

Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults

Xiaotian Zou, Lei Chu

Outline

- Introduction
- System Model
- Problem Recasting
- Basic Structure
- Protocol description
- Analysis
- Evaluation

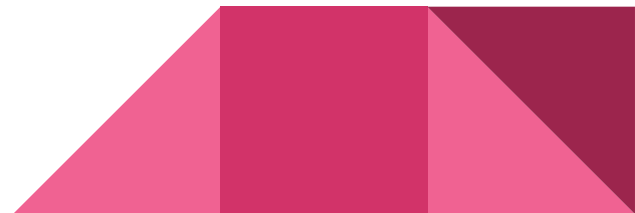


Introduction

1) Current shortcomings:


- Can't tolerate Byzantine faults very well

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



Introduction

2) New approach Robust BFT (RBFT):

- Shift the focus from maximizing the performance of the best case to providing acceptable and predictable performance under the broadest possible set of circumstances—including when faults occur
 - Focus both the design of the system and the engineering choices involved in its implementation on the stress that failures can impose on performance.
- 

System model

- No restriction on clients, at most $f=(n-1)//3$ servers are faulty.
- Faulty nodes (servers or clients) can behave arbitrarily.
- The adversary cannot break cryptographic techniques like MACs
- Asynchronous network with synchronous intervals



Recasting the problem

- Foundations of modern BFT state machine replication:
 - an impossibility result
 - two principles
 - synchrony must not be needed for safety
 - synchrony must play a role in liveness
- The normal case must be fast. The worst case must make some progress



Recasting the problem

The design of BFT is misguided:

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

They provide impressive throughput but weak liveness guarantees in the presence of Byzantine failures

A System that can be made completely unavailable by a simple Byzantine failure can hardly be said to tolerate Byzantine faults



Recasting the problem

New design ideas:

- it provides acceptable performance
- it is easy to implement
- it is robust against Byzantine attempts to push the system away from it



Basic Structure

Aardvark protocol contains 3 stages:

- Client request transmission
- Replica agreement
- Primary view change



Basic Structure

There are 3 differences from the previous BFT systems:

- Signed client requests
- Resource allocation
- Regular view changes



Basic Structure

Signed client requests:

- Current BFT systems mainly use message authentication code(MAC) to authenticate client requests, which could be faster but does not provide non-repudiation property.
- Aardvark clients use digital signatures to authenticate their requests.



Basic Structure

Using Digital signatures takes longer time than MAC:

- Aardvark servers only do verification operations.
- There could be potential denial-of-service(DoS) attacks. To avoid this, Aardvark
 - utilizes a hybrid MAC-signature mechanism to limit the number of faulty signature verifications a client can have
 - forces a client to complete one request before issuing the next

Basic Structure

Resource allocation:

- Aardvark uses separate network interface controllers(NICs) and wires to connect each pair of replicas, preventing a single broken NIC from shutting down the whole system.
- Therefore, aardvark cannot use hardware multicast to optimize all-to-all communications.

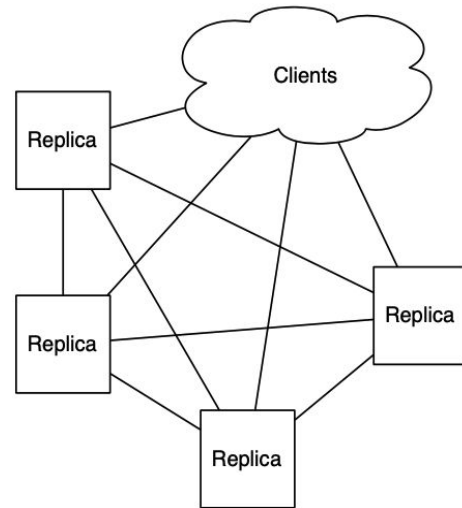
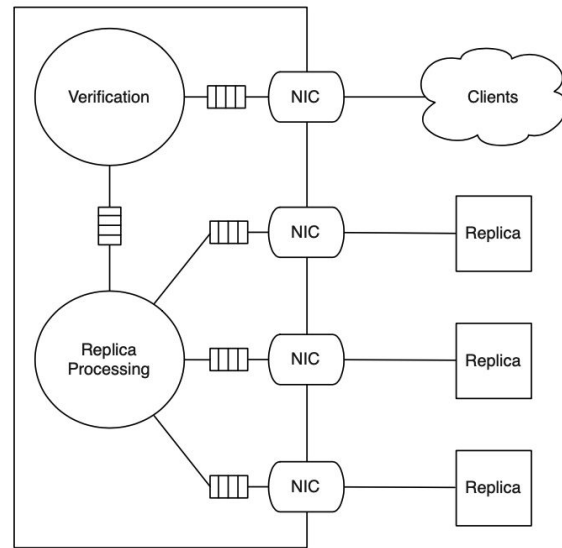


Figure 1: Physical network in Aardvark.

Basic Structure

- A separate queue for client requests prevents client from flooding the replica-to-replica communications
- A separate work queue for each replica allows to schedule message processing fairly

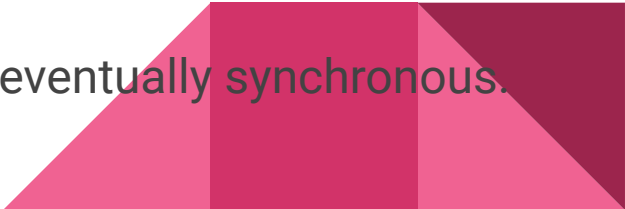


Basic Structure

Regular View Changes:

- This is operated on a regular basis.
- Replicas monitor the performance of the current primary and slowly increase the level of the minimal throughput threshold.

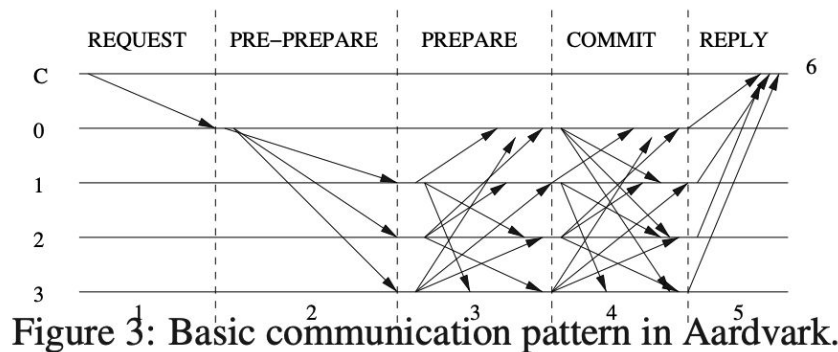
Key properties:

1. During uncivil intervals, system throughput remains high even when replicas are faulty.
 2. Eventual progress is guaranteed when the system is eventually synchronous.
- 

Protocol description

For this part we will mainly discuss these following points:

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



Protocol description

To guard against DoS attacks, the processing of a client request is broken into a sequence of increasingly expensive steps.

Details will be discussed in the next few slides.



Protocol description

1. Client sends a request to a replica.

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

o	operation
s	Sequence number
c	client
σ_c	signature
$\mu_{c,p}$	MAC

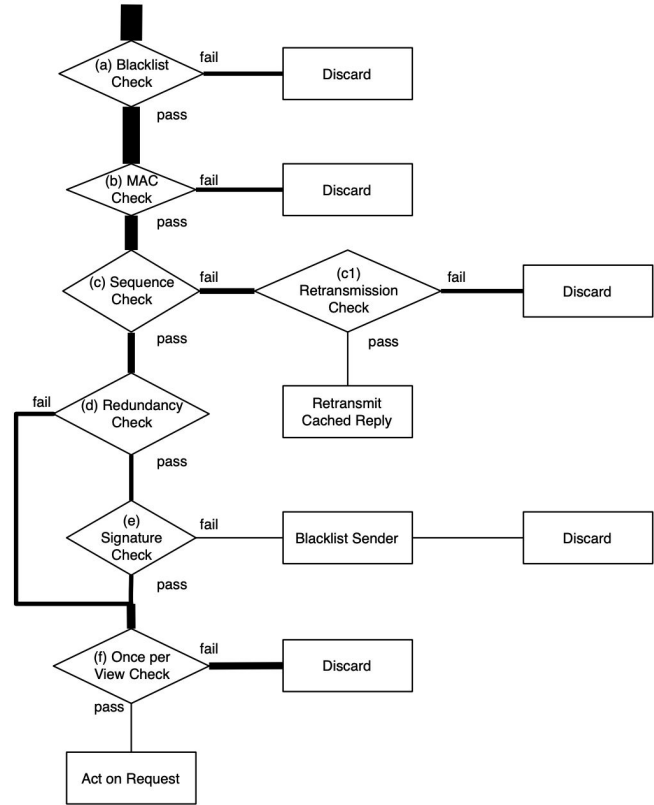
Protocol description

- The client will send a request to the primary node.
- If no response before timeout, the client retransmits the request to all replicas r.



Protocol description

- After receiving a client request, a replica will verify it by following the sequence of steps.



Protocol description

a) Blacklist check

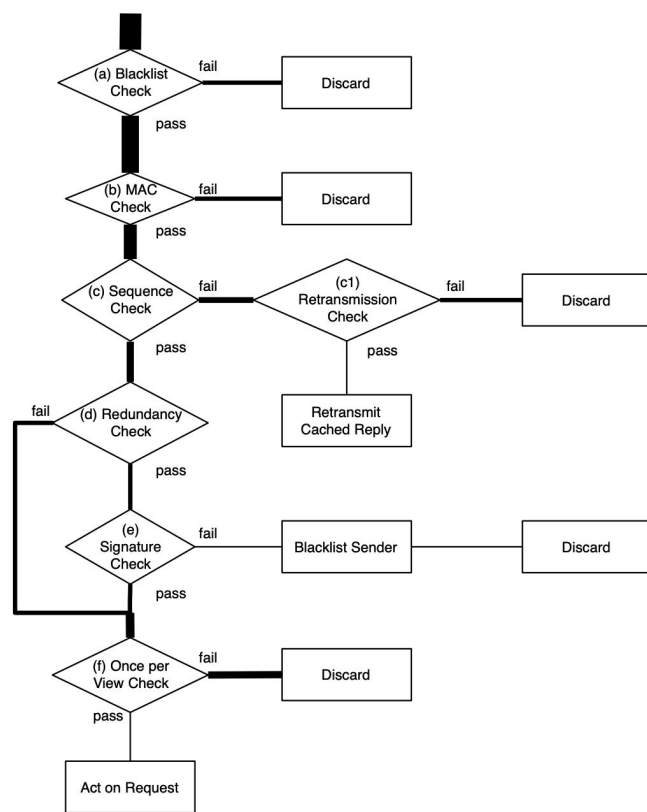
b) MAC check

c) Sequence Check

- Examine the most recently cached reply to c with seq number s_{cache} . If $s_{\text{req}} = s_{\text{cache}} + 1$, continue to step d)

c1) Retransmission Check

- Each replica uses an exponential back off to limit the rate of client reply retransmission. If a reply has not been sent to c recently, then retransmit the last reply sent to c



Protocol description

d) Redundancy check

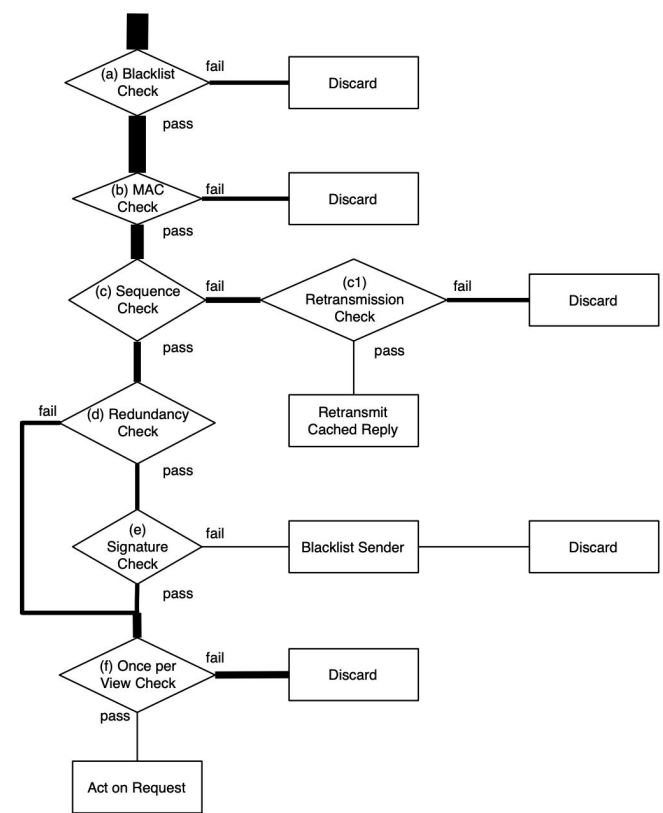
- Check if the current s_{req} has been verified already

e) Signature check

- If the signature is incorrect, blacklist the sender

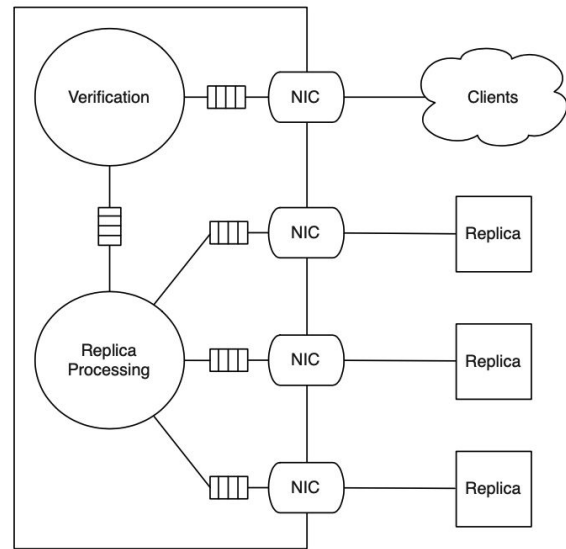
f) Once per view check

- If an identical request has been verified in a previous view, but not processed during the current view, then act on the request



Protocol description

- Separate work queues can ensure the client requests won't be able to flood the replicas.
- The prototype is runned on the dual core machine, one core is used for verification and the other is for replica processing.



Protocol description

Replica agreement

- Challenge: To quickly collect the quorums of PREPARE and COMMIT messages.
- Solution: designing a reasonable agreement protocol



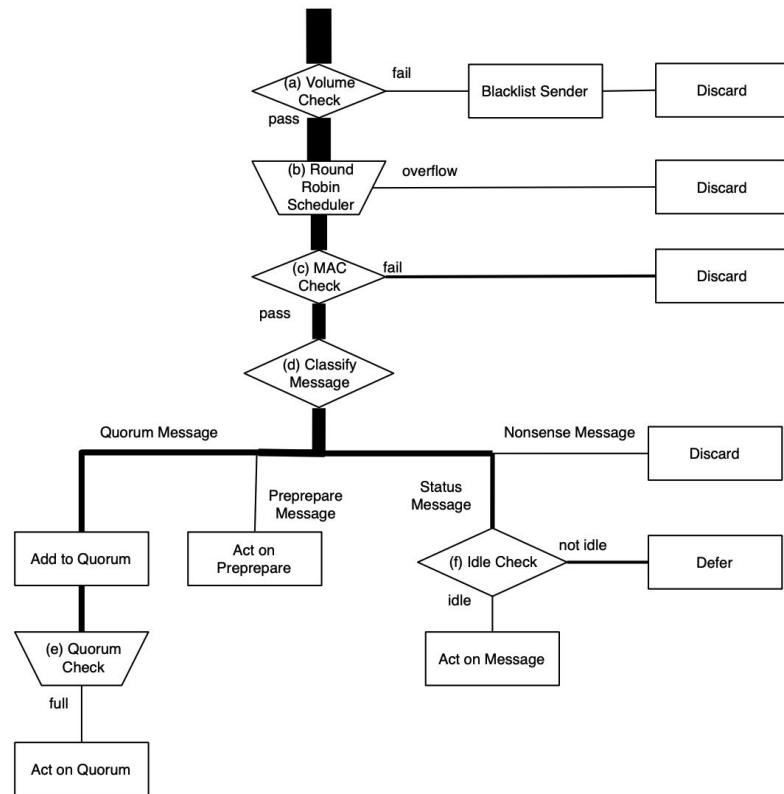
Protocol description

a) Volume check

- If replica q is sending too many messages, then it would be blacklisted.
- Separate NICs can be used to silence the malicious replica
- After disconnecting q, r reconnects q after 10 mins, or when f other replicas are also disconnected for flooding.

b) Round-Robin Scheduler

- Select the next message to process from the currently available messages in round-robin order based on the sending replica
- Discard the message if buffer is full



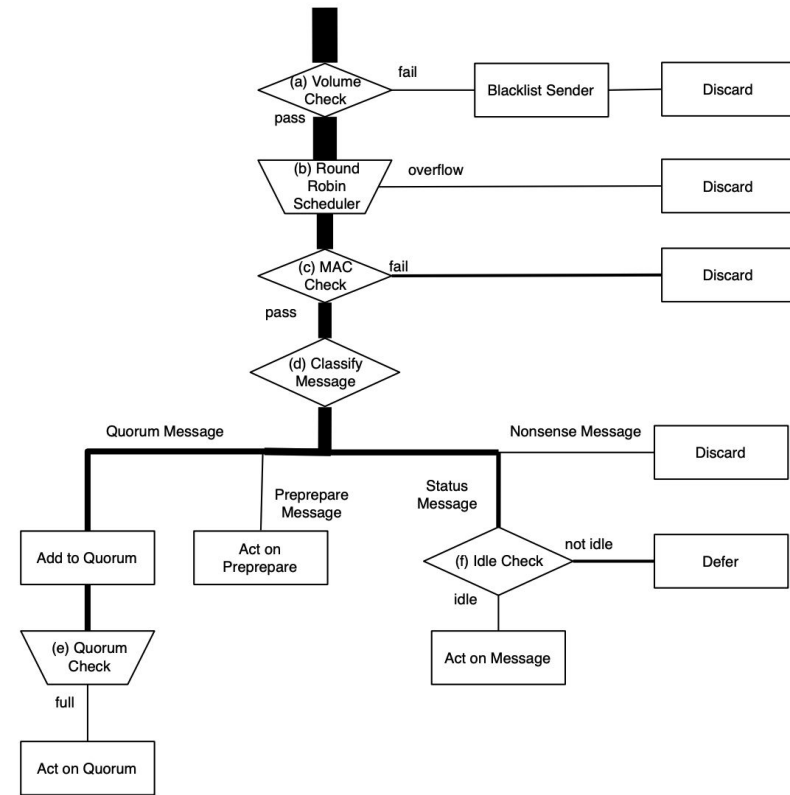
Protocol description

c) MAC check

d) message classification

e) Quorum check

f) Idle check



Protocol description

The following phases:

- Once the messages are filtered and classified, primary forms a PRE-PREPARE message containing a set of valid requests and sends it to all replicas.
- Replica receives PRE-PREPARE from the primary, authenticates it and sends a PREPARE to the rest of the replicas
- Replica receives $2f$ PREPARE messages whose sequence number is consistent with those in the PRE-PREPARE message, then it sends a COMMIT to all other replicas



Protocol description

The following phases:

- Replica receives $2f + 1$ COMMIT, commits and executes the request, sends a REPLY to the client
- The client receives $f + 1$ matching REPLY messages and the request is complete.



Protocol description

Primary view change

- Challenge: faulty primary node could hurt performance in a wide range of ways.
- Solution: regular view changes



Protocol description

- After a view change is completed, heartbeat timer for every replica will be reset whenever the next PRE-PREPARE message is received.
- If one timer expires, a new view change will be initiated by this replica.
- If the observed throughput between 2 checkpoints is below a threshold, a view change will be initiated.
- A view change won't influence the overall performance.



Analysis

- Restrict our attention to an Aardvark implementation on a single-core machine with a processor speed of κ GHz.
- verifying signatures, generating MACs, and verifying MACs, requiring θ , α , and α cycles, respectively




Analysis

Theorem 1. *Consider a gracious view during which the system is saturated, all requests come from correct clients, and the primary generates batches of requests of size b . Aardvark's throughput is then at least $\frac{\kappa}{\theta + \frac{(4n-2b-4)}{b}\alpha}$ operations per second.*

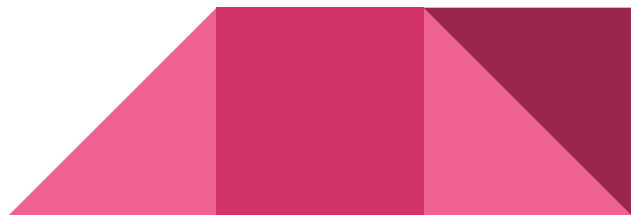
Analysis

Theorem 2. *Consider an uncivil view in which the primary is correct and at most f replicas are Byzantine. Suppose the system is saturated, but only a fraction of the requests received by the primary are correct. The throughput of Aardvark in this uncivil view is within a constant factor of its throughput in a gracious view in which the primary uses the same batch size.*



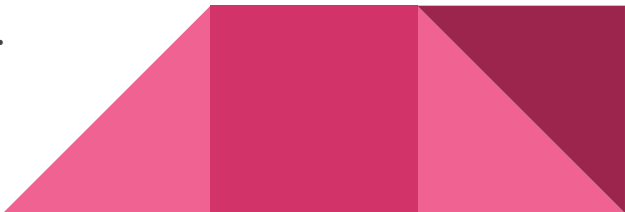
Analysis

Theorem 3. *For sufficiently long uncivil executions and for small f the throughput of Aardvark, when properly configured, is within a constant factor of its throughput in a gracious execution in which primary replicas use the same batch size.*



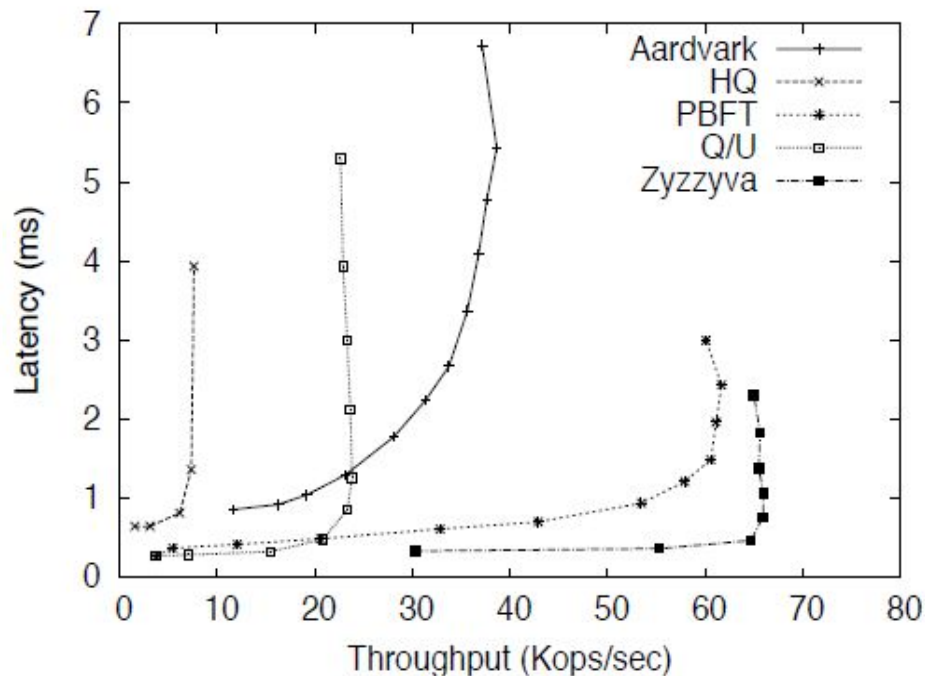
Evaluation

Three points:

- Despite our choice to use signatures, change views regularly, and forsake IP multicast, Aardvark's peak throughput is competitive with that of existing systems
 - Existing systems are vulnerable to significant disruption as a result of a broad range of Byzantine behaviors
 - Aardvark is robust to a wide range of Byzantine behaviors. When evaluating existing systems, we attempt to identify places where the prototype implementation departs from the published protocol.
- 

Evaluation

Aardvark's peak throughput is competitive



Evaluation

Aardvark incorporates several key design decisions that enable it to perform well in the presence of Byzantine failure.

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



Evaluating faulty systems

Evaluate Aardvark and existing systems in the context of failures

Two aspects of client behavior:

- Request dissemination
- Network flooding



Evaluating faulty systems

Request dissemination:

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



Evaluating faulty systems

Network flooding:

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



Faulty Primary

In systems that rely on a primary, the primary controls the sequence of requests that are processed during the current view.

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

Non-Primary Replicas

implement a faulty replica that fails to process protocol messages and instead blasts network traffic at the other replicas and show the results in Table.

System	Peak Throughput	Replica Flooding	
		UDP	TCP
PBFT	61710	251	-
Q/U	23850	19275	crash
HQ	7629	crash	crash
Zyzzyva	65999	0	-
Aardvark	38667	11706	-

Thank You!

