Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults *(Superseded by UT TR-08-44)*

Allen Clement[1], Mirco Marchetti[2], Edmund Wong[1],
Lorenzo Alvisi[1], and Mike Dahlin[1]
[1]The University of Texas at Austin, [2],University of Modena and Reggio Emilia,
{aclement, mirco, elwong, lorenzo, dahlin}@cs.utexas.edu

Abstract: This paper is motivated by a simple observation: although recently developed BFT state machine replication protocols are quite fast, they don't actually tolerate Byznatine faults very well. In particular a single faulty client or server in PBFT, Q/U, HQ, and Zyzzyva can render each of these systems effectively unusable for many applications by reducing their throughput by two orders of magnitude or more, from thousands of requests per second to fewer than 10 requests per second. The problem comes not because these systems fail to meet the guarantees they promise, but because the guarantees they promise are insufficient for the high assurance systems for which BFT techniques are likely to be of most interest.

In this paper, we describe Aardvark, a new BFT replication protocol that guarantees good performance during *uncivil* periods, when the network is reliable but when up to f servers and any number of clients are faulty. Aardvark gives up some performance compared to protocols that focus on optimizing for the best case, but Aardvark's peak throughput of 40527 requests per second seems sufficient for many applications. Because Aardvark is less aggressively tuned for the fault free case, it is guaranteed to remain within a constant factor of 40527 when faults occur. We observe throughputs of between 11706and 40527for 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 [8, 11, 1, 21], the reality is that they don't actually 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 of throughput and even long periods of complete unavailability. Such degradation or unavailability is unacceptable in many of the high assurance environments where BFT replication might otherwise be an attractive option.

For example, Figure 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. These slowdowns and crashes are caused by two distinct factors: (i) the protocol designs handle faults through alternate execution paths that are complex and can be an order of magnitude or more slower than the optimized best-case path and (ii) the implementations fail to cover several of the intricate corner cases that can arise along these slow paths. As we show later, a wide range of other behaviors—a faulty primary, a recovering replica server, etc.—can have similar impacts on performance.

The problem comes not because current BFT systems fail to meet the guarantees they promise, but because they promise insufficient guarantees for many high value systems. In particular, although these systems provide strong safety guarantees, they promise extremely weak liveness guarantees. For example, PBFT promises only that "clients eventually receive replies to their requests." [8]

We introduce Aardvark, a new BFT state machine replication protocol that continues to provide the strong safety guarantees of existing protocols but that also remains usable even when faults occur. In particular, Aardvark ensures strong liveness guarantees not only during *gracious* intervals—synchronous network, timely and fault-free replicas, correct clients—but also during *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.

Aardvark is a simple protocol that avoids introducing *fragile optimizations.* A fragile optimization is one that can improve best-case performance but that introduces expensive alternative protocol paths down which faulty nodes can send the system. More specifically,

1. *Aardvark limits vulnerability to disruption by clients* by using a hybrid signature/MAC authentication construct that safeguards request submission against manipulation by faulty clients and by iso-

lating client request processing so that load from clients cannot prevent progress by servers.

2. *Aardvark limits vulnerability to disruption by a primary* by defining an autonomous view change condition that determines when a server may unilaterally vote for a view change. Subject to this condition, Aardvark servers then implement a self-tuning view change protocol to keep performance near what would be provided during periods led by a fast, correct primary.

3. *Aardvark limits vulnerability to disruption by any server* by limiting the extra work servers can impose on each other by (a) only processing catch-up messages from slow servers when such processing cannot interfere with progress and (b) using dedicated physical links between each pair of servers and deactivating links from servers imposing excess load.

To eliminate fragile optimizations, Aardvark takes a number of steps that previous high-performance BFT systems have been wary of. In Aardvark, clients use signatures instead of MACs, and servers trigger frequent view changes. Surprisingly, these controversial choices impose only a modest cost on Aardvark's peak performance. As Figure 1 illustrates, Aardvark sustains peak throughput of 40527 requests/second, which is good enough for many interesting services. Notably, this throughput is within a factor of 3 of PBFT and 4 of Zyzzyva, comparable to QU and better than HQ. At the same time, Aardvark's fault tolerance is dramatically improved: for a broad range of client, primary, and server misbehaviors, Aardvark's performance remains between 11706and 40527requests/second.

In Section 2 we describe our system model and the guarantees appropriate for high assurance systems. In Section 3 we explore the limitations of existing systems and the root causes that preclude their use in high assurance environments. In Section 4 we present the Aardvark

System	Peak Performance	Big MAC Attack
PBFT [8]	36350	0
QU [1]	26786	0^*
HQ [11]	15873	SAFETY VIOLATION†
Zyzzyva [21]	48253	0
Aardvark	40527	40527

Figure 1: Observed Peak throughput of BFT systems in fault-free case and when a single faulty client submits a carefully crafted series of requests. We detail our measurements in Section 3. * is reported from [1]. † The HQ prototype does not allow servers to use distinct MAC keys.

protocol. In Section 5 we present an analysis of Aardvark's expected performance. In Section 6 we present our experimental evaluation. In Section 7 we discuss related work.

2 A new problem statement

This section defines our system model and argues that the demands of high assurance systems require us to rethink the liveness and performance objectives that BFT services should attempt to deliver.

To provide context, we first revisit the standard system model and safety guarantees, which Aardvark adopts as well. We then describe the liveness goals adopted by published BFT service replication protocols, discuss the limitations of these goals, and define new liveness/performance goals that we argue are a better match for most BFT services.

2.1 System model and safety guarantees

We take our basic model for the system's safety properties directly from prior work [21]. We assume the Byzantine failure model where faulty nodes (servers or clients) may behave arbitrarily [24]. We assume a strong adversary that can coordinate faulty nodes to compromise the replicated service. We do, however, assume the adversary cannot break cryptographic techniques like collision-resistant hashes, message authentication codes, encryption, and signatures. We denote a message X signed by principal Y's public key as $\langle X \rangle_{\sigma_Y}$. Our system ensures its safety and liveness properties if at most $f = \lfloor \frac{n-1}{3} \rfloor$ replicas are faulty. We assume a finite client population, any number of which may be faulty.

Our system implements a BFT service using state machine replication [8, 23, 22, 34]. Traditional state machine replication techniques can be applied only to deterministic services. We cope with the non-determinism present in many real-word applications (such as file systems [28] and databases [38]) by abstracting the observable application state at the replicas and using the agreement stage to coordinate all nondeterministic decisions [31].

Services limit the damage done by Byzantine clients by authenticating clients, enforcing access control to deny clients access to objects they do not have a right to, and (optionally) by maintaining multiple versions of shared data (e.g., snapshots in a file system [33, 32]) so that data can be recovered from older versions if a faulty client destroys data that it is authorized to access [20].

Our system's safety properties hold in any asynchronous distributed system where nodes are connected by a network that may fail to deliver messages, corrupt them, delay them, or deliver them out of order.

Under these assumptions, the safety property offered by Aardvark is a form of linearizability [17], modified, similar to [26] and [8], to account for Byzantine clients: the replicated service behaves as a single correct replica that executes requests atomically one at a time.

2.2 Standard liveness and performance properties

Since Aardvark attempts to provide useful performance despite faults, it is important to carefully define the timing model. Liveness and performance guarantees can not be made without assumptions that limit worst-case message delivery delays [14]. Given this fundamental limitation and given the initial presumption that BFT replication would be extremely expensive, published BFT state machine replication protocols have focused on two extreme cases, providing weak guarantees during intervals where weak assumptions hold and maximizing performance during intervals where strong assumptions about both timing and machine behavior hold [8, 1, 11, 21].

On the weak end, these systems focus on some variation of eventual synchrony, which can be defined with respect to a *synchronous interval*:

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.*

If we assume that network faults are eventually repaired, then it is a relatively weak assumption to presume that the time to deliver a message will not grow without bound and that eventually there will be an arbitrarily long synchronous interval for some bound T. Under such weak assumptions, systems can make the following (weak) liveness guarantee:

$L1$ If a correct client submits a request r, and if eventually there is a sufficiently long synchronous interval, then the service eventually executes r.

Although details vary, many systems attempt to provie some guarantee similar to $L1$ [4, 1, 8, 11, 21].

At the other extreme, systems have demonstrated excellent peak performance during *gracious intervals* and *semi-gracious intervals*.

Definition 2 (Gracious interval)**.** *An interval is gracious iff (a) the interval is synchronous with some implementation-dependent short bound on message delay and (b) all clients and servers behave as if they were correct.*

Definition 3 (Semi-gracious interval)**.** *An interval is semi-gracious iff (a) the interval is synchronous with some implementation-dependent short bound on message delay, (b) up to f servers crash, and (c) all clients and all remaining servers behave as if they were correct.*

This assumption yields the second "standard" performance property:

$L2_{std}$ Maximize performance during sufficiently long gracious or semi-gracious intervals.

Resulting efforts to maximize the best case performance have been important in combating the presumption that BFT is too expensive for practical use. These efforts have also been extremely successful—state of the art protocols can execute tens of thousands of requests per second and approach the performance of an unreplicated, non-BFT service [1, 8, 11, 21].

2.3 A case for a new goal

We argue that as important as properties $L1$ and $L2_{std}$ are, designing protocols around them has yielded systems that cannot meet the needs of many high-assurance systems. In particular, for many high assurance services, ensuring high availability may be nearly as important as ensuring integrity [12], so to truly be regarded as tolerating Byzantine faults, systems must continue to be useful even when faults occur and the system shows minimal graciousness.

Given that it is only possible to provide meaningful guarantees when the network is well behaved [14], we are interested in the performance of systems during *uncivil intervals* that impose significantly fewer restriction on the behavior of clients and servers.

Definition 4 (Well-behaved network)**.** *The network is well-behaved when messages sent by a correct node i to correct node j are received within a small time interval δ.*

Definition 5 (Uncivil interval)**.** *During uncivil intervals the network continues to be well-behaved, but an arbitrary number of clients and faulty servers can be Byzantine and renounce any pretension of graciousness.*

We then define a new requirement for our BFT replicated service:

$L3$ Provide useful performance during sufficiently long uncivil intervals.

Note that the threshold for what constitutes useful performance will depend on a service's anticipated demand, so our task is to design a protocol with good performance during uncivil intervals so as to maximize the range of workloads for which the system is of use. Additionally, we must engineer Aardvark to achieve long intervals where the network is well-behaved even if servers are faulty, so our prototype physically isolates servers' network connections.

In order to achieve $L3$, we are willing to relax the goal of $L2_{std}$—from maximizing best case performance, to providing adequate best case performance:

*L*2 Provide good performance during sufficiently long gracious and semi-gracious intervals

Discussion We argue that many BFT systems should be willing to give up some best case performance in order to provide good performance over a wider range of situations for two reasons.

First, in current systems the best case is fragile, so building a system around its best case performance may be dangerous. In particular, (1) the best case is achieved only when very strong assumptions hold and (2) departing from the best case can devastate performance because the system then provides at best the weak guarantee *L*1. This fragility is not just a theoretical problem. We observe both of these issues in existing protocols: as we show in Section 3, any server or client can knock the system off the best case and the resulting performance degradation can be enormous.

Second, many systems may be insensitive to modest reductions in peak agreement throughput because of *limited demand* or *other bottlenecks.*

In particular, many services' peak demands are far under the best case throughput offered by existing BFT replication protocols. For such systems, *good enough is good enough*, and modest reductions in best case agreement throughput will have little effect on end to end system performance. In such systems, increased robustness may come at effectively no cost.

Similarly, when systems have other bottlenecks, Amdahl's law limits the impact of changing the performance of agreement. For example, Zyzzyva can execute about 85,000 null requests per second [21], suggesting that agreement consumes 11.8μs per request. If, rather than a null service, we replicate a service for which executing an average request consumes 100μs of processing time, then peak throughput with Zyzzyva would be about 8945 requests per second. If, instead, agreement were accomplished via a protocol with double the overhead of Zyzzyva (e.g., 23.6μs per request), peak throughput would still be about 8090 requests/second. In this hypothetical example, doubling agreement overhead reduces peak end-to-end throughput by less than 10%.

Thus, Aardvark seeks to provide eventual progress (*L*1), good performance during gracious and semi-gracious intervals (*L*2), and useful performance during uncivil intervals (*L*3).

A non-goal. We deliberately reject an alternate goal—maximizing average performance.

One could imagine arguing that if a system spends a fraction g of its time in gracious intervals with throughput t_g and most of the rest of its time (1-g) in uncivil intervals with throughput t_u, then a system should be designed to maximize $gt_g * (1-g)t_u$. If one assumes $g \gg 1-g$, then one would focus on maximizing t_g and one would largely ignore t_u. I.e., one would focus on $L1$ and $L2_{std}$ and ignore $L3$ as in past systems.

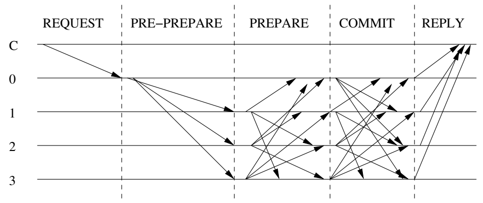

Figure 2: Basic communication pattern in PBFT [8].

We do not believe such a formulation of average performance captures the requirements of many of the high-assurance systems for which BFT replication might otherwise be attractive. The premise of BFT replication is that an important service might be willing to pay for replication to ensure integrity despite the presumably rare case of server failures. We similarly believe that at least some high-assurance services might be willing to pay some modest additional overheads to protect availability and performance.

3 Challenges

To understand the challenges BFT state machine replication systems face when trying to deliver good performance in the presence of faulty nodes, we look at each of the participants in turn: the clients, the primary (if present), and the replicas.

One might hope that in some high-assurance environments, servers might be well-enough controlled that exotic "malicious server" behaviors could be considered a tolerably low risk. Unfortunately, even if one were willing to give up the promise of BFT replication to tolerate arbitrary server failures, many existing protocols are sufficiently high-strung that they can be significantly disrupted by decidedly non-exotic behaviors including malformed requests from a single faulty client or catch-up messages from a slow or recently rebooted correct server.

As we note below, some of the individual challenges have been identified in the past; other challenges, once identified, have obvious solutions. Given the standard objective of maximizing best case performance, however, it is not surprising that existing systems retain a wide range of vulnerabilities to performance degradation by faulty nodes.

3.1 Basic operation of existing protocols

For review, Figure 2 illustrates the basic communication patterns in Castro and Liskov's PBFT protocol [8]. A set of $n \geq 3f+1$ servers select one of their number to be the *primary*. A client sends its REQUEST message to the primary, and the primary assigns the request a sequence number and sends a PRE-PREPARE message to the other servers. The servers then do an all-to-all exchange of PREPARE messages and then of COMMIT messages. Once a sufficient number of servers agree on the request's order in a linearizable total order of all requests, they execute the request and send a REPLY message to the client. A clients acts on the REPLY once it has at least $f+1$ matching replies.

To ensure progress, a client retransmits a request to all replicas if it does not receive a reply by a timeout, and the replicas forward the request to the primary. Each replica then expects the request to complete execution of a request by a timeout. If no requests complete execution in time, the replica assumes that the primary is faulty and initiates a *view change* by stopping all processing of messages in the current view and sending a VIEW-CHANGE message to all servers. Once a sufficient number of replicas initiate a view change, they are able to start the next view with a different primary. Additionally, if a server falls behind in this asynchronous system, it is able to catch up by fetching a recent checkpoint and recent messages from its peers.

A key performance optimization is the use of message authentication codes (MACs) for authentication rather than digital signatures. In particular, REQUEST, PRE-PREPARE, PREPARE, and COMMIT messages contain an *authenticator*—an array of n MACs, one for each server. For practical values of n, generating n MACs is at least an order of magnitude faster than verifying a signature. For example, on a 2.0GHz Pentium-M, openssl 0.9.8g can compute over 500,000 MACs per second for 64 byte messages, but it can only verify 6455 1024-bit RSA signatures per second or produce 309 1024-bit RSA signatures per second.[1]

Other representative protocols are similar in principle, but vary their message patterns. For example, Zyzzyva [21] speculatively executes requests, replies to clients after the PRE-PREPARE phase, and can skip the subsequent steps if enough replies match. Q/U [1] eliminates the primary and uses client retransmissions to resolve conflicting updates. HQ [11] is a hybrid protocol that resembles Q/U in the absence of contention and relies on a protocol like PBFT rather than client backoff to resolve conflicts.

3.2 Clients

The clients' basic role in replicated services is quite simple—they are expected to transmit messages to the servers so the servers can act. Client behaviors can thus be understood in two dimensions: message contents and message frequency.

[1]Elliptic curve algorithms have faster signature generation (e.g., 2275 per second for 160-bit signatures, which are believed to be approximately equivalent in strength to 1024-bit RSA signatures) but slower signature verification (e.g., 499/s for 160-bit signatures); most PBFT messages are generated once and verified $n-1$ times.

For message contents, we distinguish three classes of message. First, *correct* messages are properly constructed protocol messages that can be authenticated by all correct servers if they are accurately conveyed to their destinations. Second, *unfaithful* messages are protocol messages that would fail to be authenticated at one or more correct servers. Third, *other* messages do not purport to be protocol messages; *other* messages can be rejected by a server without any cryptographic processing.

Note that a *correct* (properly authenticated) message might contain a request that the underlying replicated service rejects due to an access control list (ACL) or other service-specific security violation. From the point of view of the replication protocol, such messages are still *correct*: the associated request should be executed at some well-defined point in the linearizable execution of the underlying state machine, which may choose to execute the request by generating an error code.

For message frequency, we must consider the load from *correct*, *unfaithful*, and *other* messages.

3.2.1 Unfaithful messages

Systems are vulnerable if they allow *unfaithful* messages to generate disproportionate amounts of work for servers. Although the use of MAC authenticators for client messages reduces the cost of processing *correct* messages, this technique can significantly increase the cost of handling *unfaithful* messages.

The problem with MACs arises because they fail to provide non-repudiation. As a result, even after checking a message's MAC, the recipient cannot be sure that any other node in the system will also consider the message and associated MAC to be valid. This fundamental limitation of MAC semantics is problematic because replicated services require correct replicas to ultimately make the same decision with respect to the validity of messages.

Note that if a message contains incorrect or inconsistent MACs, the fault may lie with the (purported) originator of the message, the network, any node through which the message was relayed, or (in the case of a message with no verifiable MACs) any other party. This frustrates many simple approaches to limiting the disruption from *unfaithful* messages. For example, if a PBFT or Zyzzyva replica receives a client request that it cannot authenticate, it faces a he-said/she-said problem: the replica cannot easily know if the fault lies with the client, primary, or network.

In existing protocols, clients send one or both of two classes of MAC-authenticated messages: (1) initial request transmission and (2) certificates that the client is expected to gather and retransmit to replicas. In both cases a faulty client can force the system down expensive paths by modifying a MAC authenticator to cause some servers to succeed in authenticating the message while others fail.

Request Transmission. Clients are responsible for sending requests to the servers. Because clients act alone and request submission is a necessary part of replicated services, this step presents an obvious chance for faulty clients to disrupt the system.

For example, in PBFT, as Castro notes in his thesis ([7], pp. 42–43), a single faulty client can transmit an *unfaithful* request that can force the system down execution paths that prevent progress for any client's request until a timeout and view change occur. In one such behavior, the client sends a request with a good MAC for the primary but a bad MAC for all other replicas. After authenticating the request, the primary assigns a sequence number s to the request and sends a PRE-PREPARE message to the replicas. The replicas, however, are unable to authenticate the request and refuse to generate a PREPARE message for sequence number s. At this point, no request with sequence number $s' > s$ can be executed, and progress stops until a timeout triggers a view change. The client can repeat its behavior in the new view.

Similar scenarios arise in Zyzzyva and HQ, which attempt to avoid suffering a view change by introducing a request validation sub-protocol that can require all-to-all communication and can require servers to generate signatures to combat the he-said/she-said problem.

Q/U avoids using the primary to order requests, but this omission allows the faulty client to frustrate servers by talking to them directly. In particular, a client can send a request that some replicas will accept and some will refuse, putting the system in an inconsistent state. Q/U relies on correct clients to eventually resolve such inconsistencies by reading state and the associated MAC authenticators from a quorum and then writing new state including MAC authenticators to a quorum. However, as Abd-El-Malek et al. note [1], progress is only ensured when clients follow an exponential backoff strategy [9].

Certificate Gathering. In addition to having clients issue requests, Q/U, HQ, and Zyzzyva require clients to gather certificates of replica responses before entering into the second phase of their protocols. Certificates generally consist of a threshold number of replica responses that the replicas must authenticate in order to decide what action to take in the next step of the protocol. When these replica messages are authenticated by MACs, a faulty client can produce unfaithful certificates that different subsets of replicas can authenticate.

Protocols respond to such situations by falling back on more expensive paths. For example, in HQ this client behavior results in the replicas signing a message as a token for the client to use in order to enter the conflict resolution phase of the protocol; in Q/U additional barrier operations and backoff is required; and in Zyzzyva an extended version of its slower 2-phase execution path is required.

Measured performance. As Figure 1[2] in Section 1 shows, a single client's big MAC attack can devastate performance in several representative existing systems. In this and all subsequent experiments, we use the most recent implementations provided by the systems' authors and run these systems on machines with dual-core 3GHz Intel Pentium-IV Xeon processors, 1GB of memory, and six 1Gb/s Ethernet NICs. We enable multicast for PBFT and Zyzzyva; multicast is not exploited by HQ or Q/U.

When we subject these systems [1, 8, 11, 21] to a load of 255 correct clients issuing back-to-back 1-byte requests that require no processing by the execution phase and one client that sends a series of *faithless* requests with inconsistent MACs, the throughput of the PBFT and Zyzzyva prototypes fall to zero. A big MAC attack cannot be implemented in the HQ prototype without violating safety as all clients and replicas share the same set of symmetric keys and the initial client request is authenticated by a single MAC. We were unable to replicate the expected performance of Q/U so rely on an analytic model discussed in [21]; we base the expected throughput on the authors' admission that the system "ensures safety, but not progress, in the face of malevolent components" [1].

The disparity between predicted and measured results arises from incomplete design and implementations. In particular, a big MAC attack stresses corner cases of the protocols that are not always fully specified and are frequently either left unimplemented or not fully tested to verify that they behave as expected. PBFT and Zyzzyva, for example, thrash on these poisoned requests, failing to either make progress in the current view or initiate a view change. The HQ prototype, for example, was constructed to faithfully model system performance for fault-free executions, and it expressly does not attempt to handle other cases; the HQ protocol design, in contrast with the implementation, is expected to be safe but the design does not help in understanding the system's performance in the presence of failures.

Design Principles Although it is perhaps not surprising that intricate alternative protocol paths fail to perform as expected, this experience suggests a second motivation for Aardvark's KISS approach: eliminating alternate protocol paths not only avoids the risk that clients can drive the system to use expensive paths, but it also makes it easier to produce a protocol that works as expected.

[2]For the moment, please ignore the intruding Aardvark; we discuss Aardvark's design and resilience in Section 4, but we include this line here to avoid repeating the table.

System	Peak Performance	Faulty Retransmission
PBFT [8]	36350	crash
HQ [11]	15873	0
Zyzzyva [21]	48253	crash
Aardvark	40527	7873

Figure 3: Observed peak throughput of BFT systems in the fault free case and under heavy client retransmission load.

3.2.2 Load from correct, unfaithful, and other messages

Performance is affected when correct or faulty clients send *correct* messages, when faulty clients send *unfaithful* messages, and when faulty clients send *other* messages. Figure 3 shows the impact of a single client that spams the replicas. In the case of PBFT and Zyzzya a single faulty client spams the replicas with 9kB messages; in HQ a faulty cliet spams the replicas with TCP connection requests.

The implementation decision that makes these protocols vulnerable to thrashing under high client load is that client requests and intra-server messages share a single FIFO network queue. As a result, increasing client request rate reduces the resources available for processing server requests.

Similarly, note that *unfaithful* (inauthentic) and *other* (non-protocol) messages can load the servers, crowd out legitimate client requests, or both. For example, if a client can spoof its source IP address, it can send large volumes of requests purporting to be from other clients, forcing the receiving server to cryptographically check and then reject these requests. Or, if an attacker controls a botnet, she can impose almost arbitrarily high levels of load.

Design Principles Such volume-based brute-force DoS behaviors are a potential problem for both BFT and non-BFT services, and solving the general DoS problem is beyond the scope of this paper. Formally, we will restrict our attention to periods when some fraction g of incoming packets carry *good* (non-DoS) requests and we will try to ensure that our throughput is within a factor of $(1-g)$ of our throughput if all requests were good. Other research efforts aimed at the volume-based DoS problem [6, 29] may help systems keep g high.

Aardvark's design goal is to avoid introducing vulnerabilities that make it significantly more susceptible to DoS or DDoS than non-replicated services. For example, requiring all client messages to be authenticated with a digital signature rather than a MAC [5] might increase vulnerability to DoS behaviors if verifying a signature is much more expensive than generating a message with a bogus signature.

System	Peak Throughput	1 ms	10 ms	100 ms
PBFT	36350	5396	4635	1097
Zyzzya	48253	14547	5141	crash
Aardvark	40527	38084	39089	37903

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

3.3 Primary Disruption

Employing a primary to order requests enables batching [8, 15] and avoids the need to trust clients to obey a backoff protocol [1, 9]. However, because primaries are 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 [4, 5] and the throughput observed by 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. To illustrate some of these challenges, we discuss two specific ways in which a primary can adversely impact system performance.

3.3.1 Delaying requests

A slow primary can significantly reduce system throughput by delaying requests. In particular, existing systems rely on a timer to initiate view changes when progress in the current view is not sufficient. Unfortunately, this timer is easily abused by a faulty primary that sends PRE-PREPARE messages late and that limits the rate at which it sends PRE-PREPARE messages. In Figure 4 we show the maximal throughput that can be achieved in a view where the primary delays sending the PRE-PREPARE message by varying times. We note that since the default view change timeout in PBFT and Zyzzyva is on the order of 500ms, the throughput of both systems is severely limited by the slow primary.

The Prime system [5] limits its vulnerability to slow primaries and inconsistent MACs by adding a pre-agreement stage that servers use to ensure that all servers have seen a request before it is ordered, by changing all MAC authenticators into signatures, and by enforcing a short timeout from when a server sees a request until it expects the leader to issue a pre-prepare message for that request. Unfortunately, as noted in Section 3.2.2, authenticating client requests with signatures may make it easier for clients or attackers to overload the system. Fur-

System	Peak Throughput	Faulty Throughput
HQ [11]	15873	0
PBFT [8]	36350	0
Zyzzyva [21]	48253	0
Aardvark	40527	11706

Figure 5: Observed peak throughput and observed throughput when one replica floods the network with 9k byte messages.

thermore, the signatures and extra phases come at a significant cost to throughput: Amir et al. measure a peak throughput of about 800 requests per second. The benefit of the approach is the ability to maintain a throughput of 400 requests per second despite a slow-primary. Aardvark's goal is to enjoy a similar ratio of best-case to uncivil throughput without paying such heavy costs in absolute performance.

3.3.2 Fairness Violation

A faulty primary might deny service to a victim client c by refusing to order requests from c. For example, in PBFT a client whose request is not satisfied retransmits the request to all replicas, and they set a timer and expect to see progress before a timeout. However, in accordance with liveness goal L1, replicas are satisfied with any progress, so they clear the timer when a retransmitted request for any client is processed.

3.4 Non-Primary Replica Load

The main avenue for a non-primary replica to limit performance is by imposing extra load. In particular, in BFT protocols, non-primary replicas do not initiate work and because the protocols are designed to tolerate faults, these systems typically continues to make rapid progress if a faulty replica is arbitrarily slow or omits sending one or more messages.

However, non-primary replicas can significantly slow their peers in two ways.

First, *correct* requests must be authenticated and processed, so retransmitted or extra *correct* request can increase load on peers. For example, even a correct server may sometimes impose significant extra load because, in an asynchronous system, it may fall behind in processing requests and then need to ask other servers to send a checkpoint of the system's state and recent requests. [18]

Second, *faithless* (inauthentic) or *other* (non-protocol) messages impose verification or I/O costs, even if they don't trigger protocol actions. While we might hope that network flooding behaviors by servers are rare, they do happen and they can allow one faulty node to bring down a collection of servers. For example, a faulty network card flooded the network and disabled a large group of immigration computers at LAX for 8 hours in 2007 [10, 36].

Figure 5 shows that one faulty server flooding the network with junk messages of 9k bytes has a significant impact on overall system throughput.

Design Principles Faulty replicas wreck havoc on system performance by imposing additional work on the non-faulty replicas in the system and interfering with the communication between non-faulty replicas. Aardvark's design goal is to limit the ability of faulty replicas to introduce extraneous work and interfere with the actions of non-faulty replicas.

4 Aardvark

The design and implementation of Aardvark emphasizes simplicity: we explicitly aim at a protocol that offers the same execution path during both gracious and uncivil interval with the dual goals of achieving good, predictable performance in all circumstances and of avoiding obscure corner cases. The prototype we have implemented is an extension of the recent new release of PBFT, which is currently the most stable of the BFT code bases, and follows the basic three round structure of Figure 2. In Section 4.4 we discuss how these principles can be applied to other protocols, including Zyzzyva. We now discuss how Aarwark addresses the challenges posed by clients, primary, and replicas that were identified in the previous section.

4.1 Trusting the Client

Clients can negatively affect the performance of correct replicas in two ways. First, they can craft unfaithful requests that force replicas to utilize execution paths that are slow, not carefully tested execution paths, or simply cause replicas to undertake a disproportionate amounts of work. Second, they can flood servers with their messages, drowning communication between replicas.

The second problem is easy to solve: Aardvark replicas simply listen to client requests and replica traffic on different network devices so that messages from clients and servers are placed in distinct network queues.

Aardvark addresses the first problem in two steps. First, it explicitly avoids relying on clients for anything other than just sending their requests. In particular, unlike Q/U, HQ, or Zyzzyva, Aardvark does not involve clients with gathering certificates of replica responses that can serve as input to the system. This "just the facts" approach eliminates a faulty client's ability to influence the execution path that the system will choose. Second, Aardvark requires clients to authenticate their requests using a hybrid MAC/signature construct. As we saw in Section 3.2.1, MAC-based authentication of requests leaves the system vulnerable to unfaithful messages; worse, it does not allow the system to determine where the blame for them lies. To gain non repudiation, we depart from the current BFT orthodoxy and reintroduce, judiciously, signatures in our protocol. Starting with PBFT, signatures have been (rightly) considered too expensive for practical use in BFT and have been relegated to the role of pedagogical tools for simplifying the exposition of a BFT protocol's structure. Aardvark limits the performance impact of signatures in three ways. First, it uses signatures only for authenticating client requests (all communication between replicas continues to be authenticated using MACs). Second, it uses the SFS implementation of the Rabin-Williams signature scheme [27] which places disproportionally the computational load on the signature generator (the client) rather than on the verifier (the server). Third, it combines signature with MAC authentication to defend against a potential denial of service attack that exploits the fact that, even with Rabin-Williams, signature verification is at least two order of magnitude slower than MAC verification. A faulty client could significantly slow down the system by intentionaly submitting requests with the wrong signature, which a servers would have to nonetheless verify.

The MAC portion of our hybrid MAC/signature construct protects Aardvark from this vulnerability. In Aardvark, a client who submits a request is expected to first sign the request and then to authenticates the signed request using a MAC. On receipt of a request, the server first uses the MAC to authenticate the sender. If the MAC is not valid, the server simply discards the request without further processing. Only if the MAC is successfully verified does the server proceed to verify the signature. If it finds the signature to be invalid, the server discards all future messages from that sender, preventing it from creating further spurious load.

4.1.1 Implementation Details

We leverage the increasing availability of multiple cores on commodity machines to mitigate the additional costs of using signatures to authenticate client requests by processing client requests on one core and messages received from replicas on the second core of our dual core machines. By doing this we are able to completely devote one core to the task of authenticating signatures, bringing our effective overheads more in line with existing systems (PBFT, Zyzzyva).

4.1.2 Experimental Evaluation

As Figures 1 and 3 show, the steps taken by Aardvark to mitigate faulty client behaviors are effective in increasing the attained throughput of the system.

4.2 Primary vulnerability

Aardvark orders requests using a primary. By enabling batching, this choice offers significant performance advantages, and by providing a single point of request serialization, it does not leave the system's liveness at the mercy of a client-driven backoff protocol during periods of contention. It does, however, risk leaving the system at the mercy of a faulty primary.

As we noted in Section 3.3, the fundamental challenge for primary-based BFT protocols is to somehow defend against the wide range of threats to both safety and liveness that can be brought by a faulty primary.

Hence, rather than designing specific mechanism to defend against each of these threats, past BFT systems [8, 21] 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 takes a more aggressive stance on view changes, regularly changing views following even minor indications that the system progress could improve. Aardvark relies on an adaptive throughput requirement to ensure that adequate long term progress is made during a view and a PRE-PREPARE heartbeat to ensure that once progress starts it continues.

These additional constraints on the primary result in regular view changes by the system. Surprisingly, these view changes do not have a significant negative impact on system performance. By changing views regularly, we prevent any individual primary from achieving tenure and encourage the current primary to work hard in order to stay in power as long as possible. We construct our view change guidelines to ensure that a primary providing adequate throughput remains primary for at least 5 seconds, and that a primary is allowed to make inadequate progress for at most 5 seconds.

4.2.1 Adaptive Throughput

When a new view starts, Aardvark defines a minimum acceptable throughput that the primary must maintain in order to remain in charge of the view. Aardvark measures the throughput of the system every time a checkpoint is taken, in our prototype this is every 128 batches. In order to measure throughput, Aardvark counts the number of requests executed since the last checkpoint was taken and records the time between checkpoints. These values define the observed throughput for that checkpoint interval; this observed throughput is compared against the required throughput, and if the primary is found lacking then the replicas call for a view change.

We impose an initial *grace period* during which the screws are not tightened and the required throughput does not increase. This ensures that well behaved primaries remain in charge for long enough to make useful progress during their view. In our prototype implementation, the grace period is 5 seconds.

In order to define the initial acceptable throughput we record the maximum throughput we observe in each view. We select the maximum value in the previous n views, and set our base initial required throughput to be 90% of this value. Every checpoint interval after the grace period expires, we increase the throughput requirement by 0.1% of the current required throughput.

4.2.2 PRE-PREPARE Heartbeat

The PRE-PREPARE heartbeat ensures that a primary makes nominal progress in the current view. A replica starts the heartbeat timer is sent following the first PRE-PREPARE received during the current view. If the timer expires before a checkpoint occurs, then the replica initiates a view change; otherwise the replica restarts the heartbeat timer at ech checkpoint interval. The heartbeat timer guarantees that, once a view is started, a primary makes consistent progress towards the next checkpoint interval and ocrresponding throughput check. We set the heartbeat timer to ensure that the maximum time between a pair of checkpoint intervals is identical to the grace period. This step bounds the amount of time that a primary can provide throughput that is considered unacceptable before a view change occurs.

4.2.3 Fairness

When replicas receive a request from a client that they have not seen in a PRE-PREPARE message, they add the message to their request queue and record the sequence number k of the most recent PRE-PREPARE that they have received during the current view before forwarding the request to the primary. The replica monitors future PRE-PREPARE messages for that request, and if it receives a PRE-PREPARE for sequence number $k+c$ where c is the number of clients before receiving a PRE-PREPARE that includes a request from that client then it declares the current primary to be unfair and initiates a view change.

4.2.4 Experimental Evaluation

Figure 4 demonstrates that the adaptive throughput techniques employed by Aardvark effectively diffuse any attempts by the primary to delay ordering of batches.

4.3 Spurious Servers

Replicas impose load on each other based on the messages that they send. This load can come in the form of legitimate requests allowed by the protocol, possibly to account for network failures or reconcile inconsistencies introduced by faulty clients and servers, and generic network flooding.

We limit the impact of catch up messages required in the former case by de-prioritizing catch up work. If the system is making adequate progress, then processing catch up messages is unnecessary. The key observation here is that it is *OK* for f replicas to be arbitrarily far behind and not participate as long as the system is in fact making progress. It is only necessary to bring this replicas up to speed when their participation is needed in order to ensure that adequate progress continues. Notice that if a replica's participation is needed for progress, other messages stop, and these catchup messages are processed.

We limit the load imposed by generic network flooding by relying on distinct network devices to communicate with each replica. This step takes our decision to explicitly separate the clients from the replicas to its natural conclusion. By handling traffic from each replica on separate devices, we prevent replicas from drowning each other out as each device relies on its own network queue. We are also able to handle incoming messages fairly, selecting incoming messages from replicas in a round robin fashion when necessary. This is especially convenient in Aardvark as each instance of consensus requires a server to receive exactly two messages from each other server.

4.3.1 Experimental Evaluation

Figure 5 shows that the steps taken by Aardvark effectively isolate replicas from eath other and mitigate the impact of faulty replica behaviors on system throughtput.

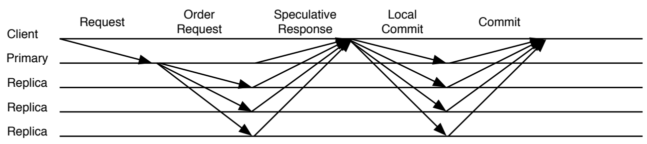

Figure 6: Basic communication pattern in Zyzzyva [21].

4.4 Zyzzyva Operation

While the structure of current Aardvark prototype more closely resembles PBFT, the principle that guides our design—eliminate fragile optimization—can be equally applied to other BFT systems. Specifically, Aardvark could be adapted to incorporate Zyzzyva style speculative execution.

For review, the communication pattern for Zyzzyva is included in Figure 6. In Zyzzyva, the client sends a request to the primary who in turn forms a batch and sends an ORDER-REQUEST for that batch to the replicas. Upon receipt of an ORDER-REQUEST, replicas execute each request specified by the ORDER-REQUEST and send a SPECULATIVE-RESPONSE to the client. The client gathers SPECULATIVE-RESPONSES from replicas until it has received $3f+1$ matching responses indicating that the request has executed and is stable. If the client does not receive $3f+1$ SPECULATIVE-RESPONSES in a timely manner and has received a certificate of $2f+1$ SPECULATIVE-RESPONSES, then the client can initiate the slow path of the protocol corresponding to the PBFT PREPARE phase. Replicas receive the LOCAL-COMMIT certificate from the clients and, if it is complete, reply with a COMMIT message; clients accept a response after having received $2f+1$ matching COMMIT messages.

The most obvious step in Zyzzyva-izing Aardvark is to incorporate speculative execution and require replicas to execute requests and send responses to clients immediately after receiving a PRE-PREPARE message. The more subtle steps relate to how we address the slow path through the protocol and I-HATE-THE-PRIMARY messages during view changes.

As we saw in Section 3.2.1, relying on clients to gather and faithfully relay certificates is tenuous and can require complicated conflict resolution. Rather than requiring clients to initiate the second phase, a Zyzzyva-ized Aardvark would execute only the speculative phases for the vast majority of instances of consensus. Every checkpoint interval, Zyzzyva-ized Aardvark would execute the full PBFT agreement protocol: PRE-PREPARE, PREPARE, COMMIT. Executing the third phase ensures that clients can eventually act on $f+1$ matching responses, and the Aardvark PRE-PREPARE heartbeat ensures that requests are committed promptly, either by reaching the checkpoint interval quickly or through the view change process.

A second benefit of scheduling the full PBFT execution path on a regular basis is that replicas that suspect the primary to be faulty are no longer required to stay active in the current view in order to maintain progress. Recall that a replica initiates a view change in Zyzzyva by first stating I-HATE-THE-PRIMARY; only after hearing that f other replicas are also interested in changing views can the replica drop out of the view. Until that happens the replica must stay up to date in the view, a prospect that can impose signficant load on all servers when the primary is faulty.

5 Analysis

We explicitly design Aardvark to reduce the number of execution paths that the system can take, whether it is operating during gracious or uncivil intervals. This simplicity not only increases Aardvark's robustness, but also makes it possible to model its behavior analytically.

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

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 crypto operations—verifying signatures, generating MACs, and verifying MACs, requiring θ, α, and α respectively. Since these operations track closely message transmission and reception, 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 executes within a gracious interval. To assess the loss in throughput incurred by Aardvark during uncivil intervals, we proceed in two steps. First, we bound the throughput during uncivil views in which the primary is correct. Then, we show that Aardvark limits the additional drop in throughput that can be caused by faulty primaries.

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 PrePrepare and verifies $n-1$ MACs upon receipt of the Prepare messages; replicas instead verify one MAC in the primary's PrePrepare, 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 per second. □

Lemma 1. *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 α/θ 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 b/g signatures and MACs, and correct replicas b/g signatures and an additional $(b/g)(1-g)$ 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 at least $\frac{g\kappa}{\theta + \frac{(b(1+g)+4g(n-1+f)}{b}\alpha}$ requests per second: as the ratio α/θ tends to 0, the ratio between the uncivil and gracious throughput approaches g. $\square$

Theorem 2. *For sufficiently long uncivil intervals and for small f the throughput of a properly configured Aardvark is within a constant factor of its throughput in a gracious interval in which primaries use the same batch size.*

Proof. First consider the case in which all the uncivil views have correct primaries. 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 α/θ 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 f/n of its views will have a Byzantine primary. Aardwark's PrePrepare heartbeat 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 throughpiut expected of a primary at the beginning of a view is a constant fraction of the maximum throughput achieved by the primaries of the last $f+1$ views, faulty primaries 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 $\frac{t_{gracePeriod}}{t_{maxViewTimeout}+t_{gracePeriod}} \frac{n-f}{n}$

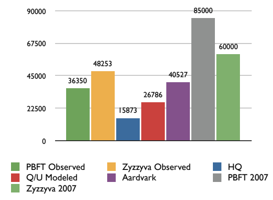

Figure 7: The throughput of various RSMs as the number of clients increases.

of that of Lemma 1, and, as α/θ tends to 0, the ratio between the throughput during uncivil and gracious intervals approaches $g\frac{t_{gracePeriod}}{t_{maxViewTimeout}+t_{gracePeriod}} \frac{(n-f)}{n}$. $\square$

6 Evaluation

All experiments are carried out on machines with dual 3GHz Intel Pentium-IV Xeon processors, 1GB of memory, and 1Gb/s Ethernet.

All numbers reported are based on executig the codebases provided by the respective systems' authors in our environment. Note that because we could not acquire measurements consistent with those reported by Q/U [1] in our environment, here we report peak throughput values scaled from [1] based on CPU frequency.

We note that the peak throughput numbers we have observed for PBFT and Zyzzyva are consistently $\frac{1}{2}$ the throughput we observe on the same machines in September 2007 as we prepared [21]. For reference, we include Figure 7 which reports the numbers from [21]. To date we have been unable to identify the precise cause of the performance degredation; the known changes in the experimental set up are (1) enabling the machines as full emulab [37], (2) transition from Debian to Red Hat Linux, and (3) replacement of the SFS library with SFS-lite. We are continuing our efforts to identify the source of the difference, but believe that the relative results we record are accurate as Zyzzyva and Aardvark are both based on the PBFT code base.

As noted in the figures in Section 3, existing systems perform poorly in the presence of faults but aardvark performs well. Aardvark's robustness comes largely from avoiding fragile optimizations that both complicate designs and that provide ways for unexpected node behaviors to drive the system down expensive paths.

System	Peak Performance
Aardvark	40527
PBFT	36350
PBFT w/ client signatures	23361
Aardvark w/o signatures	50852
Aardvark w/o adaptive throughput	39771
Aardvark w/o separate replica NICs	*xx*30000

Figure 8: Peak throughput of Aardvark and incremental versions of the Aaarvark protocol

Surprisingly, this robustness comes at only a modest cost to peak throughput. The peak throughput of Aardvark is 80% of the peak throughput we observe for Zyzzyva and outperforms our observed throughput for PBFT, HQ, and Q/U. For the higher throughputs of PBFT and Zyzzyva reported in [21], we are within a factor of 1.5 and 2.5 respectively. The absolute performance provided by Aardvark appears sufficient to be useful in many environments.

The rest of this section dissects the cost of each of Aarvark's key design decisions. Figure 8 reports the peak throughput of PBFT, Aardvark, and several conceptually intermediate steps between the two systems.

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 modest 20% hit to system throughput. Aardvark pays a smaller cost for incorporating signatures thanks to the usage of the second core on the machine. Utilizing the second core masks the additional computational costs associated with verifying signatures.

Peak throughput for Aardvark with and without the adaptive throughput timers is equivalent and within the experimental error. The reason for this is rather straightforward: when both the new and old primaries are nonfaulty, a view change requires the same amount of work as a single instance of consensus. Each view in Aardvark consists of 6000 instances of consensus, so the additional costs associated with view changing are minimal.

We observe a moderate performance boost by isolating the replicas to their own NICs. This performance boost results from our ability to better schedule incoming requests and ensure that we get the most relevant messages next.

7 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. [35] show that PBFT [8], Q/U [1], HQ [11], and Zyzzyva [21] are all sensitive to network performance. They provide a thorough examination of the gracious executions of the four canonical systems through a ns2 [30] 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. [4] and Amir et al [5] note that a slow primary can result in dramatically reduced throughput. We discuss the details of Amir et al. in Section 3.3.1. Aiyer et al. combat this problem by frequently rotating the primary.

PBFT [8], Q/U [1], HQ [11], and Zyzzyva [21] all include at least on graph covering the behavior under the systems in the presence of up to f failed replicas. The failures considered in these cases are benign crash failures in which the server does not participate in the protocol. These experiments demonstrate that the systems require only $n-f$ servers to make progress. We instead look at faulty behaviors that stress paths of the protocol that are considered to be corner cases and outside of the "common case" execution path. We show that it is possible to maintain good performance in both the presence and absence of failures. Hendricks et al. [16] explore the use of erasure coding to make BFT replicate storage more efficient; their work emphasizes increasing the bandwidth and storage efficiency of a replication protocol similar to Q/U and not the fault tolerance of the underlying protocol.

A number of researchers have explored the impact of weakening or strengthening timing assumptions for distributed protocols. Keidar and Shraer [19] propose a general approach for evaluating the impact of different timing assumptions on consensus performance. Aguilera et al. [2] and Malkhi et al. [25] explore the limits of what assumptions are needed for liveness for consensus and leader election. Conversely, Aguilera et al. [3] explore how small strengthenings on timing assumptions can yield algorithms more suitable for real-time, mission-critical systems, and Dutta et al. [13] explore how quickly consensus can be achieved under eventual synchrony.

8 Conclusion

We claim that high assurance systems require BFT protocols that are more robust to failures than existing systems. 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, but provides several orders of magnitude improvement in throughput during uncivil intervals.

References

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable byzantine fault-tolerant services. In *Proc. 20th SOSP*, Oct. 2005.

[2] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Consensus with byzantine failures and little system synchrony. In *DSN 2006*, pages 147–155, Washington, DC, USA, 2006. IEEE Computer Society.

[3] M. K. Aguilera, G. L. Lann, and S. Toueg. On the impact of fast failure detectors on real-time fault-tolerant systems. In *DISC 2002*, pages 354–370, London, UK, 2002. Springer-Verlag.

[4] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. BAR fault tolerance for cooperative services. In *Proc. 20th SOSP*, Oct. 2005.

[5] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Byzantine replication under attack. In *DSN 2008*, 2008.

[6] G. Badishi, I. Keidar, and A. Sasson. Exposing and eliminating vulnerabilities to denial of service attacks in secure gossip-based multicast. In *Proc. DSN-2004*, page 223, Washington, DC, USA, 2004. IEEE Computer Society.

[7] M. Castro. *Practical Byzantine Fault Tolerance*. PhD thesis, Jan. 2001.

[8] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 2002.

[9] G. Chockler, D. Malkhi, and M. Reiter. Backoff protocols for distributed mutual exclusion and ordering. In *ICDCS-21*, pages 11–20, 2001.

[10] Contingency planning, for technology and terrorism. http://www.washingtonpost.com/wp-dyn/content/article/2007/08/15/AR2007081502282.html.

[11] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proc. 7th OSDI*, Nov. 2006.

[12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, 2007.

[13] P. Dutta, R. Guerraoui, and L. Lamport. How fast can eventual synchrony lead to consensus? In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 22–27, 2005.

[14] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[15] R. Friedman and R. V. Renesse. Packing messages as a tool for boosting the performance of total ordering protocls. In *HPDC '97: Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing*, page 233, Washington, DC, USA, 1997. IEEE Computer Society.

[16] J. Hendricks, G. R. Ganger, and M. K. Reiter. Low-overhead byzantine fault-tolerant storage. In *SOSP 2007*, pages 73–86, New York, NY, USA, 2007. ACM.

[17] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[18] K. Keeton, D. Beyer, E. Brau, A. Merchant, C. Santos, and A. Zhang. On the road to recovery: restoring data after disasters. *SIGOPS Oper. Syst. Rev.*, 40(4):235–248, 2006.

[19] I. Keidar and A. Shraer. Timeliness, failure-detectors, and consensus performance. In *PODC 2006*, pages 169–178, New York, NY, USA, 2006. ACM.

[20] S. T. King and P. M. Chen. Backtracking intrusions. *SIGOPS Oper. Syst. Rev.*, 37(5):223–236, 2003.

[21] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative byzantine fault tolerance. In *Symposium on Operating Systems Principles (SOSP)*, Oct. 2007.

[22] R. Kotla and M. Dahlin. High throughput Byzantine fault tolerance. In *Conference on Dependable Systems and Networks, DSN'04*, June 2004.

[23] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 1978.

[24] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 1982.

[25] D. Malkhi, F. Oprea, and L. Zhou. Omega meets paxos: Leader election and stability without eventual timely links. In *The 19th Intl. Symposium on Distributed Computing (DISC)*, Sept. 2005.

[26] D. Malkhi, M. Reiter, and N. Lynch. A correctness condition for memory shared by byzantine processes. Unpublished manuscript, Sept. 1998.

[27] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proc. 17th SOSP*, pages 124–139, Dec. 1999.

[28] S. Microsystems. Nfs: Network file system protocol specification, 1989.

[29] D. Moore, C. Shannon, D. J. Brown, G. M. Voelker, and S. Savage. Inferring internet denial-of-service activity. *ACM Trans. Comput. Syst.*, 24(2):115–139, 2006.

[30] The Network Simulator NS-2. `http://www.isi.edu/nsnam/ns/`.

[31] R. Rodrigues, M. Castro, and B. Liskov. BASE: using abstraction to improve fault tolerance. In *Proc. 18th SOSP*, pages 15–28. ACM Press, Oct. 2001.

[32] D. J. Santry, M. J. Feeley, N. C. Hutchinson, and A. C. Veitch. Elephant: The file system that never forgets. In *Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems*, page 2. IEEE Computer Society, 1999.

[33] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the elephant file system. *SIGOPS Oper. Syst. Rev.*, 33(5):110–123, 1999.

[34] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, Sept. 1990.

[35] A. Sing, T. Das, P. Maniatis, P. Druschel, and T. Roscoe. Bft protocols under fire. In *NSDI*, 2008.

[36] Single network card downed lax computers. http://www.tgdaily.com/content/view/33398/113/.

[37] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. 5th OSDI*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.

[38] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *ICDCS 2000*, page 464, Washington, DC, USA, 2000. IEEE Computer Society.