Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults

Group 6

- Eric Han
- Mayuri Wadkar
- Sonali Mishra

Agenda

- Problem with the existing BFT protocols.
- Practical Byzantine Fault Tolerance (PBFT)
- Robust Byzantine Fault Tolerance (RBFT)
- Gracious Execution
- Uncivil Execution
- Aardvark - Overview
- Aardvark - Protocols

Overview of the Paper

In this paper:

- It is demonstrated that the existing Byzantine Fault Tolerant (hereby referred as BFT) protocols are dangerously fragile.
- A set of principles are defined for constructing BFT services, that remain useful even when Byzantine Faults occur.
- These principles are applied to construct a new protocol -- Aardvark.

Existing Problem:

Most of the BFT protocols are designed with a single minded approach based on the best case performance of the system.

Proposed Solution:

Primary contribution of this paper is to propose a new approach by shifting the focus from building high performance systems that are designed based on the best case scenario, to building systems that offer acceptable performance under all circumstances, including when faults occur.

Practical Byzantine Fault Tolerance (PBFT)

- Most existing BFT systems are based on this approach.
- PBFT: First characterize what defines normal case and then make the system perform well for that case. Normal and Worst cases are handled separately.
- Problem with this approach:
- The normal case includes only *gracious executions*.

Gracious Executions

An execution is gracious when:

- The execution is synchronous with some implementation - dependent short bound on message delay

And

- All clients and servers behave correctly.

Uncivil Execution

An execution is uncivil if:

- The execution is synchronous with some implementation - dependent short bound on message delay
- Up to f servers and an arbitrary number of clients are Byzantine
- All remaining clients and servers are correct

Proposed Approach - Robust Byzantine Fault Tolerance (RBFT)

A BFT system should be designed based on three properties:

1. It provides acceptable performance
2. It is easy to implement
3. It is robust against Byzantine faults.

Hence RBFT is proposed in this paper to build systems that provide adequate performance during *uncivil executions*.

RBFT implemented : Aardvark

- Aardvark is a new BFT system designed and implemented to be robust to failures.
- Aardvark is based on the approach this paper proposed, i.e. acceptable performance, ease of implementation, and robustness against Byzantine disruptions.

Aardvark vs previous BFT systems

There are three main design differences between Aardvark and previous BFT systems.

1. Signed Client Requests
2. Resource Isolation
3. Regular View Changes

1. Signed Client Requests

Aardvark clients use digital signatures to authenticate their requests.

Digital signatures provide non-repudiation.

Also ensures that all correct replicas make identical decisions about the validity of each client request.

Existing protocols make use of message authentication codes (MAC) for authentication.

1. Signed Client Requests

Challenge #1 with using Digital Signatures:

Digital signatures are usually seen as too expensive to use.

Handled in Aardvark:

Aardvark uses them only for client requests pushing the expensive act of generating signatures on clients while leaving servers with less expensive verification.

1. Signed Client Requests

Challenge #2 with using Digital Signatures:

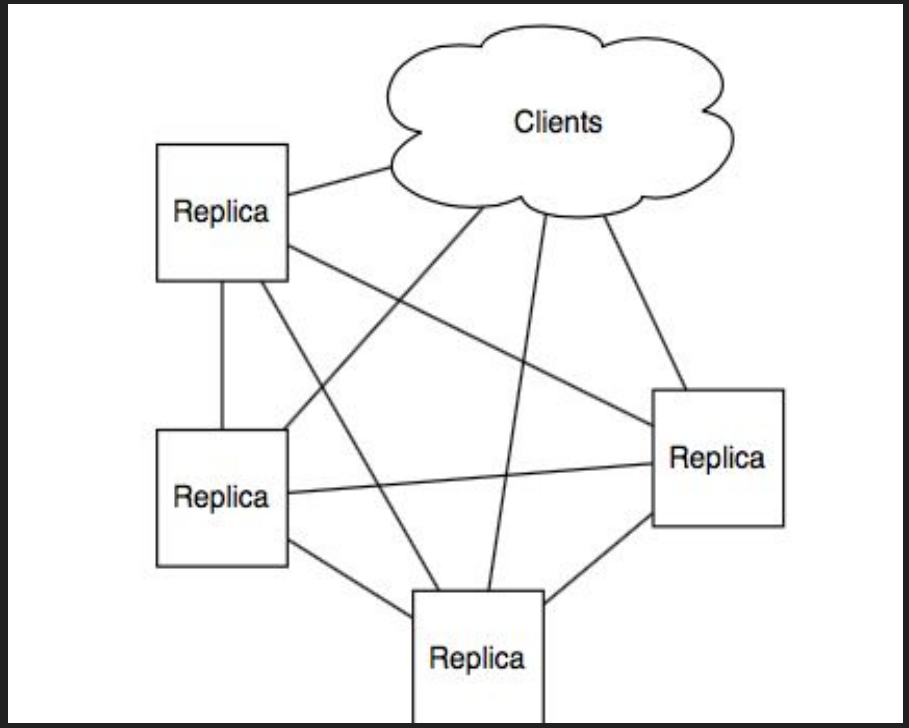
Due to the additional costs associated with verifying signatures instead of MAC's, Aardvark must guard against denial-of-service attacks where the system receives a large numbers of requests with signatures that need to be verified.

Handled in Aardvark:

This implementation limits the number of signature verifications a client can inflict on the system by forcing a client to complete one request before issuing the next.

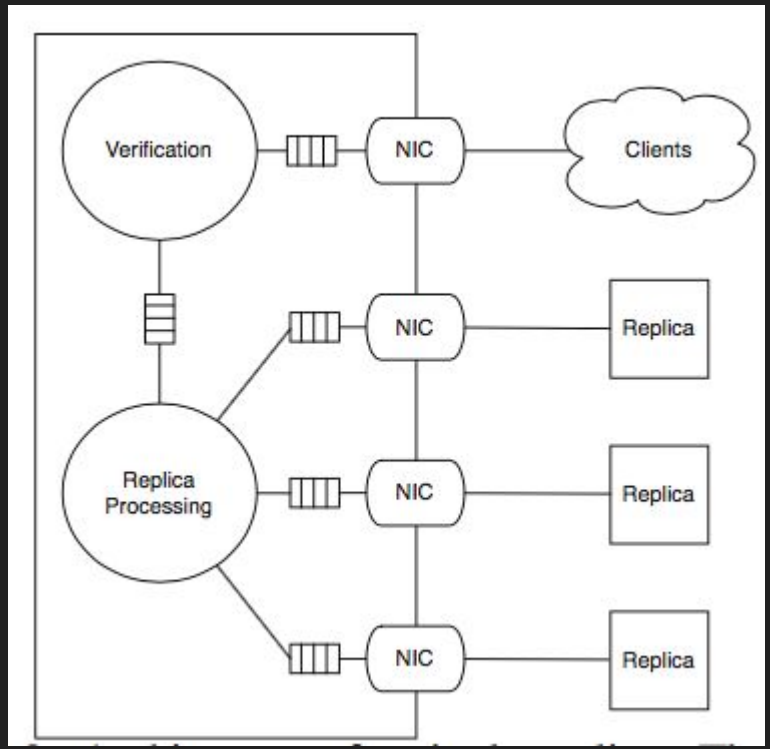
2. Resource Isolation

- Aardvark implementation isolates network and computational resources.
- Separate NICs and wires are used to connect each pair of replicas.
- Prevents faulty server from interfering with the timely delivery of messages from good servers.



2. Resource Isolation

- Aardvark uses separate work queues for processing message from clients and individual replicas.
- This prevents client traffic from drowning replica to replica communications.
- Replica uses a separate NIC for communicating with other replica and a final NIC to communicate with the collection of clients.



3. Regular view changes

A primary replica remains high only if it achieves some increasing level of throughput.

Aardvark prevents throughput degradation caused by a faulty primary, by either forcing the primary to be fast or selecting a new primary.

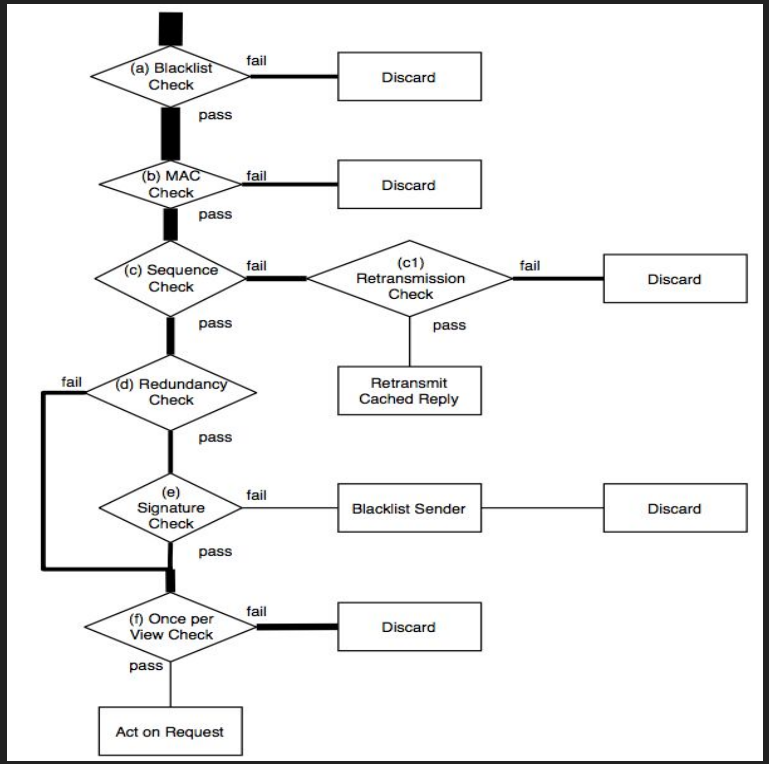
Aardvark Protocol

Aardvark protocol consists of three stages:

1. Client Request Transmission
2. Replica Agreement
3. Primary view change

1. Client Request Transmission

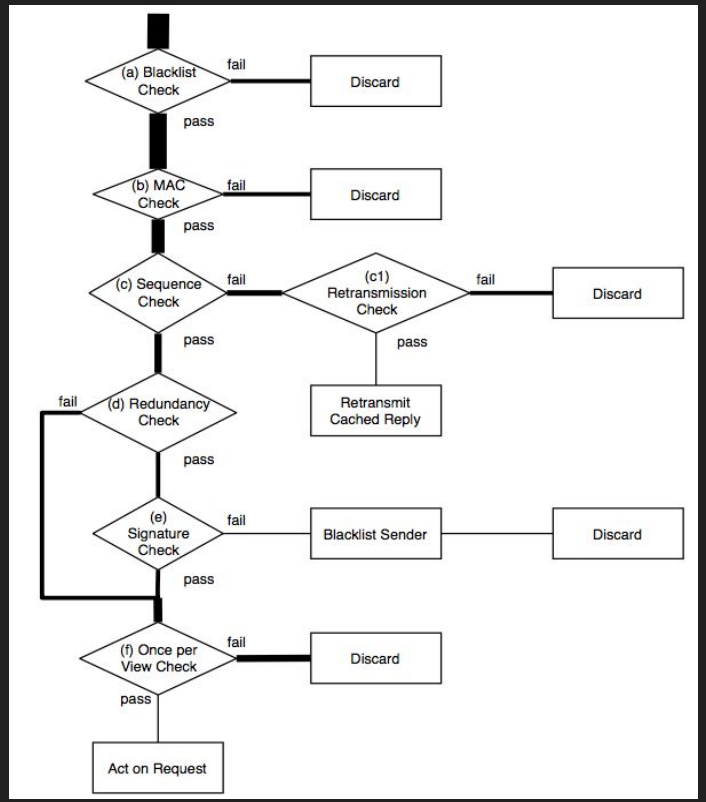
a. **Blacklist Check:** If the sender *c* is not blacklisted, then proceed to step (b). Else discard the message.
b. **MAC check:** If the MAC is valid, then proceed to step (c). Else discard the message.



1. Client Request Transmission

c. **Sequence check:** Check the most recent cached reply to *c* with sequence number s_{cache}. If the request sequence number s_{req} is exactly s_{cache} + 1, then proceed to step (d).

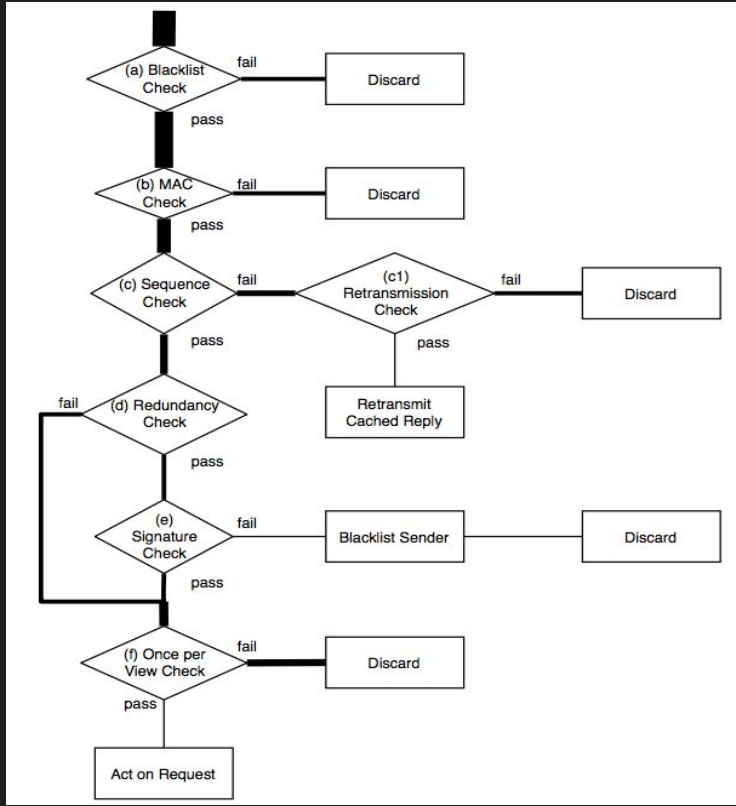
c1. **Retransmission check:** If a reply has not been sent to *c* recently, then retransmit the last reply sent to c. Else discard the message.



1. Client Request Transmission

d. **Redundancy check:** Check the most recent cached request from *c*. If no request from *c* with sequence number s_{req} has previously been verified, proceed to step (e). Else if the request matches, signature check step is skipped.

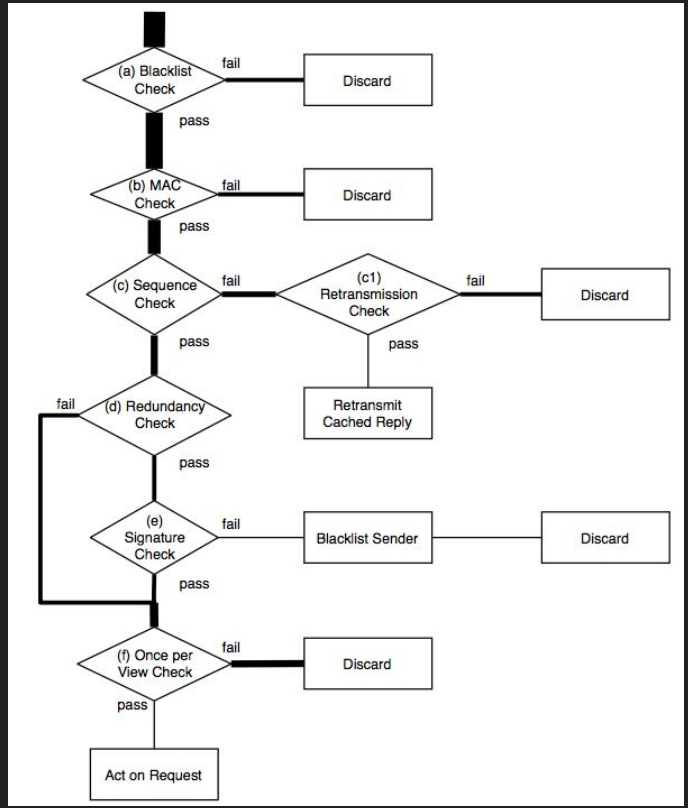
e. **Signature check:** If the signature is valid, proceed to step (f). Else blacklist the sender and discard the message.



1. Client Request Transmission

f. **Once per view check:** If an identical request has been identified in the previous view, but has not been processed in the current view, act on the request.

Else, discard the message.

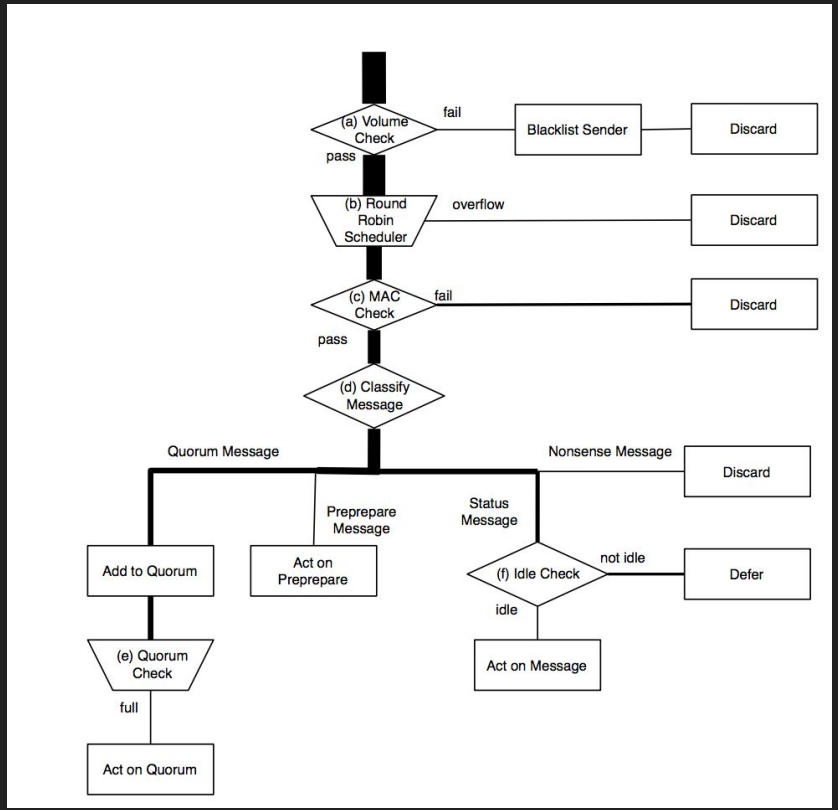


2. Replica Agreement

(a). If replica *q* is sending too many messages, blacklist *q* and discard te message.

Else proceed to step (b).

(b). **Round Robin Scheduler:** Select the next message from pending messages in round robin order. Discard received messages when the buffers are full.

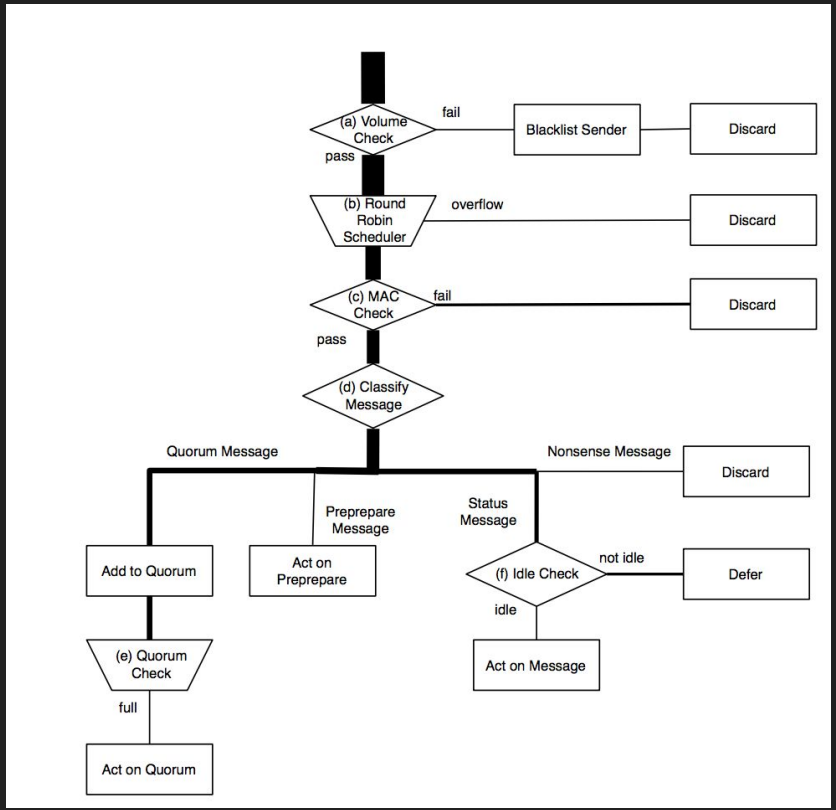


2. Replica Agreement

(c). **MAC check:** If the selected message has a valid MAC, proceed to step (d).

Else discard the message.

(d). **Classify Message:** Classify the authenticated message as per its type.



2. Replica Agreement

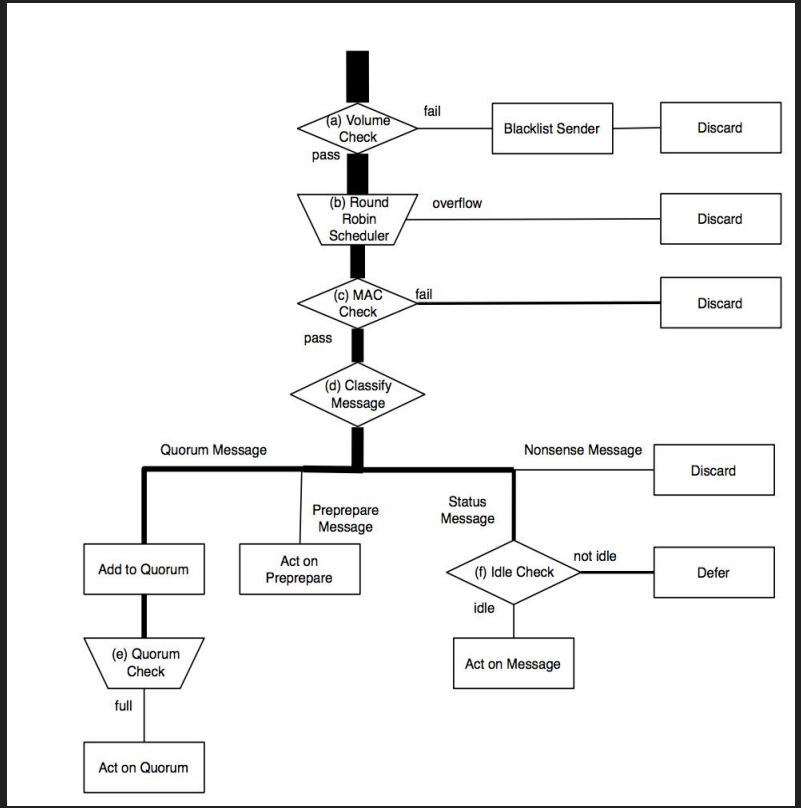
Message Type:

PRE-PREPARE: Process it in protocol. STEP 3.

PREPARE or COMMIT: Add it to appropriate quorum and proceed to step (e).

Catch up message: Proceed to step (f).

Anything else: Discard message.

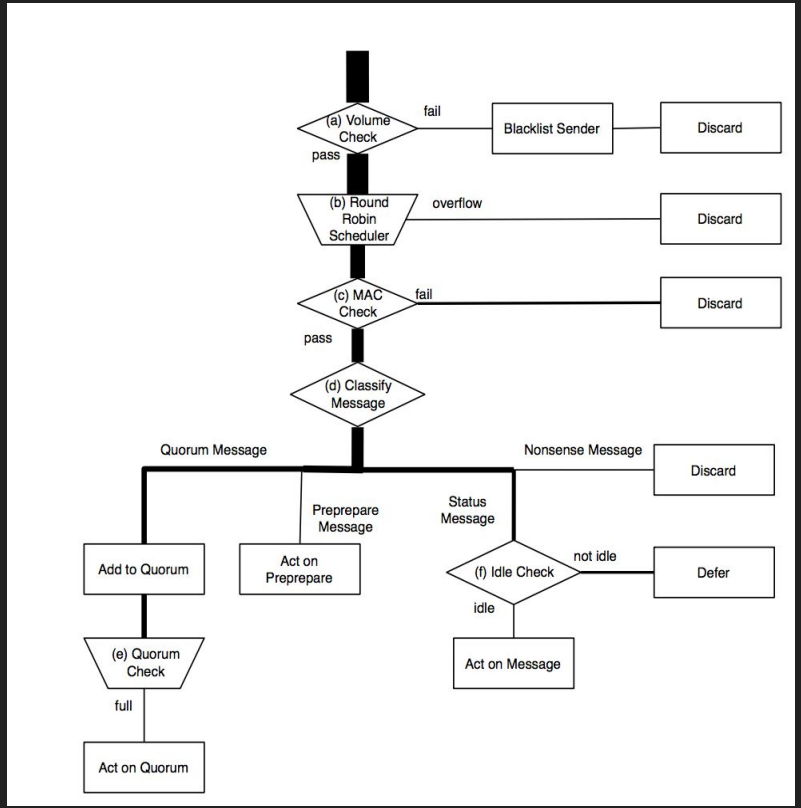


2. Replica Agreement

(e). **Quorum check:** If the quorum to which the message was added is complete, act as appropriate in protocol STEP 4-6.

(f). **Idle check:** If system has free cycles, process catchup message.

Else wait until system is idle.



2. Replica Agreement

Agreement Protocol STEP 2-6:

STEP 2: Primary replica forms a PRE-PREPARE message with a set of valid requests and sends the PRE-PREPARE to all replicas.

STEP 3: Replica receives PRE-PREPARE, authenticates it and sends a PREPARE to all other replicas.

STEP 4: Replica receives *2f* PREPARE messages that are consistent with PRE-PREPARE and sends a COMMIT message to all other replicas.

STEP 5: Replica receives *2f + 1* COMMIT messages, commits and executes the request, sends a REPLY message to the client.

2. Replica Agreement

STEP 6: The client receives *f + 1* matching REPLY messages and accepts the request as complete.

3. Primary View Changes

This approach employs a primary replica to order requests.

Employing a primary avoids the need to trust clients to obey a backoff protocol.

However since the primary is responsible for selecting which requests to execute, the system throughput is at most the throughput of the primary.

The primary is in a position to control overall system progress and fairness to individual clients.

3. Primary View Changes - Monitoring

Replicas monitor the throughput of the current primary.

If a replica judges the primary’s performance to be insufficient, then the replica initiates a view change.

Replicas in Aardvark expect two things from the Primary:

1. A regular supply of PRE-PREPARE message.
2. High sustained throughput.

When a view change is completed, each replica starts a heart-beat timer that is reset whenever the next valid PRE-PREPARE message is received.

If a replica does not receive the PRE-PREPARE message before the timer times out, it initiates a view change.

3. Primary View Changes - Fairness to Clients

Primary replicas can influence which requests are processed.

A faulty primary could be unfair to a specific client by neglecting to order requests from that client.

To avoid this, replicas track fairness of request ordering.

When a replica receives a request from a client *c* for which it has not received PRE-PREPARE message, it records the sequence number of most recent PRE-PREPARE message.

If if receives two PRE-PREPARE messages for another client before receiving for client *c*, it declares the current primary to be unfit and initiates a view change.

Conclusion

- The paper claims that high assurance systems require BFT protocols that are more robust to failures than existing systems.
- The paper presents Aardvark as the first BFT protocol designed to provide good performance for Byzantine faults.

Thank You