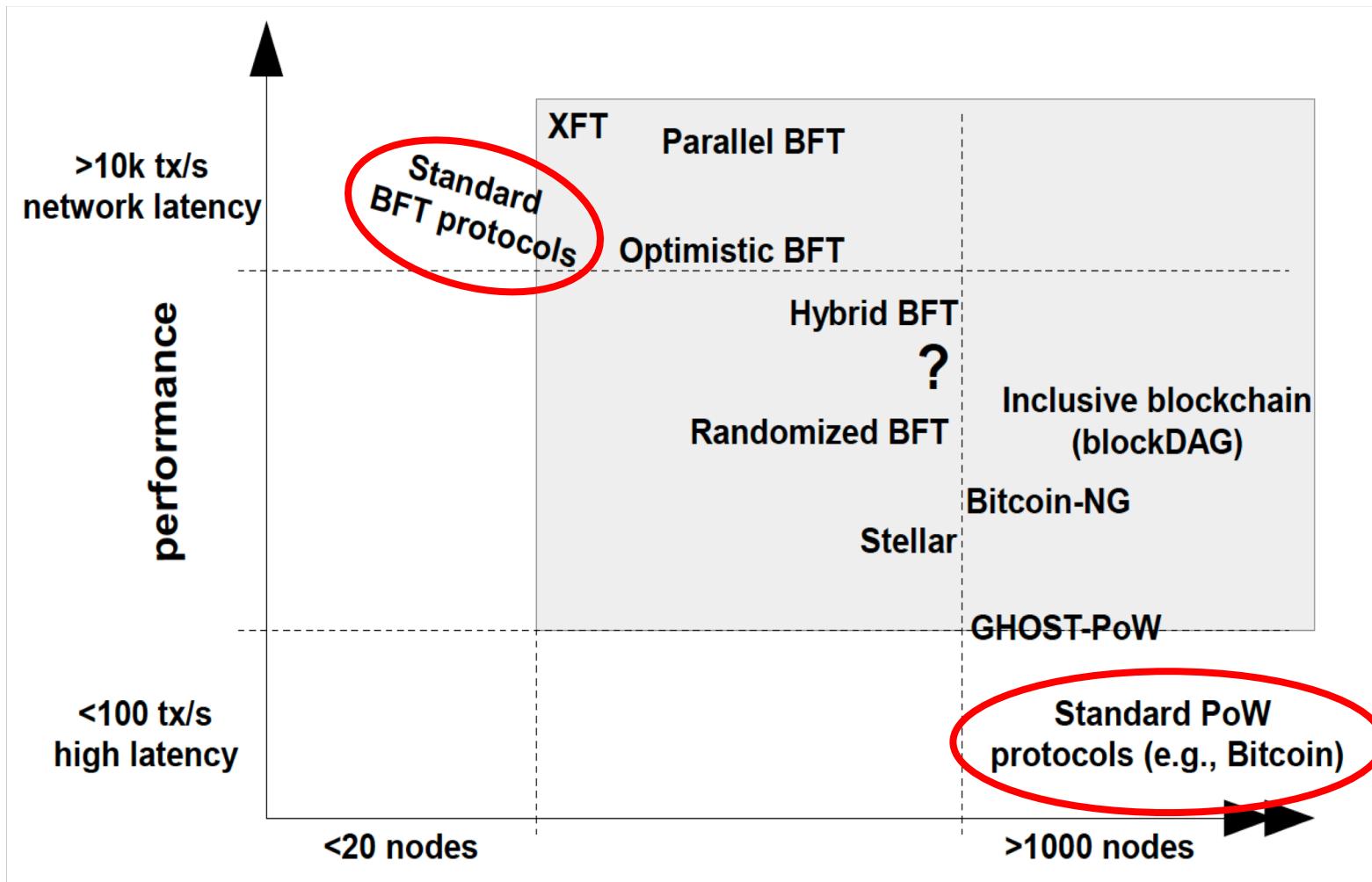
### Byzantine Fault Tolerant

- Confirmed to have sustained tens of thousands of transactions with network speed latencies

### Proof of work

- Limited to 7 TX/s (optimistically)

# Tradeoffs



# BFT vs PoW. A comparison

## Adversary

### Byzantine Fault Tolerant

- Robust up to **33%** of adversary nodes

### Proof of work

- Robust up to **25%** of adversary hash power (considering the selfish mining attack)

# BFT vs PoW. A comparison

## Network synchrony

### Byzantine Fault Tolerant

- Loosely synchrony communication is needed (i.e. at some point replicas should be able to receive the message)
- Otherwise it would break the FLP impossibility result

### Proof of work

- Loosely clock synchrony is needed for liveness
- We should be able to receive the mined block before the next one i.e. 10 minutes

# BFT vs PoW. A comparison

## Correctness proof

Byzantine Fault Tolerant	Proof of work
<ul style="list-style-type: none"><li>• Subject to detailed academic scrutiny</li><li>• Formal proofs can take an entire PhD thesis</li></ul>	<ul style="list-style-type: none"><li>• No thorough security analysis</li><li>• But after all it works in practice, right?</li></ul>

# Summary of BFT vs PoW

	PoW consensus	BFT consensus
Node identity management	<b>open, entirely decentralized</b>	permissioned, nodes need to know IDs of all other nodes
Consensus finality	no	<b>yes</b>
Scalability (no. of nodes)	<b>excellent (thousands of nodes)</b>	limited, not well explored (tested only up to $n \leq 20$ nodes)
Scalability (no. of clients)	<b>excellent (thousands of clients)</b>	<b>excellent (thousands of clients)</b>
Performance (throughput)	limited (due to possible of chain forks)	<b>excellent (tens of thousands tx/sec)</b>
Performance (latency)	high latency (due to multi-block confirmations)	<b>excellent (matches network latency)</b>
Power consumption	very poor (PoW wastes energy)	<b>good</b>
Tolerated power of an adversary	$\leq 25\%$ computing power	$\leq 33\%$ voting power
Network synchrony assumptions	physical clock timestamps (e.g., for block validity)	<b>none for consensus safety</b> (synchrony needed for liveness)
Correctness proofs	no	<b>yes</b>

# Conclusion

- We have introduced a practical algorithm to reach consensus in presence of malicious elements: **PBFT**
- If you think about it, it's very cool that a solution exists for such a difficult problem
- We compared the family of BFT state machine replication to PoW-backed blockchains, analyzing pros and cons of both
- Hopefully I have been able to give some insight with respect to this topic

# **Thank you very much**

# Question for discussion

1. Are PoS, PoW, etc. consensus algorithms? What do you think?
2. Do you have some idea about how to exchange information in a Byzantine environment? How would you do that?
3. Can you think about another cool use case?
4. Can you come up with a justification for the faulty nodes to be more? Or maybe less?
5. What if the Client itself is Byzantine. What could happen? How would you handle it?
6. Why Replies are sent back directly to the Client? Why don't we use a Primary like for the Request?
7. Why do we need all the phases? Couldn't be less?
8. Why do we need a Primary at all? Can't we do without one?
9. How should these timers be set? How long should we wait?
10. Do sequence numbers grow limitlessly?
11. What if Byzantine nodes change over time but within the limit? Would this be a problem?
12. We would require independent implementation of the code. Is it doable in the real world?
13. How would you subvert the protocol?
14. Do you like more BFT or PoW?

D3 hint. Another cool use case

