Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults

Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin
The University of Texas at Austin

Mirco Marchetti
The University of Modena and Reggio Emilia

Abstract

This paper argues for a new approach to building Byzantine fault tolerant replication systems. We observe that although recently developed BFT state machine replication protocols are quite fast, they don't tolerate Byzantine faults very well: a single faulty client or server is capable of rendering PBFT, Q/U, HQ, and Zyzzyva virtually unusable. In this paper, we (1) demonstrate that existing protocols are dangerously fragile, (2) define a set of principles for constructing BFT services that remain useful even when Byzantine faults occur, and (3) apply these principles to construct a new protocol, Aardvark. Aardvark can achieve peak performance within 40% of that of the best existing protocol in our tests and provide a significant fraction of that performance when up to f servers and any number of clients are faulty. We observe useful throughputs between 11706 and 38667 requests per second for a broad range of injected faults.

1 Introduction

This paper is motivated by a simple observation: although recently developed BFT state machine replication protocols have driven the costs of BFT replication to remarkably low levels [1, 8, 12, 18], the reality is that they don't tolerate Byzantine faults very well. In fact, a single faulty client or server can render these systems effectively unusable by inflicting multiple orders of magnitude reductions in throughput and even long periods of complete unavailability. Performance degradations of such degree are at odds with what one would expect from a system that calls itself Byzantine fault tolerant—after all, if a single fault can render a system unavailable, can that system truly be said to tolerate failures?

To illustrate the the problem, Table 1 shows the measured performance of a variety of systems both in the absence of failures and when a single faulty client submits a carefully crafted series of requests. As we show later, a wide range of other behaviors—faulty primaries, recovering replicas, etc.—can have a similar impact. We believe that these collapses are byproducts of a single-minded focus on designing BFT protocols with ever more impressive best-case performance. While this focus is understandable—after years in which BFT replication was dismissed as too expensive to be practical, it was important to demonstrate that high-performance BFT is not an oxymoron—it has led to protocols whose complexity undermines robustness in two ways: (1) the protocols' *design* includes *fragile optimizations* that allow a faulty client or server to knock the system off of the optimized execution path to an expensive alternative path and (2) the protocols' *implementation* often fails to handle properly all of the intricate corner cases, so that the implementations are even more vulnerable than the protocols appear on paper.

The primary contribution of this paper is to advocate a new approach, *robust BFT* (RBFT), to building BFT systems. Our goal is to change the way BFT systems are designed and implemented by shifting the focus from constructing high-strung systems that maximize best case performance to constructing systems that offer acceptable and predictable performance under the broadest possible set of circumstances—including when faults occur.

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

Table 1: Observed peak throughput of BFT systems in a fault-free case and when a single faulty client submits a carefully crafted series of requests. We detail our measurements in Section 7.2. † The result reported for Q/U is for correct clients issuing conflicting requests. ‡ The HQ prototype demonstrates fault-free performance and does not implement many of the error-handling steps required to handle inconsistent MACs.

RBFT explicitly considers performance during both *gracious* intervals—when the network is synchronous, replicas are timely and fault-free, and clients correct—and *uncivil* execution intervals in which network links and correct servers are timely, but up to $f = \lfloor \frac{n-1}{3} \rfloor$ servers and any number of clients are faulty. The last row of Table 1 shows the performance of Aardvark, an RBFT state machine replication protocol whose design and implementation are guided by this new philosophy.

In some ways, Aardvark is very similar to traditional BFT protocols: clients send requests to a primary who relays requests to the replicas who agree (explicitly or implicitly) on the sequence of requests and the corresponding results—not unlike PBFT [8], High throughput BFT [19], Q/U [1], HQ [12], Zyzzyva [18], ZZ [32], Scrooge [28], etc.

In other ways, Aardvark is very different and challenges conventional wisdom. Aardvark utilizes signatures for authentication, even though, as Castro correctly observes, "eliminating signatures and using MACs instead eliminates the main performance bottleneck in previous systems" [7]. Aardvark performs regular view changes, even though view changes temporarily prevent the system from doing useful work. Aardvark utilizes point to point communication, even though renouncing IP-multicast gives up throughput deliberately.

We reach these counter-intuitive choices by following a simple and systematic approach: without ever compromising safety, we deliberately refocus both the design of the system and the engineering choices involved in its implementation on the stress that failures can impose on performance. In applying this strategy for RBFT to construct Aardvark, we choose an extreme position inspired by maxi-min strategies in game theory [26]: we reject any optimization for gracious executions that can decrease performance during uncivil executions.

Surprisingly, these counter-intuitive choices impose only a modest cost on its peak performance. As Table 1 illustrates, Aardvark sustains peak throughput of 38667 requests/second, which is within 40% of the best performance we measure on the same hardware for four state-of-the-art protocols. At the same time, Aardvark's fault tolerance is dramatically improved. For a broad range of client, primary, and server misbehaviors we prove that Aardvark's performance remains within a constant factor of its best case performance. Testing of the prototype shows that these changes significantly improve robustness under a range of injected faults.

Once again, however, the main contribution of this paper is neither the Aardvark protocol nor implementation. It is instead a new approach that can—and we believe should—be applied to the design of other BFT protocols. In particular, we (1) demonstrate that existing protocols and their implementations are fragile, (2) argue that BFT protocols should be designed and implemented with a focus on robustness, and (3) use Aardvark to demonstrate that the RBFT approach is viable: we gain qualitatively better performance during uncivil intervals at only modest cost to performance during gracious intervals.

In Section 2 we describe our system model and the guarantees appropriate for high assurance systems. In Section 3 we elaborate on the need to rethink Byzantine fault tolerance and identify a set of design principles for RBFT systems. In Section 4 we present a systematic methodology for designing RBFT systems and an overview of Aardvark. In Section 5 we describe in detail the important components of the Aardvark protocol. In Section 6 we present an analysis of Aardvark's expected performance. In Section 7 we present our experimental evaluation. In Section 8 we discuss related work.

2 System model

We assume the Byzantine failure model where faulty nodes (servers or clients) can behave arbitrarily [21] and a strong adversary can coordinate faulty nodes to compromise the replicated service. We do, however, assume the adversary cannot break cryptographic techniques like collision-resistant hashing, message authentication codes (MACs), encryption, and signatures. We denote a message X signed by principal p's public key as $\langle X \rangle_{\sigma_p}$. We denote a message X with a MAC appropriate for principals p and r as $\langle X \rangle_{\mu_{r,p}}$. We denote a message containing a *MAC authenticator*—an array of MACs appropriate for verification by every replica—as $\langle X \rangle_{\vec{\mu}_r}$

Our model puts no restriction on clients, except that their number be finite: in particular, any number of clients can be arbitrarily faulty. However, the system's safety and liveness properties are guaranteed only if at most $f = \lfloor \frac{n-1}{3} \rfloor$ servers are faulty.

Finally, we assume an asynchronous network where *synchronous intervals*, during which messages are delivered with a bounded delay, occur infinitely often.

Definition 1 (Synchronous interval)**.** *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.*

3 Recasting the problem

The foundation of modern BFT state machine replication rests on an impossibility result and on two principles that assist us in dealing with it. The impossibility result, of course, is FLP [13], which states that no solution to consensus can be both safe and live in an asynchronous systems if nodes can fail. The two principles, first applied by Lamport to his Paxos protocol [20], are at the core of Castro and Liskov's seminal work on PBFT [7]. The first states that 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. The second recognizes, given FLP, that 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.

Within these boundaries, the engineering of BFT protocols has embraced Lampson's well-known recommendation: "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" [22]. Ever since PBFT, the design of BFT systems has then followed a predictable pattern: first, characterize what defines the normal (common) case; then, pull out all the stops to make the system perform well for that case. While different systems don't completely agree on what defines the common case [16], on one point they are unanimous: the common case includes only *gracious executions*, defined as follows:

Definition 2 (Gracious execution)**.** *An execution is gracious iff (a) the execution is synchronous with some implementation-dependent short bound on message delay and (b) all clients and servers behave correctly.*

The results of this approach continue to be spectacular. Since Zyzzyva last year reported a throughput of over 85,000 null requests per second [18], several new protocols have further improved on that mark [16, 28].

Despite these impressive results, we argue that a single minded focus on aggressively tuning BFT systems for the best case of gracious execution, a practice that we have engaged in with relish [18], is increasingly misguided, dangerous, and even futile.

It is misguided, because it encourages the design and implementation of systems that fail to deliver on their basic promise: to tolerate Byzantine faults. While providing impressive throughput during gracious executions, today's high-performance BFT systems are content to guaranteeing weak liveness guarantees (e.g. "eventual progress") in the presence of Byzantine failures. Unfortunately, as we previewed in Figure 1 and show in detail in Section 7.2, these guarantees are weak indeed. Although current BFT systems can *survive* Byzantine faults without compromising safety, we contend that a system that can be made completely unavailable by a simple Byzantine failure can hardly be said to *tolerate* Byzantine faults.

It is dangerous, because it encourages *fragile optimizations*. Fragile optimizations are harmful in two ways. First, as we will see in Section 7.2, they make it easier for a faulty client or server to knock the system off its hard-won optimized execution path and enter an alternative, much more expensive one. Second, they weigh down the system with subtle corner cases, increasing the likelihood of buggy or incomplete implementations.

It is (increasingly) futile, because the race to optimize common case performance has reached a point of diminishing return where many services' peak demands are already far under the best-case throughput offered by existing BFT replication protocols. For such systems, *good enough is good enough*, and further improvements in best case agreement throughput will have little effect on end-to-end system performance.

In our view, a BFT system fulfills its obligations when it provides acceptable and dependable performance across the broadest possible set of executions, including executions with Byzantine clients and servers. In particular, the temptation of fragile optimizations should be resisted: a BFT system should be designed around an execution path that has three properties: (1) it provides acceptable performance, (2) it is easy to implement, and (3) it is robust against Byzantine attempts to push the system away from it. Optimizations for the common case should be accepted only as long as they don't endanger these properties.

FLP tells us that we cannot guarantee liveness in an asynchronous environment. This is no excuse to cling to gracious executions only. In particular, there is no theoretical reason why BFT systems should not be expected to perform well in what we call *uncivil executions*:

Definition 3 (Uncivil execution)**.** *An execution is uncivil iff (a) the execution is synchronous with some implementation-dependent short bound on message delay, (b) up to f servers and an arbitrary number of clients are Byzantine, and (c) all remaining clients and servers are correct.*

Hence, we propose to build RBFT systems that provide adequate performance during uncivil executions. Although we recognize that this approach is likely to reduce the best case performance, we believe that for a BFT system a limited reduction in peak throughput is preferable to the devastating loss of availability that we report in Figure 1 and Section 7.2.

Increased robustness may come at effectively no additional cost as long as a service's peak demand is below the throughput achievable through RBFT design: as a data point, our Aardvark prototype reaches a peak throughput of 38667 req/s.

Similarly, when systems have other bottlenecks, Amdahl's law limits the impact of changing the performance of agreement. For example, we report in Section 7 that PBFT can execute almost 62,000 null requests per second, suggesting that agreement consumes 16.1μs per request. If, rather than a null service, we replicate a service

for which executing an average request consumes $100\mu s$ of processing time, then peak throughput with PBFT settles to about 8613 requests per second. For the same service, a protocol with twice the agreement overhead of PBFT (i.e., $32.2\mu s$ per request), would still achieve peak throughput of about 7564 requests/second: in this hypothetical example, doubling agreement overhead would reduce peak end-to-end throughput by about 12%.

4 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, and primary view change. This is the same basic structure of PBFT [8] and its direct descendants [4, 18, 19, 33, 32], but revisited with the goal of achieving an execution path that satisfies the properties outlined in the previous section: acceptable performance, ease of implementation, and robustness against Byzantine disruptions. To avoid the pitfalls of fragile optimizations, we focus at each stage of the protocol on how faulty nodes, by varying both the nature and the rate of their actions and omissions, can limit the ability of correct nodes to perform in a timely fashion what the protocol requires of them. This systematic methodology leads us to the three main design differences between Aardvark and previous BFT systems: (1) signed client requests, (2) resource isolation, and (3) regular view changes.

Signed client requests. Aardvark clients use digital signatures to authenticate their requests. Digital signatures provide non-repudiation and ensure that all correct replicas make identical decisions about the validity of each client request, eliminating a number of expensive and tricky corner cases found in existing protocols that make use of weaker (though faster) message authentication code (MAC) authenticators [7] to authenticate client requests. The difficulty with utilizing MAC authenticators is that they do not provide the non-repudiation property of digital signatures—one node validating a MAC authenticator does not guarantee that any other nodes will validate that same authenticator [2].

As we mentioned in the Introduction, digital signatures are generally seen as too expensive to use. Aardvark uses them only for client requests where it is possible to push the expensive act of generating the signature onto the client while leaving the servers with the less expensive verification operation. 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.

Because of the additional costs associated with verifying signatures in place of MACs, Aardvark must guard against new denial-of-service attacks where the system receives a large numbers of requests with signatures that need to be verified. Our implementation limits the number of signature verifications a client can inflict on the system by (1) utilizing a hybrid MAC-signature construct to put a hard limit on the number of *faulty* signature verifications a client can inflict on the system and (2) forcing a client to complete one request before issuing the next.

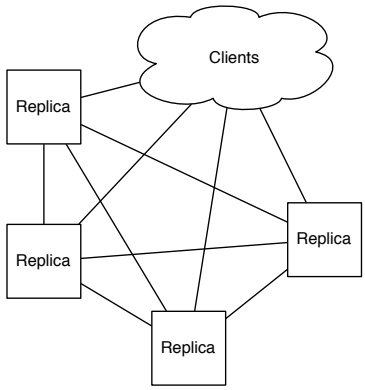

Figure 1: Physical network in Aardvark.

Resource isolation. The Aardvark prototype implementation explicitly isolates network and computational resources.

As illustrated by Fig. 1, Aardvark 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, as happened when a single broken NIC shut down the immigration system at the Los Angeles International Airport [9]. It also allows a node to defend itself against brute force denial of service attacks by disabling the offending NIC. However, using physically separate NICs for communication between each pair of servers incurs a performance hit, as Aardvark can no longer use hardware multicast to optimize all-to-all communication.

As Figure 2 shows, 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. Similarly, 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. Taking advantage of hardware parallelism allows Aardvark to reclaim part of the costs paid to verify signatures on client requests.

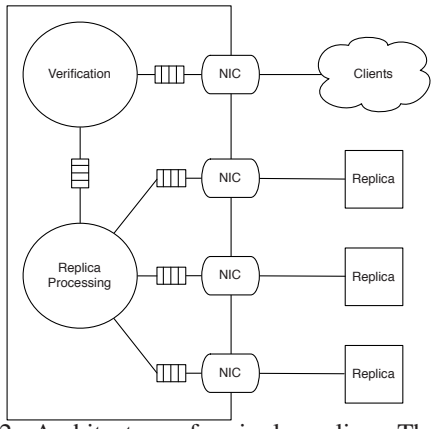

Figure 2: Architecture of a single replica. The replica utilizes a separate NIC for communicating with each other replica and a final NIC to communicate with the collection of clients. Messages from each NIC are placed on separate worker queues.

We use simple brute force techniques for resource scheduling. One could consider network-level scheduling techniques rather than distinct NICs in order to isolate network traffic and/or allow rate-limited multicast. Our goal is to make Aardvark as simple as possible, so we leave exploration of these techniques and optimizations for future work.

Regular view changes. To prevent a primary from achieving tenure and exerting absolute control on system throughput, Aardvark invokes the view change operation on a regular basis. 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:

1. 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.
2. As in prior systems, eventual progress is guaranteed when the system is eventually synchronous.

Previous systems have treated view change as an option of last resort that should only be used in desperate situations to avoid letting throughput drop to zero. However, although the phrase "view change" carries connotations of a complex and expensive protocol, in reality the cost of a view change is similar to the regular cost of agreement. Performing view changes regularly introduces short periods of time during which new requests are not being processed, but the benefits of evicting a misbehaving primary outweigh the periodic costs associated with performing view changes.

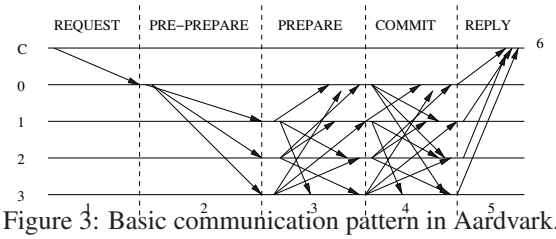

Figure 3: Basic communication pattern in Aardvark.

5 Protocol description

Figure 3 shows the agreement phase communication pattern that Aardvark shares with PBFT. Variants of this pattern are employed in other recent BFT RSM protocols [1, 12, 16, 18, 28, 32, 33], and we believe that, just as Aardvark illustrates how to adapt PBFT via RBFT system design, new Robust BFT systems based on these other protocols can and should be constructed. We organize the following discussion around the numbered steps of the communication pattern of Figure 3.

5.1 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. We ensure this property by having clients sign requests.

To guard against denial of service, we break the processing of a client request into a sequence of increasingly expensive steps. Each step serves as a filter, so that more expensive steps are performed less often. For instance, we ask clients to include also a MAC on their signed requests and have replicas verify only the signature of those requests whose MAC checks out. Additionally, Aardvark explicitly dedicates a single NIC to handling incoming client requests so that incoming client traffic does not interfere with replica-to-replica communication.

5.1.1 Protocol Description

The steps taken by an Aardvark replica to authenticate a client request follow.

1. Client sends a request to a replica.

A client c requests an operation o be performed by the replicated state machine by sending a request message $\langle\langle \text{REQUEST}, o, s, c\rangle_{\sigma_c}, c\rangle_{\mu_{c,p}}$ to the replica p it believes to be the primary. If the client does not receive a timely response to that request, then the client retransmits the request $\langle\langle \text{REQUEST}, o, s, c\rangle_{\sigma_c}, c\rangle_{\mu_{c,r}}$ to all replicas r. Note that 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.

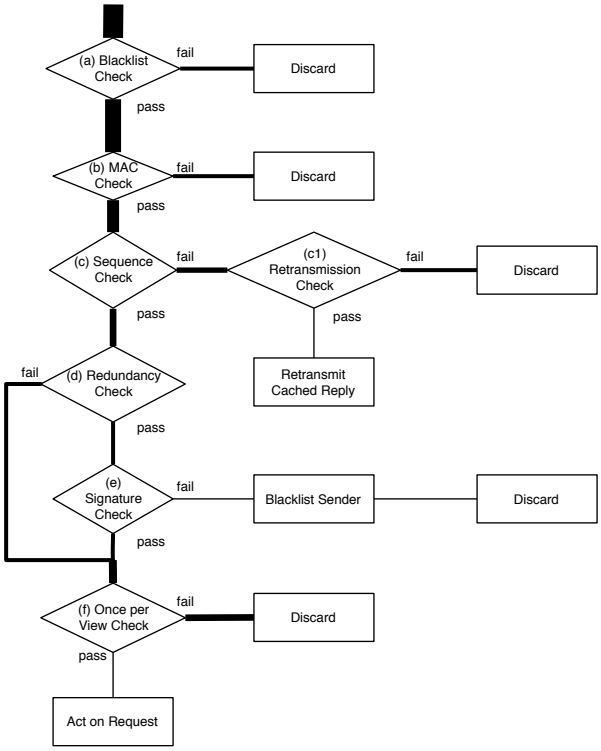

Figure 4: Decision tree followed by replicas while verifying a client request. The narrowing width of the edges portrays the devastating losses suffered by the army of client requests as it marches through the steppes of the verification process. Apologies to Minard.

Upon receiving a client request, a replica proceeds to verify it by following a sequence of steps designed to limit the maximum load a client can place on a server, as illustrated by Figure 4:

(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

(c1) **Retransmission check.** Each replica uses an exponential back off to limit the rate of client reply retransmissions. 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 number s_{req} has previously been verified or the request does not match the cached request, then proceed to step (e). Otherwise (the request matches the cached request from c) proceed to step (f).

(e) **Signature check.** If σ_c is valid, then proceed to step (f). 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.

Primary and non-primary replicas act on requests in different ways. A primary adds requests to a PRE-PREPARE message that is part of the three-phase commit protocol described in Section 5.2. 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. Note that non-primary replicas will forward each request at most once per view, but they may forward a request multiple times provided that a view change occurs between each occurrence.

Note that a REQUEST message that is verified as authentic might contain an operation that the replicated service that runs above Aardvark rejects because of an access control list (ACL) or other service-specific security violation. From the point of view of Aardvark, such messages are valid and will be executed by the service, perhaps resulting in an application level error code.

A node p only blacklists a sender c of a $\langle\langle \text{REQUEST}, o, s, c\rangle_{\sigma_c}, c\rangle_{\mu_{c,p}}$ message if the MAC $\mu_{c,p}$ is valid but the signature σ_c is not. A valid MAC is sufficient to ensure that routine message corruption is not the cause of the invalid signature sent by c, but rather that c has suffered a significant fault or is engaging in malicious behavior. A replica discards all messages it receives from a blacklisted sender and removes the sender from the blacklist after 10 minutes to allow reintegration of repaired machines.

5.1.2 Resource scheduling

Client requests are necessary to provide input to the RSM while replica-to-replica communication is necessary to process those requests. 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, ensuring that a flood of client requests is unable to prevent replicas from making progress on requests already received. Of course, as in a non-BFT service, malicious clients can still deny service to other clients by flooding the network between clients and replicas. Defending against these attacks is an area of active independent research [23, 30].

We deploy our prototype implementation on dual core machines. As Figure 2 shows, one core verifies client re-

quests and the second runs the replica protocol. This explicit assignment allows us to isolate resources and take advantage of parallelism to partially mask the additional costs of signature verification.

5.1.3 Discussion

RBFT aims at minimizing the costs that faulty clients can impose on replicas. As Figure 4 shows, there are four actions triggered by the transmission of a client request that can consume significant replica resources: MAC verification (MAC check), retransmission of a cached reply, signature verification (signature check), and request processing (act on request). The cost a faulty client can cause increases as the request passes each successive check in the verification process, but the rate at which a faulty client can trigger this cost decreases at each step.

Starting from the final step of the decision tree, the design ensures that the most expensive message a client can send is a correct request as specified by the protocol, and it limits the rate at which a faulty client can trigger expensive signature checks and request processing to the maximum rate a correct client would. The sequence check step (c) ensures that a client can trigger signature verification or request processing for a new sequence number only after its previous request has been successfully executed. The redundancy check (d) prevents repeated signature verifications for the same sequence number by caching each client's most recent request. Finally, the once per view check (f) permits repeated processing of a request only across different views to ensure progress. The signature check (e) ensures that only requests that will be accepted by all correct replicas are processed. The net result of this filtering is that, for every k correct requests submitted by a client, each replica performs at most $k + 1$ signature verifications, and any client that imposes a $k+1^{st}$ signature verification is blacklisted and unable to instigate additional signature verifications until it is removed from the blacklist.

Moving up the diagram, a replica responds to retransmission of completed requests paired with valid MACs by retransmitting the most recent reply sent to that client. The retransmission check (c1) imposes an exponential back off on retransmissions, limiting the rate at which clients can force the replica to retransmit a response. To help a client learn the sequence number it should use, a replica resends the cached reply at this limited rate for both requests that are from the past but also for requests that are too far into the future.

Any request that fails the MAC check (b) is immediately discarded. MAC verifications occur on every incoming message that claims to have the right format unless the sender is blacklisted, in which case the blacklist check (a) results in the message being discarded. The rate of MAC verification operations is thus limited by the rate at which messages purportedly from non-blacklisted clients are pulled off the network, and the fraction of processing wasted is at most the fraction of incoming requests from faulty clients.

5.2 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 [8].

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 makes it easier to ensure 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. Faulty replicas can introduce overhead also by sending messages too quickly: to protect themselves, correct replicas in Aardvark schedule messages from other replicas in a round-robin fashion.

Not all expensive operations in Aardvark are triggered by a quorum. In particular, a correct replica that has fallen behind its peers may ask them for the state it is missing by sending them a *catchup message* (see Section 5.2.1). Aardvark replicas defer processing such messages to idle periods. Note that this state-transfer procedure is self-tuning: if the system is unable to make progress because it cannot assemble quorums of PREPARE and COMMIT messages, then it will devote more time to processing catchup messages.

5.2.1 Agreement protocol

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 in Figure 5:

(a) **Volume Check.** If replica q is sending too many messages, blacklist q and discard the message. Otherwise continue to step (b). Aardvark replicas use a distinct NIC for communicating with each replica. Using per-replica NICs allows an Aardvark replica to silence replicas that flood the network and impose excessive interrupt processing load. In our prototype, we disable a network connection when q's rate of message transmission in the current view is a factor of 20 higher than for any other replica. After disconnecting q for flooding, r reconnects q after 10 minutes, or when f other replicas are disconnected for flooding.

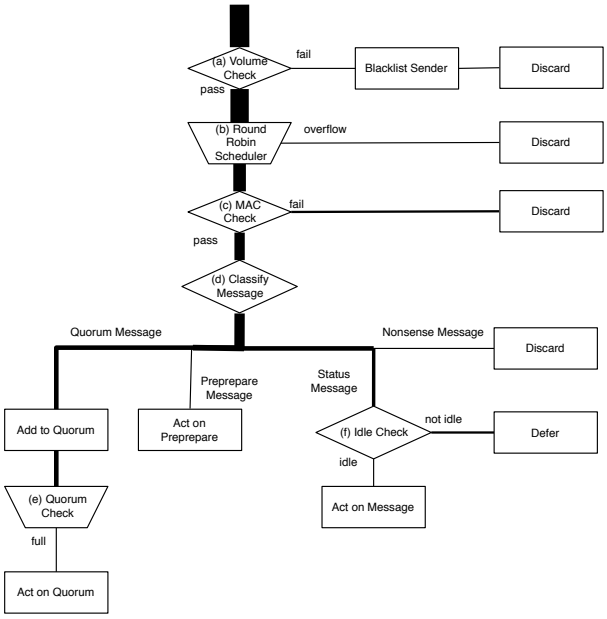

Figure 5: Decision tree followed by a replica when handling messages received from another replica. The width of the edges indicates the rate at which messages reach various stages in the processing.

(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 . Discard received messages when the buffers are full.

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

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

- If the message is PRE-PREPARE, then process it immediately in protocol step 3 below.
- 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 protocol steps 4-6 below.

(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 from step 2 of the communication pattern in Figure 3.

> 2. Primary forms a PRE-PREPARE message containing a set of valid requests and sends the PRE-PREPARE to all replicas.

The primary creates and transmits a $\langle$PRE-PREPARE, $v, n,$ $\langle$REQUEST, $o, s, c\rangle_{\sigma_c}\rangle_{\vec{\mu}_p}$ message where v is the current view number, n is the sequence number for the PRE-PREPARE, and the authenticator is valid for all replicas. Although we show a single request as part of the PRE-PREPARE message, multiple requests can be batched in a single PRE-PREPARE [8, 14, 18, 19].

> 3. Replica receives PRE-PREPARE from the primary, authenticates the PRE-PREPARE, and sends a PREPARE to all other replicas.

Upon receipt of $\langle$PRE-PREPARE, $v, n,$ $\langle$REQUEST, $o, s, c\rangle_{\sigma_c}\rangle_{\vec{\mu}_p}$ from primary p, replica r verifies the message's authenticity following a process similar to the one described in Section 5.1 for verifying requests. If r has already accepted the PRE-PREPARE message, r discards the message preemptively. If r has already processed a different PRE-PREPARE message with $n' = n$ during view v, then r discards the message. If r has not yet processed a PRE-PREPARE message for n during view v, r first checks that the appropriate portion of the MAC authenticator $\vec{\mu}_p$ is valid. If the replica has not already done so, it then checks the validity of σ_c. If the authenticator is not valid r discards the message. If the authenticator is valid and the client signature is invalid, then the replica blacklists the primary and requests a view change. If, on the other hand, the authenticator and signature are both valid, then the replica logs the PRE-PREPARE message and forms a $\langle$PREPARE, $v, n,$ h,$r\rangle_{\vec{\mu}_r}$ to be sent to all other replicas where h is the digest of the set of requests contained in the PRE-PREPARE message.

> 4. Replica receives $2f$ PREPARE messages that are consistent with the PRE-PREPARE message for sequence number n and sends a COMMIT message to all other replicas.

Following receipt of $2f$ matching PREPARE messages from non-primary replicas r' that are consistent with a PRE-PREPARE from primary p, replica r sends a $\langle$COMMIT,$v, n, r\rangle_{\vec{\mu}_r}$ message to all replicas. Note that the PRE-PREPARE message from the primary is the $2f + 1^{\text{st}}$ message in the PREPARE quorum.

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

After receipt of $2f+1$ matching $\langle \text{COMMIT}, v, n, r' \rangle_{\vec{\mu}_{r'}}$ from distinct replicas r', replica r commits and executes the request before sending $\langle \text{REPLY}, v, u, r \rangle_{\mu_{r,c}}$ to client c where u is the result of executing the request and v is the current view.

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

We also support Castro's tentative execution optimization [8], but we omit these details here for simplicity. They do not introduce any new issues for our RBFT design and analysis.

Catchup messages. State catchup messages are not an intrinsic part of the agreement protocol, but fulfill an important logistical priority of bringing replicas that have fallen behind back up to speed. 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. Aardvark explicitly delays the processing of catchup messages until there are idle cycles available at a replica—as long as the system is making progress, processing a high volume of requests, there is no need to spend time bringing a slow replica up to speed!

5.2.2 Discussion

We now discuss the Aardvark agreement protocol through the lens of RBFT, starting from the bottom of Figure 5. Because every quorum contains at least a majority of correct replicas, faulty replicas can only marginally alter the rate at which correct replicas take actions (e) that require a quorum of messages. Further, because a correct replica processes catchup messages (f) only when otherwise idle, faulty replicas cannot use catchup messages to interfere with the processing of other messages. When client requests are pending, catchup messages are processed only if too many correct replicas have fallen behind and the processing of quorum messages needed for agreement has stalled—and only until enough correct replicas to enable progress have caught up. Also note that the queue of pending catchup messages is finite, and a replica discards excess catchup messages.

A replica processes PRE-PREPARE messages at the rate they are sent by the primary. If a faulty primary sends them too slowly or too quickly, throughput may be reduced, hastening the transition to a new primary as described in Section 5.3.

Finally, a faulty replica could simply bombard its correct peers with a high volume of messages that are eventually discarded. The round-robin scheduler (b) limits the damage that can result from this attack: if c of its peers have pending messages, then a correct replica wastes at most $\frac{1}{c}$ of the cycles spent checking MACs and classifying messages on what it receives from any faulty replica. The round-robin scheduler also discards messages that overflow a bounded buffer, and the volume check (a) similarly limits the rate at which a faulty replica can inject messages that the round-robin scheduler will eventually discard.

5.3 Primary view changes

Employing a primary to order requests enables batching [8, 14] and avoids the need to trust clients to obey a back off protocol [1, 10]. However, because the primary is responsible for selecting which requests to execute, the system throughput is at most the throughput of the primary. The primary is thus in a unique position to control both overall system progress [3, 4] and fairness to individual clients.

The fundamental challenge to safeguarding performance against a faulty primary is that a wide range of primary behaviors can hurt performance. For example, the 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, etc.

Hence, rather than designing specific mechanism to defend against each of these threats, past BFT systems [8, 18] have relied on view changes to replace an unsatisfactory primary with a new, hopefully better, one. Past systems trigger view changes conservatively, only changing views when it becomes apparent that the current primary is unlikely to allow the system to make even minimal progress.

Aardvark uses the same view change mechanism described in PBFT [8]; in conjunction with the agreement protocol, view changes in PBFT are sufficient to ensure eventual progress. They are not, however, sufficient to ensure acceptable progress.

5.3.1 Adaptive throughput

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. More specifically, replicas in Aardvark expect two things from the primary: a regular supply of PRE-PREPARE messages and high sustained throughput. Following the completion of a view change, each replica starts a heartbeat timer that is reset whenever the next valid PRE-PREPARE message is received. If a replica does not

receive the next valid PRE-PREPARE message before the heartbeat timer expires, the replica initiates a view change. To ensure eventual progress, a correct replica doubles the heartbeat interval each time the timer expires. Once the timer is reset because a PRE-PREPARE message is received, the replica resets the heartbeat timer back to its initial value. The value of the heartbeat timer is application and environment specific: our implementation uses a heartbeat of 40ms, so that a system that tolerates f failures demands a minimum of 1 PRE-PREPARE every every $2^f \times 40$ms.

The periodic checkpoints that, at pre-determined intervals, correct replicas must take to bound their state offer convenient synchronization points to assess the throughput that the primary is able to deliver. If the observed throughput in the interval between two successive checkpoints falls below a specified threshold, initially 90% of the maximum throughput observed during the previous n views, the replica initiates a view change to replace the current primary. At each checkpoint interval following an initial grace period at the beginning of each view, 5s in our prototype, the required throughput is increased by a factor of 0.01. Continually raising the bar that the current primary must reach in order to stay in power guarantees that a view change will eventually be replaced, restarting the process with the next primary. Conversely, if the system workload changes, the required throughput adjusts over n views to reflect the performance that a correct primary can provide.

The combined effect of Aardvark's new expectations on the primary is that during the first 5s of a view the primary is required to provide throughput of at least 1 request per 40ms or face eviction. The throughput of any view that lasts longer than 5s is at least 90% of the maximum throughput observed during the previous n views.

5.3.2 Fairness

In addition to hurting overall system throughput, primary replicas can influence which requests are processed. A faulty primary could be unfair to a specific client (or set of clients) by neglecting to order requests from that client. To limit the magnitude of this threat, replicas track fairness of request ordering. 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 and, before forwarding the request to the primary, it records the sequence number k of the most recent PRE-PREPARE received during the current view. The replica 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. This ensures that two clients issuing comparable workloads observe throughput values within a constant factor of each other.

5.3.3 Discussion

The adaptive view change and PRE-PREPARE heartbeats leave a faulty primary with two options: it can provide substandard service and be replaced promptly, or it can remain the primary for an extended period of time and provide service comparable to what a non-faulty primary would provide. A faulty primary that does not make any progress will be caught very quickly by the heartbeat timer and summarily replaced. To avoid being replaced, a faulty primary must issue a steady stream of PRE-PREPARE messages until it reaches a checkpoint interval, when it is going to be replaced until it has provided the required throughput. To do *just* what is needed to keep ahead of its reckoning for as long as possible, a faulty primary will be forced to to deliver 95% of the throughput expected from a correct primary.

Periodic view changes may appear to institutionalize overhead, but their cost is actually relatively small. Although the term *view change* evokes images of substantial restructuring, in reality a view change costs roughly as much as a single instance of agreement with respect to message/protocol complexity: when performed every 100+ requests, periodic view changes have marginal performance impact during gracious or uncivil intervals.

6 Analysis

In this section, we analyze the throughput characteristics of Aardvark when the number of client requests is large enough to saturate the system and a fraction g of those requests is correct. We show that Aardvark's throughput during long enough uncivil executions is within a constant factor of its throughput during gracious executions of the same length provided there are sufficient correct clients to saturate the servers.

For simplicity, we restrict our attention to an Aardvark implementation on a single-core machine with a processor speed of κ GHz. We consider only the computational costs of the cryptographic operations—verifying signatures, generating MACs, and verifying MACs, requiring θ, α, and α cycles, respectively. Since these operations occur only when a message is sent or received, and the cost of sending or receiving messages is small, we expect similar results when modeling network costs explicitly.

We begin by computing Aardvark's peak throughput during a gracious view, i.e. a view that occur during a gracious execution, in Theorem 1. We then show in Theorem 2 that during uncivil views, i.e. views that occur during uncivil executions, with a correct primary Aardvark's throughput is at least g times the throughput achieved during a gracious view; as long as the primary is correct faulty replicas are unable to adversely

impact Aardvark's throughput. Finally, we show that the throughput of an uncivil execution is at least the fraction of correct replicas times g times the throughput achieved during a gracious view.

We begin in Theorem 1 by computing t_{peak}, Aardvark's peak throughput during a gracious view, i.e. a view that occurs during a gracious execution. We then show in Theorem 2 that during uncivil views in which the primary replica is correct, Aardvark's peak throughput is only reduced to $g \times t_{peak}$: in other words, ignoring low level network overheads faulty replicas are unable to curtail Aardvark's throughput when the primary is correct. Finally, we show in Theorem 3 that the throughput across all views of an uncivil execution is within a constant factor of $\frac{n-f}{n} \times g \times t_{peak}$.

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.*

Proof. We examine the actions required by each server to process one batch of size b. For each request in the batch, every server verifies one signature. The primary also verifies one MAC per request. For each batch, the primary generates $n-1$ MACs to send the PRE-PREPARE and verifies $n-1$ MACs upon receipt of the PREPARE messages; replicas instead verify one MAC in the primary's PRE-PREPARE , generate $(n-1)$ MACs when they send the PREPARE messages, and verify $(n-2)$ MACs when they receive them. Finally, each server first sends and then receives $n-1$ COMMIT messages, for which it generates and verifies a total of $n-2$ MACs, and generates a final MAC for each request in the batch to authenticate the response to the client. The total computational load per request is thus $\theta + \frac{(4n+2b-4)}{b}\alpha$ at the primary, and $\theta + \frac{(4n+b-4)}{b}\alpha$ at a replica. The system's throughput at saturation during a sufficiently long view in a gracious interval is thus at least $\frac{\kappa}{\theta+\frac{(4n+2b-4)}{b}\alpha}$ requests/sec. □

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.*

Proof. Let θ and α denote the cost of verifying, respectively, a signature and a MAC. We show that if g is the fraction of correct requests, the throughput during uncivil views with a correct primary approaches g of the gracious view's throughput as the ratio $\frac{\alpha}{\theta}$ tends to 0.

In an uncivil view, faulty clients may send unfaithful requests to every server. Before being able to form a batch of b correct requests, the primary may have to verify $\frac{b}{g}$ signatures and MACs, and correct replicas may verify $\frac{b}{g}$ signatures and an additional $(\frac{b}{g})(1-g)$ MACs. Because a correct server processes messages from other servers in round-robin order, it will process at most two messages from a faulty server per message that it would have processed had the server been correct. The total computational load per request is thus $\frac{1}{g}(\theta + \frac{b(1+g)+4g(n-1+f)}{b}\alpha)$ at the primary, and $\frac{1}{g}(\theta+\frac{b+4g(n-1+f)}{b}\alpha)$ at a replica. The system's throughput at saturation during a sufficiently long view in an uncivil interval with a correct primary thus is at least $\frac{g\kappa}{\theta+\frac{(b(1+g)+4g(n-1+f)}{b}\alpha}$ requests per second: as the ratio $\frac{\alpha}{\theta}$ tends to 0, the ratio between the uncivil and gracious throughput approaches g. □

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.*

Proof. First consider the case in which all the uncivil views have correct primary replicas. Assume that in a properly configured Aardvark $t_{baseViewTimeout}$ is set so that during an uncivil interval, a view change to a correct primary completes within $t_{baseViewTimeout}$. Since a primary's view lasts at least $t_{gracePeriod}$, as the ratio $\frac{\alpha}{\theta}$ tends to 0, the ratio between the throughput during a gracious view and an uncivil interval approaches $g\frac{t_{gracePeriod}}{t_{baseViewTimeout}+t_{gracePeriod}}$

Now consider the general case. If the uncivil interval is long enough, at most $\frac{f}{n}$ of its views will have a Byzantine primary. Aardvark's heartbeat timer provides two guarantees. First, a Byzantine server that does not produce the throughput that is expected of a correct server will not last as primary for longer than a grace period. Second, a correct server is always retained as a primary for at least the length of a grace period. Furthermore, since the throughput expected of a primary at the beginning of a view is a constant fraction of the maximum throughput achieved by the primary replicas of the last n views, faulty primary replicas cannot arbitrarily lower the throughput expected of a new primary. Finally, since the view change timeout is reset after a view change that results in at least one request being executed in the new view, no view change attempt takes longer then $t_{maxViewTimeout} = 2^f t_{baseViewTimeout}$. It follows that, during a sufficiently long uncivil interval, the throughput will be within a factor of $\frac{t_{gracePeriod}}{t_{maxViewTimeout}+t_{gracePeriod}}\frac{n-f}{n}$ of that of Theorem 2, and, as $\frac{\alpha}{\theta}$ tends to 0, the ratio between

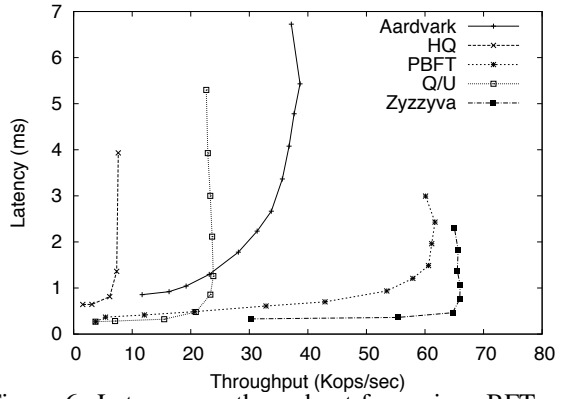

Figure 6: Latency vs. throughput for various BFT systems.

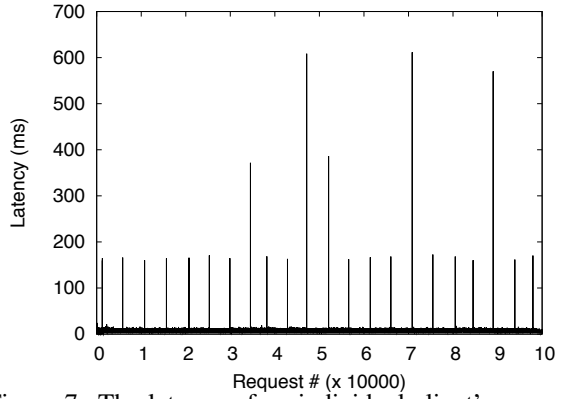

Figure 7: The latency of an individual client's requests running Aardvark with 210 total clients. The sporadic jumps represent view changes in the protocol.

the throughput during uncivil and gracious intervals approaches $g\frac{t_{gracePeriod}}{t_{maxViewTimeout}+t_{gracePeriod}}\frac{(n-f)}{n}$. □

7 Evaluation

We evaluate the performance of Aardvark, PBFT, HQ, Q/U and Zyzzyva on an Emulab cluster [31]. This cluster consists of machines with dual 3GHz Intel Pentium 4 Xeon processors, 1GB of memory, and 1 Gb/s Ethernet connections.

The code bases used to report our results are provided by the respective systems' authors. James Cowling provided us the December 2007 public release of the PBFT code base [5] as well as a copy of the HQ co-debase. We used version 1.3 of the Q/U co-debase, provided to us by Michael Abd-El-Malek in October 2008 [27]. The Zyzzyva co-debase is the version used in the SOSP 2007 paper [18]. Whenever feasible, we rely on the existing pre-configurations for each system to handle $f = 1$ Byzantine failure.

Our evaluation makes three points: (a) despite our choice to utilize signatures, change views regularly, and forsake IP multicast, Aardvark's peak throughput is competitive with that of existing systems; (b) existing systems are vulnerable to significant disruption as a result of a broad range of Byzantine behaviors; and (c) 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*.

7.1 Aardvark

Aardvark's peak throughput is competitive with that of state of the art systems as shown in Figure 6. Aardvark's throughput peaks 38667 operations per second, while Zyzzyva and PBFT observe maximum throughputs of 65999 and 61710 operations per second, respectively.

Figures 7 and 8 explore the impact of regular view changes on the latency observed by Aardvark clients in an experiment with 210 clients each issuing 100,000 requests. Figure 7 shows the per request latency observed by a single client during the run. The periodic latency spikes correspond to view changes. When a client issues a request as the view change is initiated, the request is not processed until the request arrives at the new primary following a client timeout and retransmission. In most cases a single client retransmission is sufficient, but additional retransmissions may be required when multiple view changes occur in rapid succession. Figure 8 shows the CDF for latencies of all client requests in the same experiment. We see that 99.99% of the requests have latency under 15ms, and only a small fraction of all requests incur the higher latencies induced by view changes. We configure an Aardvark client with a retransmission timeout of 150ms and we have not explored other settings.

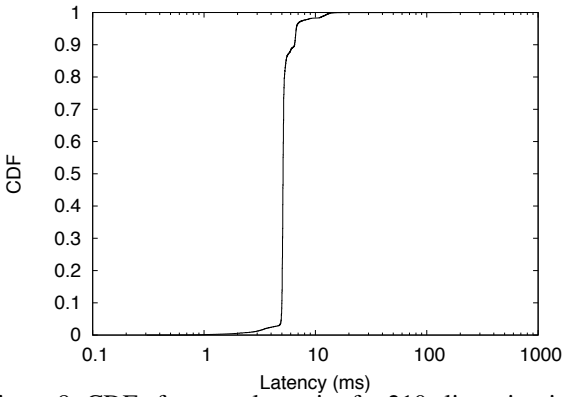

Figure 8: CDF of request latencies for 210 clients issuing 100,000 requests with Aardvark servers.

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

Table 2: Peak throughput of Aardvark and incremental versions of the Aardvark protocol

7.1.1 Putting Aardvark together

Aardvark incorporates several key design decisions that enable it to perform well in the presence of Byzantine failure. We study the performance impact of these decisions by measuring the throughput of several PBFT and Aardvark variations, corresponding to the evolution between these two systems. Table 2 reports these peak throughputs.

While requiring clients in PBFT to sign requests reduces throughput by 50%, we find that the cost of requiring Aardvark clients to use the hybrid MAC-signature scheme imposes a smaller 33% hit to system throughput. Explicitly separating the work queues for client and replica communication makes it easy for Aardvark to utilize the second processor in our test bed machines, which reduces the additional costs Aardvark pays to verify signed client requests. This parallelism is the primary source of the 30% improvement we observe between PBFT with signatures and Aardvark.

Peak throughput for Aardvark with and without regular view changes is comparable. The reason for this is rather straightforward: when both the new and old primary replicas are non-faulty, a view change requires approximately the same amount of work as a single instance of consensus. Aardvark views led by a non-faulty primary are sufficiently long that the throughput costs associated with performing a view change are negligible.

7.2 Evaluating faulty systems

In this section we evaluate Aardvark and existing systems in the context of failures. It is impossible to test every possible Byzantine behavior; consequently we use our knowledge of the systems to construct a set of workloads that we believe to be close to the worst case for Aardvark and other systems. While other faulty behaviors are possible and may stress the evaluated systems in different ways, we believe that our results are indicative of both the frailty of existing systems and the robustness of Aardvark.

7.2.1 Faulty clients

We focus our attention on two aspects of client behavior that have significant impact on system throughput: request dissemination and network flooding.

Request dissemination. Table 1 in the Introduction explores the impact of faulty client behavior related to request distribution on the PBFT, HQ, Zyzzyva, and Aardvark prototypes. We implement different client behaviors for the different systems in order to stress test the design decisions the systems have made.

In PBFT and Zyzzvya, the clients send requests that are authenticated with MAC authenticators. The faulty client includes an inconsistent authenticator on requests so that request verification will succeed at the primary but fail for all other replicas. When the primary includes the client request in a PRE-PREPARE message, the replicas are unable to verify the request.

We developed this workload because, on paper, the protocols specify what appears to be an expensive processing path to handle this contingency. In this situation PBFT specifies a view change while Zyzzyva invokes a conflict resolution procedure that blocks progress and requires replicas to generate signatures. In theory these procedures should have a noticeable, though finite, impact on performance. In particular, PBFT progress should stall until a timeout forces a new view ([6] pp. 42–43), at which point other clients can make some progress until the faulty client stalls progress again. In Zyzzyva, the servers should pay extra overheads for signatures and view changes.

In practice the throughput of both prototype implementations drops to 0. In Zyzzyva the reconciliation protocol is not fully implemented; in PBFT the client behavior results in repeated view changes, and we have not observed our experiment to finish. While the full PBFT and Zyzzyva protocol specifications guarantee liveness under eventual synchrony, the protocol steps required to handle these cases are sufficiently complex to be difficult to implement, easy to overlook, or both.

In HQ, our intended attack is to have clients send certificates during the WRITE-2 phase of the protocol with an inconsistent MAC authenticator. The response specified by the protocol is a signed WRITE-2-REFUSED message which is subsequently used by the client to initiate a call to initiate a request processed by an internal PBFT protocol. This set of circumstances presents a point in the HQ design where a single client, either faulty or simply unlucky, can force the replicas to generate expensive signatures resulting in a degradation in system throughput. We are unable to evaluate the precise impact of this client behavior because the replica processing necessary to handle inconsistent MAC authenticators from clients is not implemented.

Q/U clients, in the lack of contention, are unable to influence each other's operations. During contention, replicas are required to perform barrier and commit operations that are rate limited by a client-initiated exponential back off. During the barrier and commit opera-

tions, a faulty client that sends inconsistent certificates to the replicas can theoretically complicate the process further. We implement a simpler scenario in which all clients are correct, yet they issue contending requests to the replicas. In this setting with only 20 clients, Q/U provides 0 throughput. Q/U's focus on performance in the absence of both failures and contention makes it especially vulnerable in practice—clients that issue contending requests can decimate system throughput, whether the clients are faulty or not.

To avoid corner cases where different replicas make different judgments about the legitimacy of a request, Aardvark clients sign requests. 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. We implement both attacks and find the results to be comparable. In Table 1 we report the results for requests with invalid MACs.

Network flooding. In Table 3 we demonstrate the impact of a single faulty client that floods the replicas with messages. During these experiments 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. For PBFT and Zyzzyva, 210 clients are sufficient to saturate the servers while Q/U and HQ are saturated with 30 client processes.

The PBFT and Zyzzyva prototypes suffer dramatic performance degradation as their incoming network resources are consumed by the flooding client; processing the incoming client requests disrupt the replica-to-replica communication necessary for the systems to make progress. In both cases, the pending client requests eventually overflows internal queues and crashes the servers. Q/U and HQ suffer smaller degradations in throughput from the spamming replicas. The UDP traffic is dropped by the network stack with minimal processing because they are not valid TCP packets. The slowdowns observed in Q/U and HQ correspond to the displaced network bandwidth.

The reliance on TCP communication in Q/U and HQ changes rather than solves the challenge presented by a flooding client. For example, a single faulty client that repeatedly requests TCP connections crashes both the Q/U and HQ servers.

In each of these systems, the vulnerability to network flooding is a byproduct of the prototype implementation and is not fundamental to the protocol design. Network isolation techniques such as those described in Section 5 could similarly be applied to these systems.

In the case of Aardvark, the decision to use separate NICs and work queues for client and replica requests ensures that a faulty client is unable to prevent replicas from processing requests that have already entered the system. The throughput degradation observed by Aardvark tracks the fraction of requests that replicas receive that were sent by non-faulty clients.

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	-

Table 3: Observed peak throughput of BFT systems in the fault free case and under heavy client retransmission load. UDP network flooding corresponds to a single faulty client sending 9KB messages. TCP network flooding corresponds to a single faulty client sending requests to open TCP connections and is shown for TCP based systems.

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

Table 4: Throughput during intervals in which the primary delays sending PRE-PREPARE message (or equivalent) by 1, 10, and 100 ms.

7.2.2 Faulty Primary

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

In Table 4 we show the impact on PBFT, Zyzzyva, and Aardvark prototypes of a primary that delays sending PRE-PREPARE messages by 1, 10, or 100 ms. The throughput of both PBFT and Zyzzyva degrades dramatically as the slow primary is not slow enough to trigger their view change conditions. This throughput degradation is a consequence of the protocol design and specification of when view changes should occur. With an extremely slow primary, Zyzzyva eventually succumbs to a memory leak exacerbated by holding on to requests for an extended period of time. The throughput achieved by Aardvark indicates that adaptively performing view changes in response to observed throughput is a good technique for ensuring performance.

In addition to controlling the rate at which requests are inserted into the system, the primary is also responsible for controlling which requests are inserted into the system. Table 5 explores the impact that an unfair primary can have on the throughput of a targeted node. In

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

Table 5: Average throughput for a starved client that is shunned by a faulty primary versus the average per-client throughput for any other client.

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	-

Table 6: Observed peak throughput and observed throughput when one replica floods the network with messages. UDP flooding consists of a replica sending 9KB messages to other replicas rather than following the protocol. TCP flooding consists of a replica repeatedly attempting to open TCP connections on other replicas.

the case of PBFT and Aardvark, the primary sends a PRE-PREPARE for the targeted client's request only after receiving the the request 9 times. This heuristic prevents the PBFT primary from triggering a view change and demonstrates dramatic degradation in throughput for the targeted client in comparison to the other clients in the system. For Zyzzyva, the unfair primary ignores messages from the targeted client entirely. The resulting throughput is 0 because the implementation is incomplete, and replicas in the Zyzzyva prototype do not forward received requests to the primary as specified by the protocol. Aardvark's fairness detection and periodic view changes limit the impact of the unfair primary.

7.2.3 Non-Primary Replicas

We implement a faulty replica that fails to process protocol messages and insted blasts network traffic at the other replicas and show the results in Table 6. In the first experiments, a faulty replica blasts 9KB UDP messages at the other replicas. The PBFT and Zyzzyva prototypes again show very low performance as the incoming traffic from the spamming replica displaces much of the legitimate traffic in the system, denying the system both requests from the clients and also replica messages required to make progress. Aardvark's use of separate worker queues ensures that the replicas process the messages necessary to make progress. In the second experiment, the faulty The Q/U and HQ replicas again open TCP connections, consuming all of the incoming connections on the other replicas and denying the clients access to the service.

Once again, the shortcomings of the systems are a byproduct of implementation and not protocol design. We speculate that improved network isolation techniques would make the systems more robust.

8 Related work

We are not the first to notice significantly reduced performance for BFT protocols during periods of failures or bad network performance or to explore how timing and failure assumptions impact performance and liveness of fault tolerant systems.

Singh et al. [29] show that PBFT [8], Q/U [1], HQ [12], and Zyzzyva [18] are all sensitive to network performance. They provide a thorough examination of the gracious executions of the four canonical systems through a ns2 [25] network simulator. Singh et al. explore performance properties when the participants are well behaved and the network is faulty; we focus our attention on the dual scenario where the participants are faulty and the network is well behaved.

Aiyer et al. [3] and Amir et al. [4] note that a slow primary can result in dramatically reduced throughput. Aiyer et al. combat this problem by frequently rotating the primary. Amir et al. address the challenge instead by introducing a pre-agreement protocol requiring several all-to-all message exchanges and utilizing signatures for all authentication. Their solution is designed for environments where throughout of 800 requests per second is considered good. Condie et al. [11] address the ability of a well placed adversary to disrupt the performance of an overlay network by frequently restructuring the overlay, effectively changing its view.

The signature processing and scheduling of replica messages in Aardvark is similar in flavor to the early rejection techniques employed by the LOCKSS system [15, 24] in order to improve performance and limit the damage an adversary can inflict on system.

PBFT [8], Q/U [1], HQ [12], and Zyzzyva [18] are recent BFT replication protocols that focus on optimizing performance during gracious executions and collectively demonstrate that BFT replication systems can provide excellent performance during gracious executions. We instead focus on increasing the robustness of BFT systems by providing good performance during uncivil executions. Hendricks et al. [17] explore the use of erasure coding increase the efficiency of BFT replicated storage; they emphasizes increasing the bandwidth and storage efficiency of a replication protocol similar to Q/U and not the fault tolerance of the replication protocol.

9 Conclusion

We claim that high assurance systems require BFT protocols that are more robust to failures than existing sys-

tems. Specifically, BFT protocols suitable for high assurance systems must provide adequate throughput during uncivil intervals in which the network is well behaved but an unknown number of clients and up to f servers are faulty. We present Aardvark, the first BFT state machine protocol designed and implemented to provide good performance in the presence of Byzantine faults. Aardvark gives up some throughput during gracious executions, for significant improvement in performance during uncivil executions.

Aardvark is far from being the last word in robust BFT replication: we believe that improvements to the design and implementation of Aardvark, as well as to the methodology that led us to it, are both possible and likely. Specific challenges that remain for future work include formally verifying the design and implementations of BFT systems, developing a notion of optimality for robust BFT systems that captures the fundamental tradeoffs betwee fault-free and fault-full performance, and extending BFT replication to deployable large scale applications.

10 Acknowledgements

The authors would like to thank our shepherd, Petros Maniatis, for his detailed comments and the anonymous reviewers for their insightful reviews. This work was supported in part by NSF grants CSR-PDOS-0509338 and CSR-PDOS-0720649.

References

[1] ABD-EL-MALEK, M., GANGER, G., GOODSON, G., REITER, M., AND WYLIE, J. Fault-scalable Byzantine fault-tolerant services. In *SOSP* (2005).

[2] AIYER, A. S., ALVISI, L., BAZZI, R. A., AND CLEMENT, A. Matrix signatures: From macs to digital signatures in distributed systems. In *DISC* (2008).

[3] AIYER, A. S., ALVISI, L., CLEMENT, A., DAHLIN, M., MARTIN, J.-P., AND PORTH, C. BAR fault tolerance for cooperative services. In *SOSP* (Oct. 2005).

[4] AMIR, Y., COAN, B., KIRSCH, J., AND LANE, J. Byzantine replication under attack. In *DSN* (2008).

[5] BFT project homepage. `http://www.pmg.csail.mit.edu/bft/#sw`.

[6] CASTRO, M. *Practical Byzantine Fault Tolerance*. PhD thesis, 2001.

[7] CASTRO, M., AND LISKOV, B. Practical Byzantine fault tolerance. In *OSDI* (1999).

[8] CASTRO, M., AND LISKOV, B. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.* (2002).

[9] At LAX, computer glitch delays 20,000 passengers. http://travel.latimes.com/articles/la-trw-lax12aug12.

[10] CHOCKLER, G., MALKHI, D., AND REITER, M. Backoff protocols for distributed mutual exclusion and ordering. In *ICDCS* (2001).

[11] CONDIE, T., KACHOLIA, V., SANKARARAMAN, S., HELLERSTEIN, J. M., AND MANIATIS, P. Induced churn as shelter from routing-table poisoning. In *NDSS* (2006).

[12] COWLING, J., MYERS, D., LISKOV, B., RODRIGUES, R., AND SHRIRA, L. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *OSDI* (2006).

[13] FISCHER, M., LYNCH, N., AND PATERSON, M. Impossibility of distributed consensus with one faulty process. *JACM* (1985).

[14] FRIEDMAN, R., AND RENESSE, R. V. Packing messages as a tool for boosting the performance of total ordering protocls. In *HPDC* (1997).

[15] GIULI, T. J., MANIATIS, P., BAKER, M., ROSENTHAL, D. S. H., AND ROUSSOPOULOS, M. Attrition defenses for a peer-to-peer digital preservation system. In *USENIX* (2005).

[16] GUERRAOUI, R., QUÉMA, V., AND VUKOLIC, M. The next 700 bft protocols. Tech. rep., Infoscience — Ecole Polytechnique Federale de Lausanne (Switzerland), 2008.

[17] HENDRICKS, J., GANGER, G. R., AND REITER, M. K. Low-overhead Byzantine fault-tolerant storage. In *SOSP* (2007).

[18] KOTLA, R., ALVISI, L., DAHLIN, M., CLEMENT, A., AND WONG, E. Zyzzyva: speculative Byzantine fault tolerance. In *SOSP* (2007).

[19] KOTLA, R., AND DAHLIN, M. High throughput Byzantine fault tolerance. In *DSN* (June 2004).

[20] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst.*, 2 (1998).

[21] LAMPORT, L., SHOSTAK, R., AND PEASE, M. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.* (1982).

[22] LAMPSON, B. W. Hints for computer system design. *SIGOPS Oper. Syst. Rev. 17* (1983).

[23] MAHIMKAR, A., DANGE, J., SHMATIKOV, V., VIN, H., AND ZHANG, Y. dFence: Transparent network-based denial of service mitigation. In *NSDI* (2007).

[24] MANIATIS, P., ROUSSOPOULOS, M., GIULI, T. J., ROSENTHAL, D. S. H., AND BAKER, M. The LOCKSS peer-to-peer digital preservation system. *ACM Trans. Comput. Syst.* (2005).

[25] NS-2. `http://www.isi.edu/nsnam/ns/`.

[26] OSBORNE, M., AND RUBINSTEIN, A. *A Course in Game Theory*. MIT Press, 1994.

[27] Query/Update protocol. `http://www.pdl.cmu.edu/QU/index.html`.

[28] SERAFINI, M., BOKOR, P., AND SURI, N. Scrooge: Stable speculative byzantine fault tolerance using testifiers. Tech. rep., Darmstadt University of Technology, Department of Computer Science, September 2008.

[29] SING, A., DAS, T., MANIATIS, P., DRUSCHEL, P., AND ROSCOE, T. Bft protocols under fire. In *NSDI* (2008).

[30] WALFISH, M., VUTUKURU, M., BALAKRISHNAN, H., KARGER, D., AND SHENKER, S. DDoS defense by offense. In *SIGCOMM* (2006).

[31] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An integrated experimental environment for distributed systems and networks. In *OSDI* (2002).

[32] WOOD, T., SINGH, R., VENKATARAMANI, A., AND SHENOY, P. Zz: Cheap practical bft using virtualization. Tech. rep., University of Massachussets, 2008.

[33] YIN, J., MARTIN, J.-P., VENKATARAMANI, A., ALVISI, L., AND DAHLIN, M. Separating agreement from execution for Byzantine fault tolerant services. In *SOSP* (2003).