Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults

Dian Yu

Comparison with PBFT (Traditional BFT protocols)

Similarities:

Build practical Byzantine fault tolerance systems

Protocol: Clients → Primary → Replicas → Agreement

Differences: (Robust)

Signature for authentication

Regular view change

Point to point communication

Ideal BFT systems

“Handle normal and worst case separately as a rule because the requirements for the two are quite different. The normal case must be fast. The worst case must make some progress”

Gracious execution: synchronous execution. All clients and servers behave correctly

Uncivil execution: synchronous execution. Up to f servers and any numbers of clients are Byzantine

Problem with PBFT/Zyzzyva

Misguided: current BFT systems can survive Byzantine faults, but completely unavailable by a simple failure

Dangerous: encourages fragile optimizations

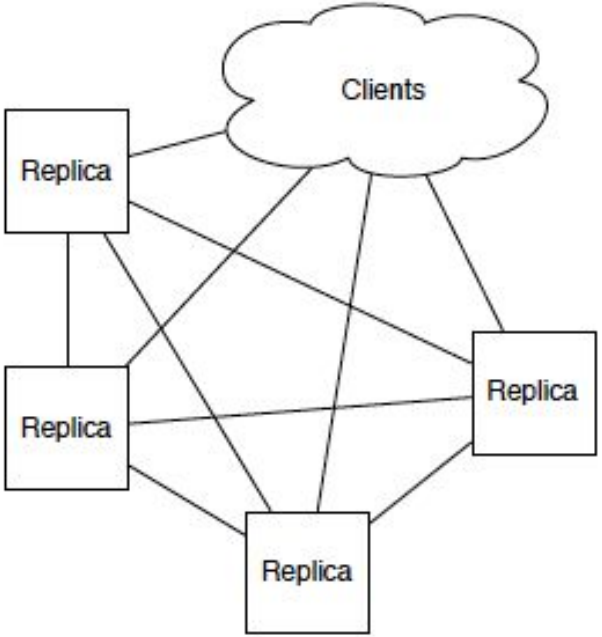
Futile: Further improvements have little effect on performance

System	Peak Throughput	Faulty Client
PBFT [8]	61710	0
Q/U [1]	23850	0†
HQ [12]	7629	N/A‡
Zyzzyva [18]	65999	0
Aardvark	38667	38667

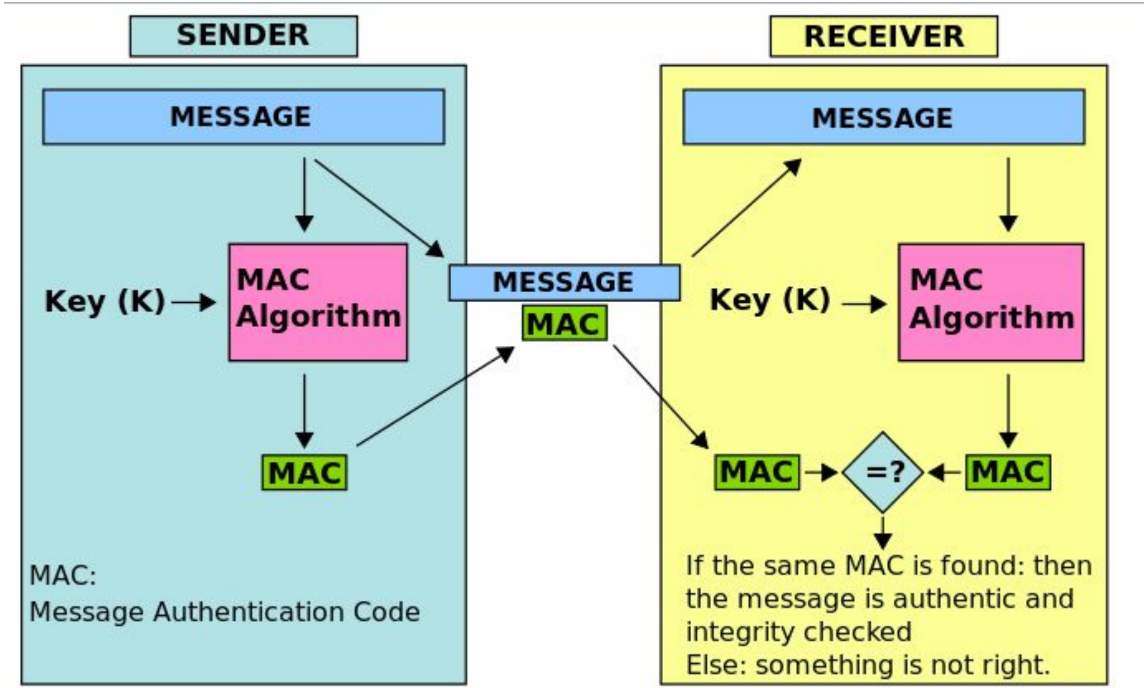
Aardvark: RBFT in action

3 stages:

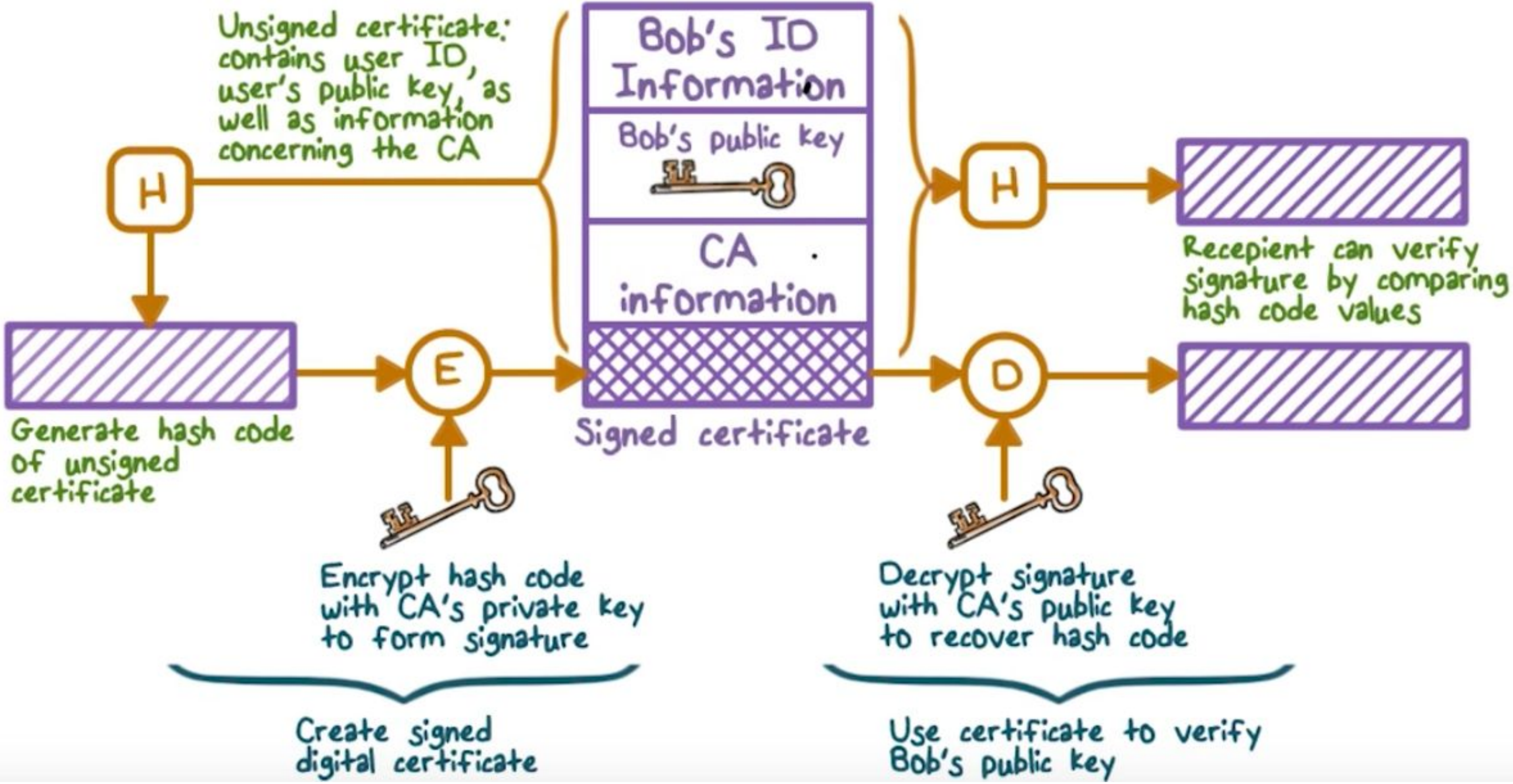
1. Client request transmission
2. Replica agreement
3. Primary view change



Signed client requests - MAC



Digital Signature



Signed client requests - digital signatures

Problem with MAC: no non-repudiation property of digital signatures

Solution: Signature

- Valid MAC but not valid signature:
  - Not routine message corruption
  - Significant fault or malicious behavior with client

Denial-of-service attack?

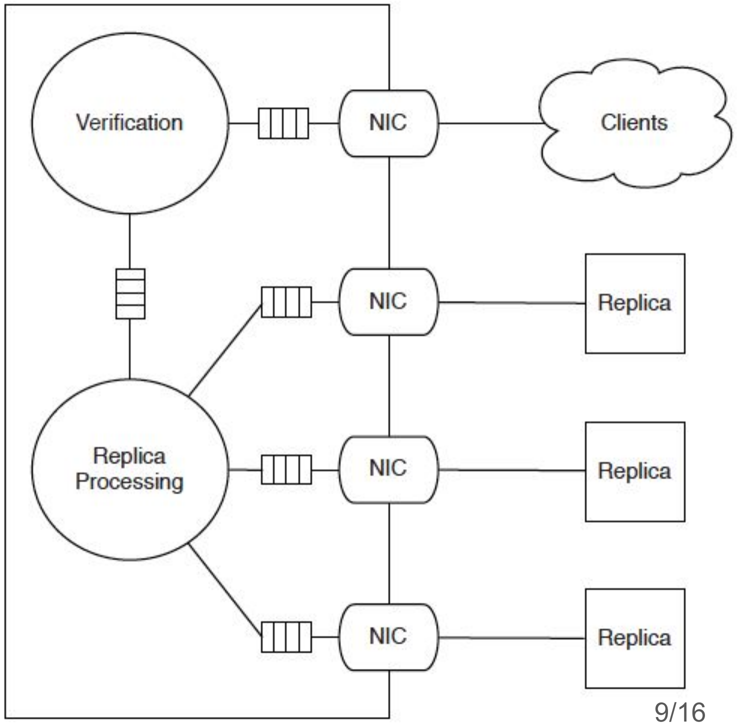
1. Hybrid MAC-signature construct
2. Complete one request first

Resource isolation

Separate network interface controllers (NICs)

Separate work queues for clients and replicas

Hardware parallelism

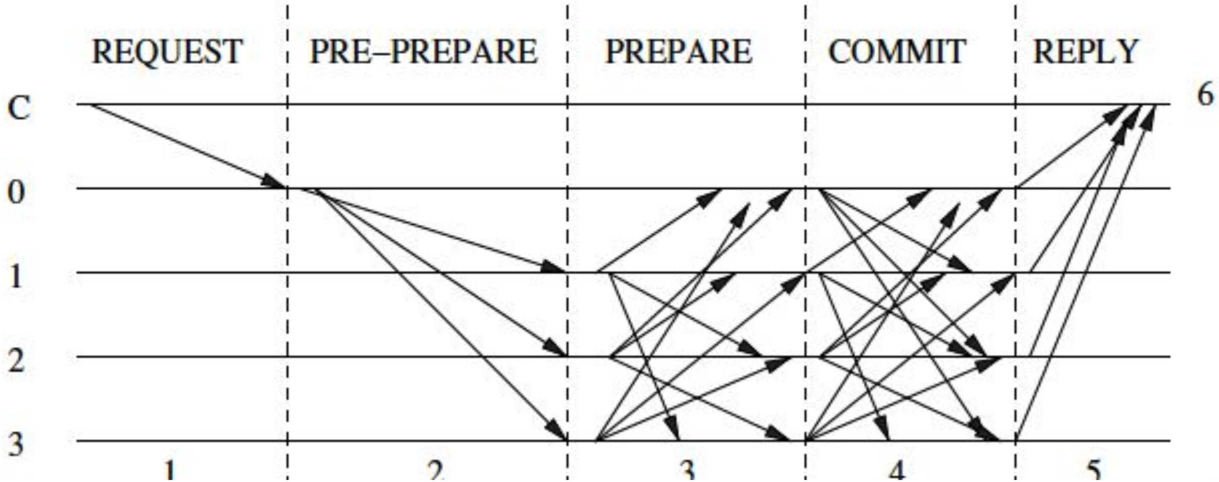


Regular view changes

System throughput remains high when replicas are faulty (uncivil intervals)

Cost of a view change is similar to the regular cost of agreement

Protocol Description



Client request transmission

Fundamental challenge:

Each replica comes to the same conclusion about the authenticity of the request

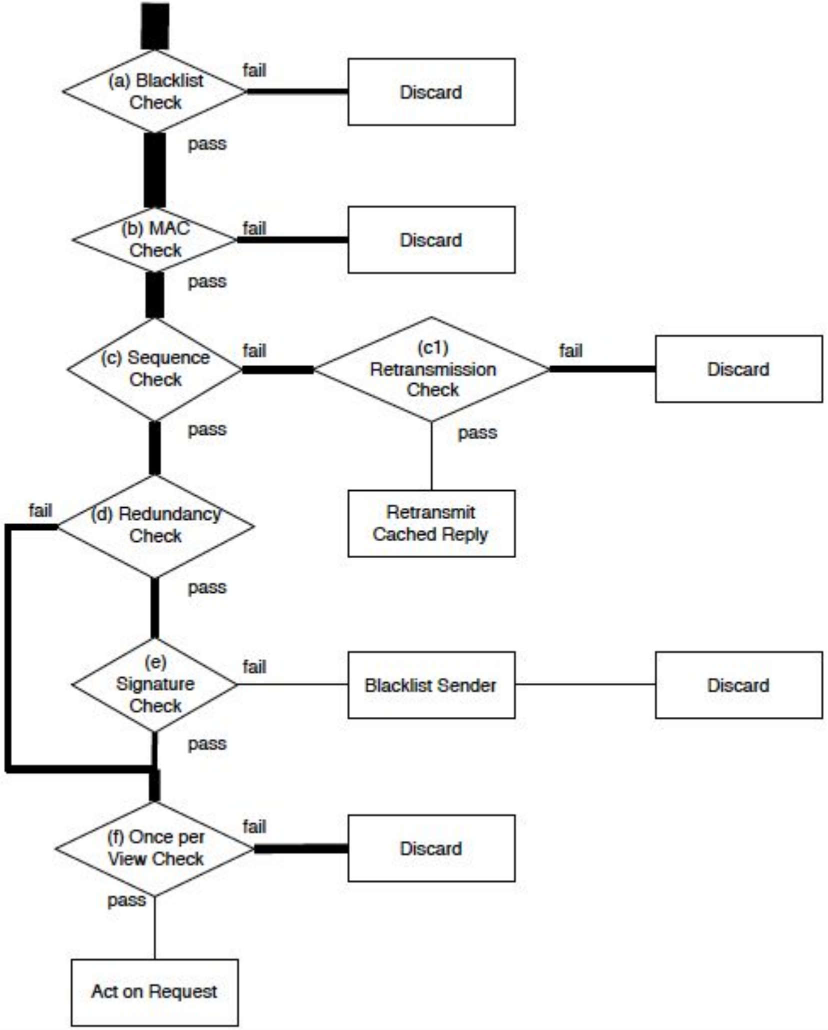
Request:

$$\langle\langle \text{REQUEST}, o, s, c\rangle_{\sigma_c}, c\rangle_{\mu_{c,p}}$$

Analysis:

Signature check: ensures only requests that will be accepted by all correct replicas are processed.

Result: for every k correct requests submitted by a client, each replica performs at most k+1 signature verifications.



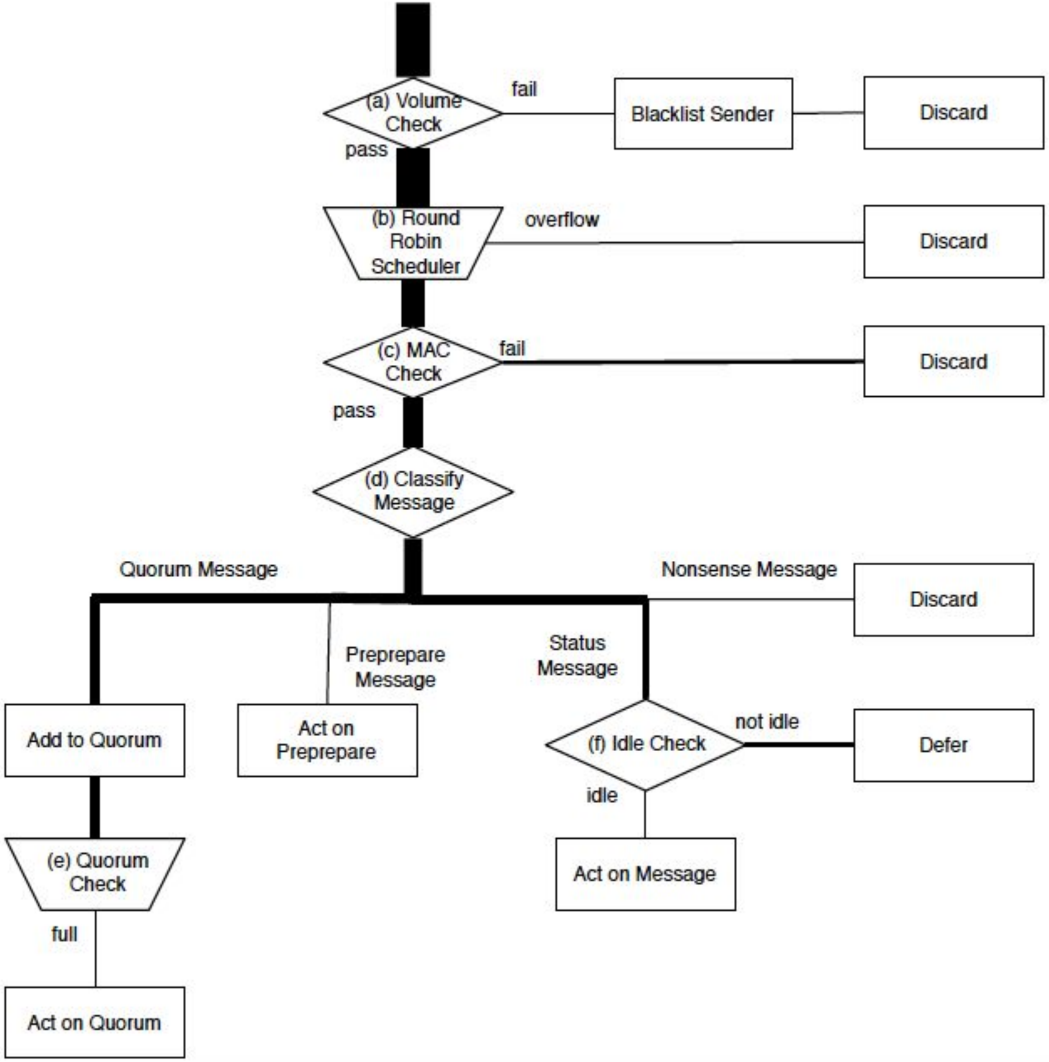
Replica agreement

Fundamental Challenge:

Ensure each replica can quickly collect the quorums of PREPARE and COMMIT messages necessary to make progress.

Potential solution:

1. Design a protocol so that incorrect messages from faulty replica will not gain quorum
2. If quorum of timely correct replicas exists, a faulty replica cannot impede progress.



Catchup messages

Benefit: allows temporarily slow replicas to avoid becoming permanently non-responsive

Downside: faulty replicas impose significant load on non-faulty counterparts

Primary view changes

Faulty primary: delay processing requests, discard requests, corrupt clients' MAC authenticators, introduce gaps in the sequence number space, unfairly delay or drop clients' requests

Past systems: conservative. Only change when the current primary does not allow the system make even minal progress

Aardvark: initiate a view change when delay exceeds heartbeat timer expires.

Fairness: PRE-PREPARES from the same client

Analysis (with proof)

1. Peak throughput during a gracious view
2. During uncivil executions, with a correct primary Aardvark's throughput at least g times the throughput of a gracious view

System	Peak Throughput	Network Flooding	
		UDP	TCP
PBFT	61710	crash	-
Q/U	23850	23110	crash
HQ	7629	4470	0
Zyzzyva	65999	crash	-
Aardvark	38667	7873	-

System	Peak Throughput
Aardvark	38667
PBFT	61710
PBFT w/ client signatures	31777
Aardvark w/o signatures	57405
Aardvark w/o regular view changes	39771

Conclusion

All previous BFT (PBFT, QU, HQ, Zyzzyva) were broken under Byzantine fault

A system surviving the worst case doesn't mean it works well. Should make it work well in worst case as well.

A small adaptation for parallelism might improve the performance a lot

A robust system should give adequate performance in any scenario

Questions?