Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults

By:
Deepthi Vishwanath
Isha Pradhan
(Group 3)

AGENDA

- Introduction
- RBFT and Aardvark
- System Model
- Recasting the problem
- Aardvark: RBFT in Action
- Protocol Description
- Aardvark and RBFT
- Aardvark: Primary View Changes
- Aardvark: Adaptive Throughput
- Aardvark: Fairness
- Performance Evaluations
- Performance: Faulty Clients
- Performance: Network Flooding
- Performance: Faulty Primary
-

INTRODUCTION

- The recently developed byzantine fault tolerant (BFT) state machine protocols do not tolerate byzantine faults very well.
- A single faulty client or server can render the whole system unusable by inflicting multiple orders which reduces the throughput or renders the system unavailable for a long time.
- Such performance degradations are not what one expects from a system that claims to be fault tolerant.

So, can these systems really be called Byzantine Fault Tolerant???

INTRODUCTION

System	Peak Throughput	Faulty Client
PBFT - Practical Byzantine Fault Tolerant System	61710	0
Q/U	23850	0
HQ - Hybrid Quorum Protocol	7629	N/A
Zyzzyva	65999	0
Aardvark	38667	38667

Table 1: Here is a table, which shows the peak throughput of different systems in a fault-free case and in the presence of a single faulty client.

INTRODUCTION

- The reason for such a collapse in the results is believed to be a single-minded focus on improving the performance of the system in a fault-free condition.
- Since many years, BFT execution is believed to be too expensive to be practical.
- This has led to protocols that undermine robustness because of its complexity:
  - The protocol design include fragile optimizations and,
  - The protocol's implementation fails to handle all the corner cases effectively
- The primary contribution of this paper is to advocate a new approach, robust BFT (RBFT), to build BFT systems.

RBFT AND AARDVARK

- RBFT considers performance during both:
  - Gracious intervals: Network is synchronous, replicas are timely and fault-free, clients are correct.
  - Uncivil intervals: Network links and correct servers are timely but upto f = (n-1)/3 servers and any number of clients are faulty.

- Aardvark is sometimes similar to traditional BFT protocols:
  - Client sends request to primary, which then relays it to the replicas who agree on the sequence of requests.

RAFT AND AARDVARK

- Aardvark is not similar when considering the following points:
  - It utilizes signatures for authentication, though, eliminating signatures and using MAC's eliminates the main performance bottleneck.
  - It performs regular view changes, though this renders the system unavailable for a short span of time.
  - It utilizes point-to-point communication, though, using IP-multicast boosts the performance significantly.
- Aardvark is designed on an approach which does not compromise safety and refocuses the design and implementation on the stress that failures can impose on the system.

RAFT AND AARDVARK

- Surprisingly, this only impacts the peak performance of Aardvark modestly.
- At the same time, the fault tolerance gets improved significantly (As shown in Table 1).
- For a broad range of client, primary and server misbehaviours, the performance remains within a constant factor of its best case performance.
- Hence, the following are the main contributions of this paper:
  - Demonstrate existing protocols and their implementations are fragile.
  - An argument to convey BFT protocols should be designed with focus on robustness.
  - To prove that, using Aardvark, RBFT approach is viable.

SYSTEM MODEL

- The assumptions made for the system model are:
  - The faulty nodes (servers or clients) can behave arbitrarily.
  - A strong adversary can coordinate faulty nodes to compromise the replicated service.
  - The adversary, however, cannot break cryptographic techniques like collision-resistant hashing, message authentication codes (MACs), encryption, and signatures.
  - There are a finite number of clients, any number of clients can be faulty.
  - However, a system's safety and liveness are guaranteed only if at most f = (n-1) / 3 servers are faulty.
  - An asynchronous network is assumed where synchronous intervals occur infinitely often.

Synchronous intervals: During a synchronous interval any message sent between correct processes is delivered within a bounded delay T if the sender retransmits according to some schedule until it is delivered.

RECASTING THE PROBLEM

- The foundation of modern BFT state machine replication rests on an impossibility result and two principles that assist us in dealing with it.
  - The impossibility result: no solution to consensus can be both safe and live in an asynchronous systems if nodes can fail.
  - The two principles:
    - Synchrony must not be needed for safety: As long as a threshold of faulty servers is not exceeded, the replicated service must always produce linearizable executions, independent of whether the network loses, reorders, or arbitrarily delays messages.
    - Given FLP, synchrony must play a role in liveness: clients are guaranteed to receive replies to their requests only during intervals in which messages sent to correct nodes are received within some fixed (but potentially unknown) time interval from when they are sent.

RECASTING THE PROBLEM

- The engineering of BFT protocols follow this well-known recommendation from Lampson's:

" 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. "

- Ever since the design of PBFT, the design of BFT systems follow a predictable pattern:
  - Characterize what is normal
  - Pull out all stops to make the system perform well for that case.

RECASTING THE PROBLEM

- The results of following the above approaches continue to be excellent.
- Despite the good results, the design of BFT systems for the best case of gracious execution is increasingly misguided, dangerous and futile.
- Though the current systems survive Byzantine faults without compromising on safety, this paper contends that the system can be made completely unavailable by a simple Byzantine failure.
- Such systems can hardly be said to tolerate Byzantine faults.

RECASTING THE PROBLEM

- A BFT system fulfils its obligations when provides acceptable and dependable performance across the broadest possible set of executions, including executions with Byzantine clients and servers.
- A BFT system should be designed around an execution path that has three properties:
  - It provides acceptable performance
  - It is easy to implement
  - It is robust against Byzantine attempts to push the system away from it.
- This paper proposes to build an RBFT system that provides adequate performance even during uncivil executions.
- This approach reduces the best case performance, but it is preferable compared to the devastating loss of availability.

AARDVARK: RBFT IN ACTION

- Aardvark is a new BFT system designed and implemented to be robust to failures.
- The Aardvark protocol consists of 3 stages:
  - Client request transmission,
  - Replica agreement,
  - Primary view change.
- To avoid pitfalls of fragile implementations, it is focused in each stage how a faulty node can be prevented from limiting the other nodes to perform correctly.

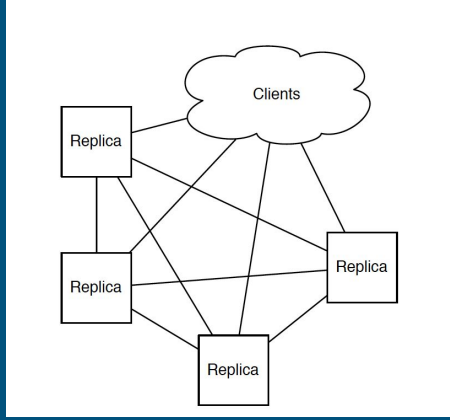
AARDVARK: RBFT IN ACTION

- This systematic methodology leads to 3 main design differences between Aardvark and previous BFT systems:
  - **Signed client requests**
  - **Resource isolation**
  - **Regular view changes.**

AARDVARK: RBFT IN ACTION

- Signed Client Requests:
  - Aardvark clients use digital signatures to authenticate their requests. This provides non-repudiation and ensures that all replicas make identical decisions, hence eliminating many tricky and expensive corner case conditions found in existing protocols that use message authentication code (MAC).
  - Digital signatures are expensive to use, hence used only for authenticating client requests.
  - Primary-to-replica, replica-to-replica, and replica-to-client communication rely on MAC authenticators.
  - The quorum-driven nature of server-initiated communication ensures that a single faulty replica is unable to force the system into undesirable execution paths.
  - Aardvark has to guard against denial-of-service attacks where the system receives a large numbers of requests with signatures that need to be verified.
  - Solution: Use of hybrid MAC-signature construct and forcing a client to finish one request at a time.

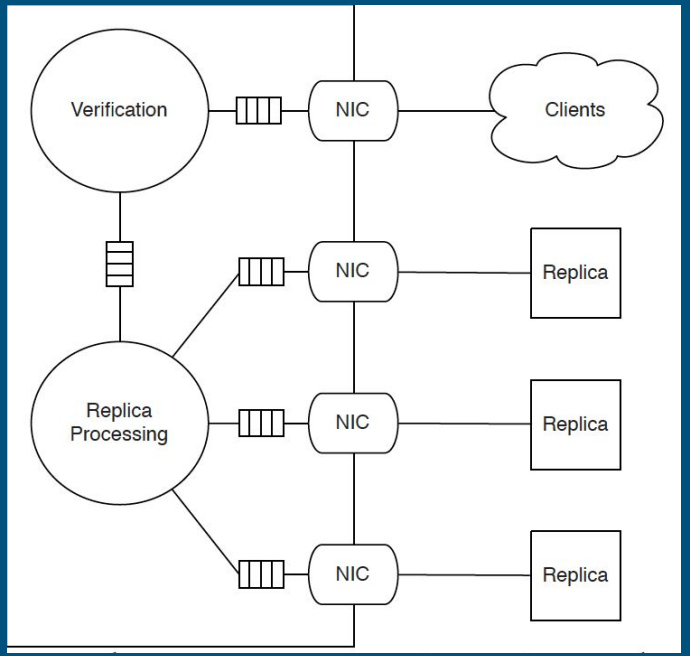
AARDVARK: RBFT IN ACTION



- Resource Isolation:
  - Aardvark isolates network and computational resources.
  - It uses separate network interface controllers (NICs) and wires to connect each pair of replicas.
  - This step prevents a faulty server from interfering with the timely delivery of messages from good servers.
  - It also allows a node to defend itself against brute force denial of service attacks by disabling the offending NIC.
  - This incurs a performance hit, as Aardvark can no longer use hardware multicast to optimize all-to-all communication.

AARDVARK: RBFT IN ACTION

- Resource Isolation:
  - Aardvark uses separate work queues for processing messages from clients and individual replicas.
  - Employing a separate queue for client requests prevents client traffic from drowning out the replica-to-replica communications required for the system to make progress.
  - Employing a separate queue for each replica allows Aardvark to schedule message processing fairly, ensuring that a replica is able to efficiently gather the quorums it needs to make progress.
  - Aardvark can also easily leverage separate hardware threads to process incoming client and replica requests.



Architecture of a Single Replica

AARDVARK: RBFT IN ACTION

- Regular View Changes
  - To prevent a primary from achieving tenure and exerting absolute control on system throughput, Aardvark invokes the view change operation.
  - Replicas monitor the performance of the current primary, slowly raising the level of minimal acceptable throughput. If the current primary fails to provide the required throughput, replicas initiate a view change.
  - The key properties of this technique are:
    - During uncivil intervals, system throughput remains high even when replicas are faulty. Since a primary maintains its position only if it achieves some increasing level of throughput, Aardvark bounds throughput degradation caused by a faulty primary by either forcing the primary to be fast or selecting a new primary.
    - As in prior systems, eventual progress is guaranteed when the system is eventually synchronous.

PROTOCOL DESCRIPTION

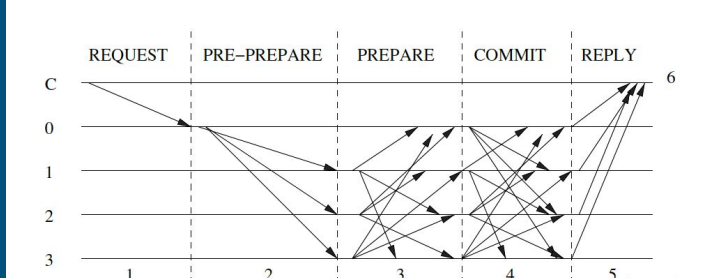


Figure 3: Basic communication pattern in Aardvark.

- Figure 3 shows the agreement phase communication pattern that Aardvark shares with PBFT.
- The topics below discuss the numbered steps in the above figure:
- **Client Request Transmission:**
  - The fundamental challenge in transmitting client requests is ensuring that, upon receiving a client request, every replica comes to the same conclusion about the authenticity of the request.
  - This property is ensured by having clients sign requests.
  - To guard against denial of service, the processing of a client request is broken into a sequence of increasingly expensive steps. Each step serves as a filter, so that more expensive steps are performed less often.
  - Aardvark dedicates a single NIC to handling incoming client requests so that incoming client traffic does not interfere with replica-to-replica communication.

PROTOCOL DESCRIPTION

- Protocol Description
  - The steps taken by a replica to authenticate a client request:
    - A client c requests an operation o be performed by the replicated state machine by sending a request message to the replica p it believes to be the primary.

$$<<\text{REQUEST}, o, s, c>\sigma_c, c>\mu_{c,p}$$

  - If the client does not receive a timely response to that request, then the client retransmits the request to all replicas r.

$$<<\text{REQUEST}, o, s, c>\sigma_c, c>\mu_{c,r}$$

  - The request contains the client sequence number s and is signed with signature σ_c . The signed message is then authenticated with a MAC $\mu_{c,r}$ for the intended recipient.

PROTOCOL DESCRIPTION

a) Blacklist check: If the sender c is not blacklisted, then proceed to step (b). Otherwise discard the message.

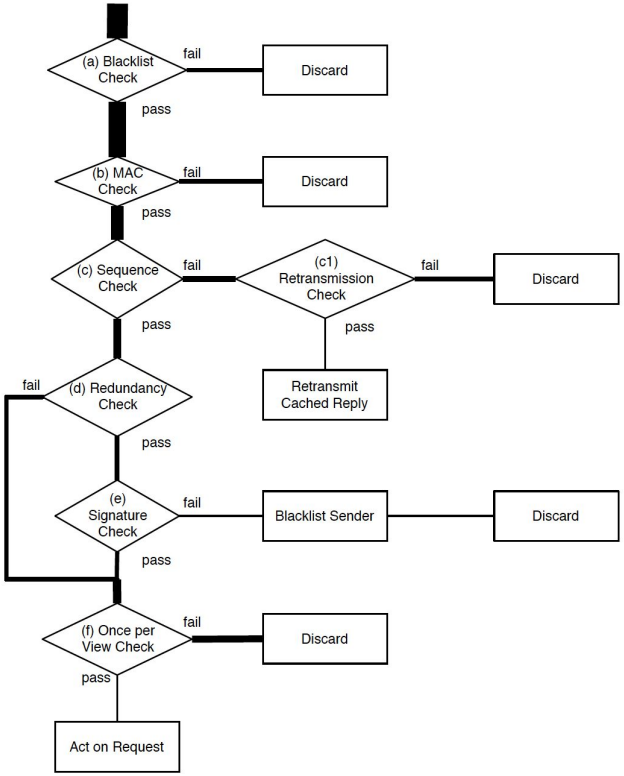
b) MAC check: If $\mu_{c,p}$ is valid, then proceed to step (c). Otherwise discard the message.

c) Sequence check: Examine 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). Otherwise,

   i) (c1) Retransmission check: If a reply has not been sent to c recently, then retransmit the last reply sent to c. Otherwise discard the message.

d) Redundancy check: Examine the most recent cached request from c. If no request from c with sequence

e) number s_{req} has previously been verified or the request then proceed to (e), otherwise, to (f).

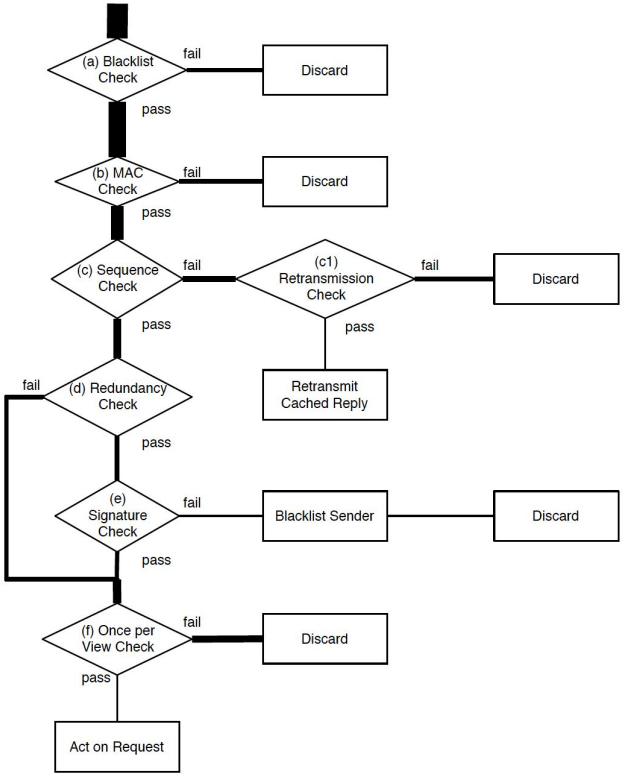


Decision Tree followed by replicas while authenticating client requests

PROTOCOL DESCRIPTION

e) Signature check: If σ_c is valid, then proceed to step Additionally, if the request does not match the previously cached request for s_{req}, then blacklist c. Otherwise if σ_c is not valid, then blacklist the node x that authenticated $\mu_{x,p}$ and discard the message.

(f) Once per view check: If an identical request has been verified in a previous view, but not processed during the current view, then act on the request. Otherwise discard the message.



Decision Tree followed by replicas while authenticating client requests

PROTOCOL DESCRIPTION

- Primary and non-primary replicas act on requests in different ways.
- A primary adds requests to a PREPREPARE message that is part of the three-phase commit protocol.
- A non-primary replica r processes a request by authenticating the signed request with a MAC $\mu_{r,p}$ for the primary p and sending the message to the primary.
- A node p only blacklists a sender c of a REQUEST message if the MAC $\mu_{c,p}$ is valid but the signature σ_c is not.

PROTOCOL DESCRIPTION

- Resource Scheduling:
  - Aardvark leverages separate work queues for providing client requests and replica-to-replica communication to limit the fraction of replica resources that clients are able to consume.
  - This ensuring that a flood of client requests is unable to prevent replicas from making progress on requests already received.

PROTOCOL DESCRIPTION

- Replica Agreement:
  - Once a request has been transmitted from the client to the current primary, the replicas must agree on the request's position in the global order of operations.
  - Aardvark replicas coordinate with each other using a standard three phase commit protocol.
  - The fundamental challenge in the agreement phase is ensuring that each replica can quickly collect the quorums of PREPARE and COMMIT messages necessary to make progress.
  - Conditioning expensive operations on
  - the gathering of a quorum of messages ensures robustness in two ways.
    - First, it is possible to design the protocol so that incorrect messages sent by a faulty replica will never gain the support of a quorum of replicas.
    - Second, as long as there exists a quorum of timely correct replicas, a faulty replica that sends correct messages too slowly, or not at all, cannot impede progress.
  - A correct replica that has fallen behind its peers may ask them for the state it is missing by sending them a *catchup message*.

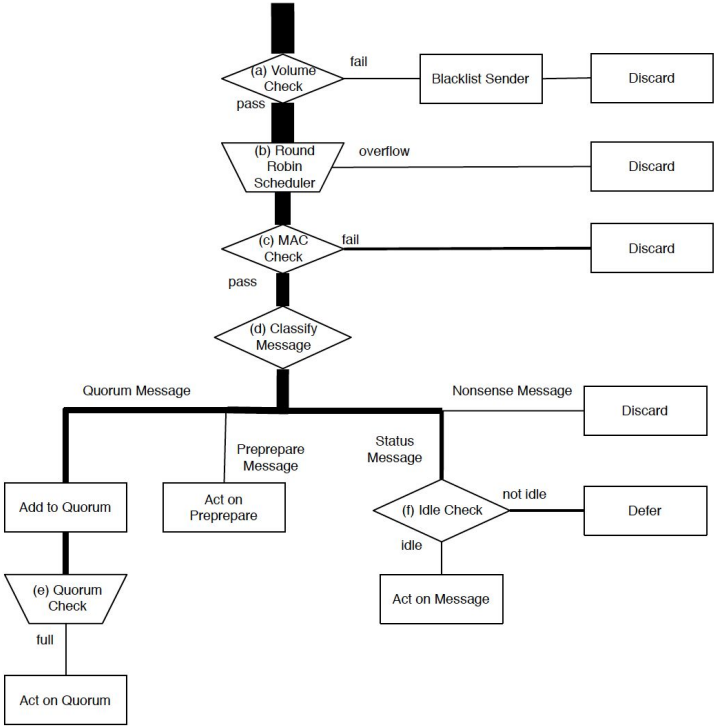
PROTOCOL DESCRIPTION

- The agreement protocol requires replica-to-replica communication. A replica r filters, classifies, and finally acts on the messages it receives from another replica according to the decision tree shown.

a) Volume Check: If replica q is sending too many messages, blacklist q and discard the message. Otherwise continue to step (b).

b) Round-Robin Scheduler: Among the pending messages, select the the next message to process from the available messages in round-robin order based on the sending replica .

c) MAC Check. If the selected message has a valid MAC, then proceed to step (d) otherwise, discard the message.



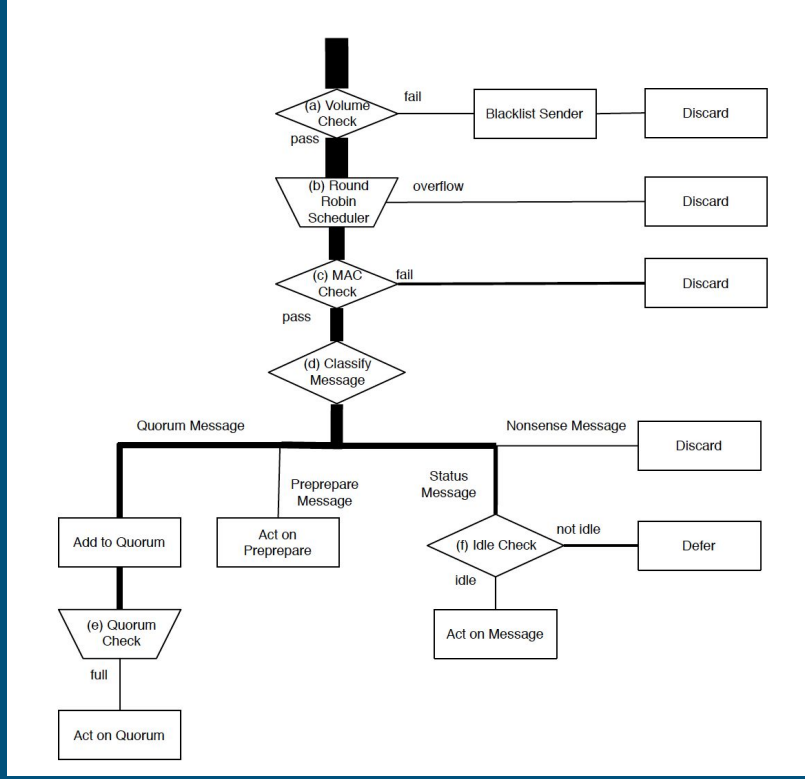
Decision Tree followed by replicas when handling messages received from another replica.

PROTOCOL DESCRIPTION

d) Classify Message: Classify the authenticated message according to its type:

- If the message is PRE-PREPARE, then process it immediately as in agreement protocol 3.
- If the message is PREPARE or COMMIT, then add it to the appropriate quorum and proceed to step (e).
- If the message is a catchup message, then proceed to step (f).
- If the message is anything else, then discard the message.

e) Quorum Check. If the quorum to which the message was added is complete, then act as appropriate in the protocol steps 4-6.

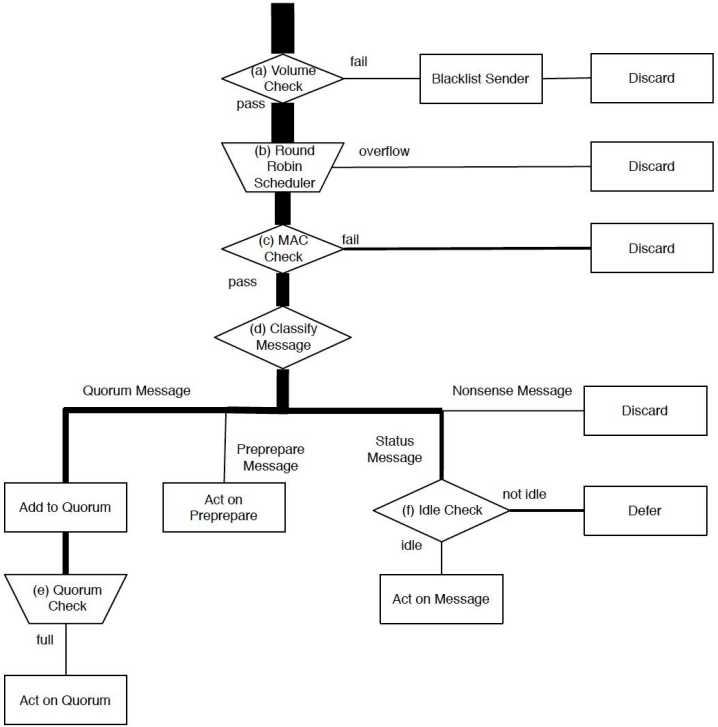


Decision Tree followed by replicas when handling messages received from another replica.

PROTOCOL DESCRIPTION

f) Idle Check: If the system has free cycles, then process the catchup message. Otherwise, defer processing until the system is idle.

- Replica r applies the above steps to each message it receives from the network.
- Once messages are appropriately filtered and classified, the agreement protocol continues.



Decision Tree followed by replicas when handling messages received from another replica.

PROTOCOL DESCRIPTION

- Catchup messages
  - If replica r receives a catchup message from a replica q that has fallen behind, then r sends q the state that q to catch up and resume normal operations.
  - Sending catchup messages is vital to allow temporarily slow replicas to avoid becoming permanently non-responsive, but it also offers faulty replicas the chance to impose significant load on their non-faulty counterparts.

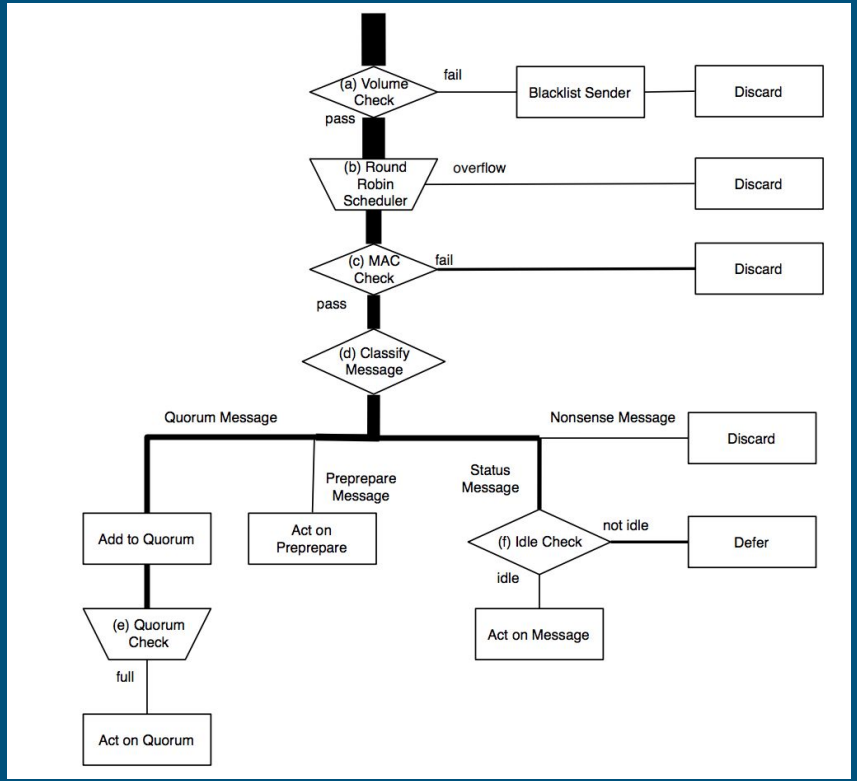
PROTOCOL DESCRIPTION

Step 1	Step 2	Step 3	Step 4	Step 5
Client sends a request to replica	Primary forms a PRE-PREPARE message containing a set of valid requests and sends the PRE-PREPARE to all replicas.	Replica receives PRE-PREPARE from the primary, authenticates the PRE-PREPARE , and sends a PREPARE to all other replicas.	Replica receives 2f PREPARE messages that are consistent with the PREPREPARE message for sequence number n and sends a COMMIT message to all other replicas.	Replica receives 2f + 1 COMMIT messages, commits and executes the request, and sends a REPLY message to the client.

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

AARDVARK AND RBFT

- Faulty replicas cannot use catchup messages to interfere with the processing of other messages because correct replica processes catchup messages only when otherwise idle
- A faulty replica could simply bombard its correct peers with a high volume of messages. The round-robin scheduler limits the damage that can result from this attack.

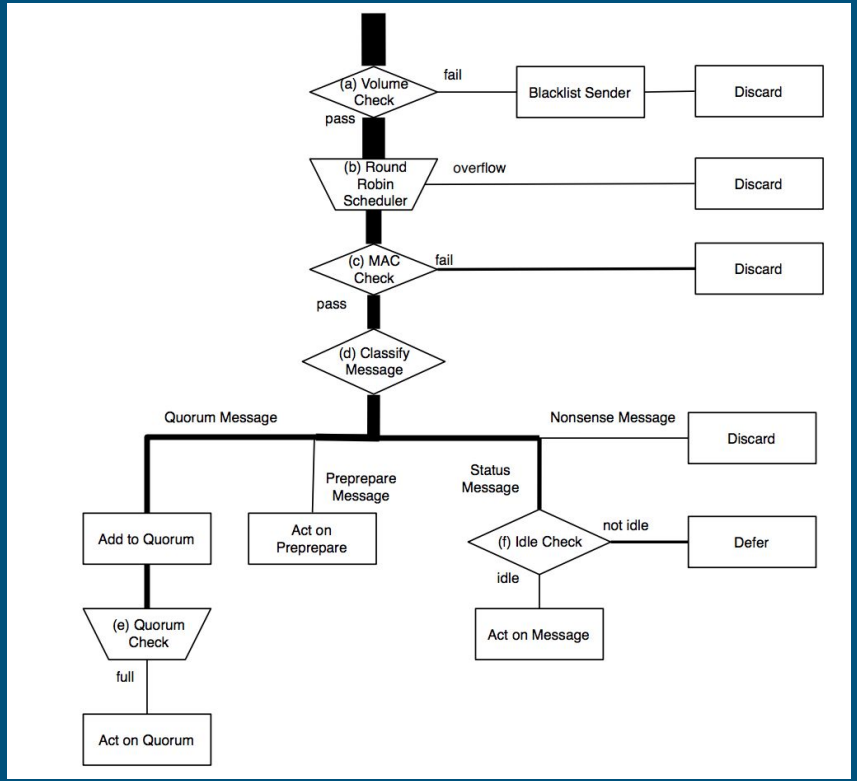


AARDVARK AND RBFT

If c of its peers have pending messages, then a correct replica wastes at most 1/c of the cycles spent checking MACs and classifying messages.

The round-robin scheduler also discards messages that overflow a bounded buffer

The volume check (a) similarly limits the rate at which a faulty replica can inject messages



AARDVARK- PRIMARY VIEW CHANGES

- Primary can delay processing requests, discard requests, corrupt clients' MAC authenticators, introduce gaps in the sequence number space, unfairly delay or drop some clients' requests but not others
- To tackle this, Aardvark uses the same view change mechanism described in PBFT. This ensures:

1. Adaptive throughput
2. Fairness

AARDVARK- ADAPTIVE THROUGHPUT

- Each replica of Primary starts a heartbeat timer that is reset whenever the next valid PRE-PREPARE message is received
- If the replica does not receive the next valid PRE-PREPARE message before the heartbeat timer expires, the replica initiates a view change
- A correct replica doubles the heartbeat interval each time the timer expires
- If the observed throughput in the interval between two successive checkpoints falls below a specified threshold,, the replica initiates a view change to replace the current primary

AARDVARK- FAIRNESS

- A faulty primary could be unfair to a set of clients by neglecting to order requests from that client.
- When a replica receives from a client a request that it has not seen in a PRE-PREPARE message, it adds the message to its request queue
- Before forwarding the request to the primary, it records the sequence number k of the most recent PRE-PREPARE received during the current view.
- It monitors future PRE-PREPARE messages for that request, and if it receives two PRE-PREPAREs for another client before receiving a PREPARE for client c, then it declares the current primary to be unfair and initiates a view change

PERFORMANCE EVALUATIONS

The performance of Aardvark, PBFT, HQ, Q/U and Zyzzyva is evaluated on an Emulab cluster

The following observations are made during the evaluations:

1. Existing systems are vulnerable to significant disruption as a result of a broad range of Byzantine behaviors
2. Aardvark is robust to a wide range of Byzantine behaviors.
3. Despite utilizing signatures and change views Aardvark’s peak throughput is competitive with that of existing systems

PERFORMANCE: FAULTY CLIENTS

- In HQ, our intended attack is to have clients send certificates with an inconsistent MAC authenticator.
- In Q/U, the clients issue contending requests to the replicas.
- In Aardvark, the closest analogous client behaviors to those discussed above for other systems are sending requests with a valid MAC and invalid signature or sending requests with invalid MACs.
- Both attacks are implemented and the results are comparable with state-of-the-art systems

PERFORMANCE: NETWORK FLOODING

- Here, correct clients issue requests sufficient to saturate each system while a single faulty client implements a brute force denial of service attack by repeatedly sending 9KB UDP messages to the replicas
- The PBFT and Zyzzyva prototypes suffer dramatic performance degradation while Q/U and HQ suffer smaller degradations
- In the case of Aardvark, a faulty client is unable to prevent replicas from processing requests that have

already entered the system.

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	-

PERFORMANCE: FAULTY PRIMARY

- The impact on PBFT, Zyzzyva, and Aardvark prototypes of a primary that delays sending PRE-PREPARE messages by 1, 10, or 100 ms are shown below:

System	Peak Throughput	1 ms	10 ms	100 ms
PBFT	61710	5041	4853	1097
Zyzzyva	65999	27776	5029	crash
Aardvark	38667	38542	37340	37903

- Throughput of both PBFT and Zyzzyva degrades dramatically. Zyzzyva eventually succumbs to a memory leak
- throughput achieved by Aardvark indicates that adaptively performing view changes in response to observed throughput is a good technique for ensuring performance

PERFORMANCE: FAULTY PRIMARY

- Primary is also responsible for controlling which requests are inserted into the system
- For Zyzzyva, the unfair primary ignores messages from the targeted client entirely. The resulting throughput is 0

System	Starved Throughput	Normal Throughput
PBFT	1.25	1446
Zyzzyva	0	1718
Aardvark	358	465

- Aardvark's fairness detection and periodic view changes limit the impact of the unfair primary
- The primary sends a PRE-PREPARE for the targeted client's request after receiving the the request 9 times.

- This prevents the PBFT primary from triggering a view change and demonstrates dramatic degradation in throughput