Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults

Lorenzo Alvisi[1], Allen Clement[1], Mike Dahlin[1]
and Mirco Marchetti[2], Edmund Wong[1]

Abstract: This paper argues for a new approach to building Byzantine fault tolerant systems. We observe that although recently developed BFT state machine replication protocols are quite fast, they don't actually 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 new principles to construct a new protocol, Aardvark, which can achieve peak performance within 25% of that of the best existing protocol in our tests and which provides a significant fraction of that performance when the network is well behaved and up to f servers and any number of clients are faulty. We observe useful throughputs between 11706 and 38667 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, 10, 13, 20], 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 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?

System	Peak Performance	Faulty Client
PBFT [10]	61710	0
QU [1]	23850	crash
HQ [13]	7629	inc†
Zyzzyva [20]	65999	0
Aardvark	38667	38667

Fig. 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 7.2. † The HQ implementation does not implement the error handling steps necessary to protect against the desired faulty client behavior.

To illustrate the extent of the problem, 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. As we show later, a wide range of other behaviors—faulty primaries, recovering replicas, etc.—can have a similar impact on performance. We believe that these bona-fide 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 in practice the protocols are even more vulnerable than they appear on paper.

The primary contribution of this paper is to put forward a case for a new approach 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 to offer acceptable and predictable performance under the broadest possible set of circumstances—including when faults occur.

The last row of of Figure 1 shows the performance of Aardvark, a new BFT state machine replication protocol whose design and implementation are guided by this new philosophy.

Taking inspiration from the restricted maxi-min strategy of maximizing the worst case payoff across all games [28], Aardvark optimize system performance across 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.

In some ways, Aardvark is very similar to traditional BFT protocols: Clients send request to a primary who relays requests to the replicas who agree on the request before responding to the client just like PBFT, Zyzzyva, HQ, QU, ZZ, Paxos, Mencius, Scrooge, High throughput BFT, etc. In other ways Aardvark is very different.

Aardvark utilizes signatures for authentication where previous systems have gone to great pains to avoid them; Aardvark performs regular view changes where previous systems have treated view changes as an option of last resort; Aardvark relies on point to point communication where previous systems have received significant benefits from utilizing IP multicast.

These design decisions directly challenge conventional wisdom. As Castro observes, "eliminating signatures and using MACs instead eliminates the main performance bottleneck in previous systems [29, 22]." [9]. All view changes distract the system from processing new requests—regular view changes institutionalize overhead. To renounce IP-multicast is to give up throughput, deliberately.

Surprisingly, Aardvark's counter-intuitive choices impose only a modest cost on its peak performance. As Figure 1 illustrates, Aardvark sustains peak throughput of 38667 requests/second, which is within 20% 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.

Once again, however, the main contribution of this paper is not the Aardvark protocol itself. It is instead a new approach that can—and we believe should—be applied design of other BFT protocols. In particular, we (1) demonstrate that existing protocol are fragile, (2) argue that protocols should be designed using the restricted maxi-min criteria, and (3) demonstrate by constructing Aardvark that the restricted maxi-min approach is viable: we gain qualitatively better robustness at only modest cost to best-case performance.

In Section 2 we describe our system model and the guarantees appropriate for high assurance systems. In Section 3 we expand on the need to rethink Byzantine fault tolerance and identify a new set of design principles for BFT systems. In Section 4 we present an overview of the design of Aardvark. In Section 5 we describe in detail the important portions 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) may behave arbitrarily [23] and 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 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, a an array of MACs appropriate for verification by every replica, as $\langle X \rangle_{\vec{\mu}_r}$

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.

We assume an asynchronous network and the existence of *synchronous intervals* during which messages are delivered with a bounded delay.

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 theoretical 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 [15], 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 [22], are at the core of Castro and Liskov seminal work on PBFT [9]. 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" [24]. 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 [17], 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 [20], several new protocols have further improved on that mark [17, 30].

Despite these impressive results, we argue that the current practice of aggressively tuning BFT systems for the common case of gracious execution, a practice that we have engaged in with relish [20] is increasingly misguided, dangerous, and even futile.

It is misguided, because it encourages the design 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 satisfied with guaranteeing "some progress" in the presence of Byzantine failures, effectively retreating to the weak liveness guarantee imposed by FLP. 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 attack can hardly be said to *tolerate* Byzantine faults.

It is dangerous, because it encourages *fragile optimizations*— design choices that speed up the common case at the cost of exposing the system to the potential for more Byzantine attacks. 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 permanently the system off its hard-won optimized execution path and enter a 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.

We believe instead that the engineering of BFT systems should be guided by a new principle, inspired by the maxi-min strategy of maximizing the worst case payoff across all possible games. 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 server. In particular, the temptation of fragile optimizations should be resisted: according to our maxi-min principle, 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.

Clearly, a literal interpretation of the maxi-min principle is meaningless in an asynchronous system, where FLP tells us the worst case does not guarantee any liveness. This is no excuse to cling to gracious executions only, however. In particular, there is no theoretical reason why BFT systems shouldn't be expected perform adequately 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 BFT systems around a restricted maxi-min strategy aimed at maximizing the system's 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 restricted maxi-min: as a data point, Aardvark, the new protocol based on restricted maxi-min that we describe in the rest of the paper, 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 about 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 100μs of processing time, then peak throughput with PBFT would be about 8613 requests per second. If, instead, agreement were accomplished via a protocol with double the overhead of PBFT (e.g., 32.2μs per request), peak throughput would still be about 7564 requests/second. In this hypothetical example, doubling agreement overhead reduces peak end-to-end throughput by about 12%.

4 Aardvark: BFT through restricted maxi-min

Aardvark is a new BFT protocol designed according to the restricted maxi-min principle. Aardvark consists of 3 stages: request distribution, agreement, and view change.

This is the same basic structure of PBFT [10] and its direct descendants [21, 35, 34, 20, 6], but Aardvark reexamines it with the intent of providing acceptable performance to the broadest set of executions. Because of this, we decided to model the basic structure of Aardvark after PBFT—the BFT protocol with the most expansive notion of what defines a gracious execution. Every step of the protocol, however, is revisited with the goal of achieving an execution path that satisfies the three properties outlined in the previous section: acceptable performance, ease of implementation, and robustness against Byzantine disruptions.

The three key ideas that Aardvark relies upon are (1) signed client requests, (2) resource isolation, and (3) regular view changes.

Signed client requests. 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) MAC authenticators [4] to authenticate client requests.

As we mentioned in the Introduction, digital signatures are generally seen as too expensive to use. Aardvark only uses them for client requests. Replica-to-replica and replica-to-client communication rely on MAC authenticators. Replica communication does not introduce dangerous corner cases because it is quorum driven—while a single replica may be faulty, the quorum is collectively guaranteed to be correct, ensuring that the system stays on the correct path. Using MAC authenticators for replica messages is important because of the asymmetric costs of signature schemes; signature generation is significantly less expensive than verification.

Because of the additional costs associated with verifying signatures in place of MACs, Aardvark needs to guard against new denial-of-service attacks where the system receives a large numbers of requests with signatures that need to be verified are received. Our implementation (1) uses 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) forces a client to complete one request before issuing the next, limiting the number of correct signature verifications a client can inflict on the system.

Resource isolation. Aardvark explicitly isolates network and computational resources.

As illustrated by Fig. 2, Aardvark uses separate NICs and wires to connect each pair of replicas. This is necessary to prevent a faulty node from breaking the "good network" assumption, as happened when a single broken NIC shut down the immigration system at LAX [11].

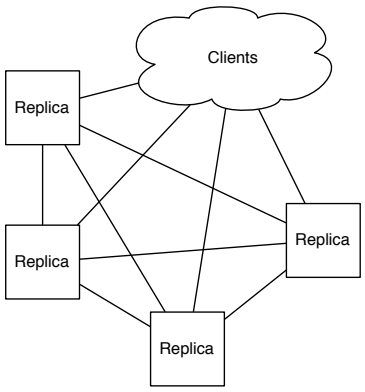

Fig. 2: Physical network in Aardvark.

As Figure 3 shows, Aardvark uses separate work queues for 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 handling fairly, ensuring that a replica is able to gather efficiently the quorums it needs to make progress.

Past protocols achieve significant throughput gains from hardware multicast because of the quantity of all-to-all communication in replication protocols. The decision to abandon multicast incurs a performance hit, as shown in Section 7, and limits fault scalability as the number of network connections required by the system is $O(f^2)$. Smarter network controls with rate-limited multicast and resource reservations are potential paths to reducing this overhead while providing the necessary isolation. Such approaches, while still requiring redundant network components to avoid a single point of failure, may reduce overheads below $O(f^2)$.

Isolating computational resources allows Aardvark to leverage separate hardware threads to process incoming client and replica requests. Taking advantage of inherent hardware parallelism allows Aardvark to reclaim part of the costs paid to verify signatures on client requests.

Regular view changes. In order 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 over time. If the current primary fails to provide this required amount of 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

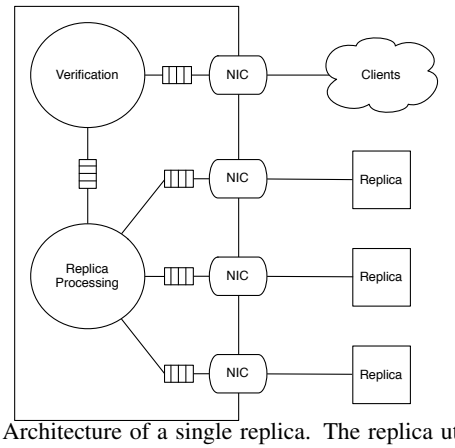

Fig. 3: Architecture of a single replica. The replica utilizes a separate NIC for communicating with each other replica and a final NIC to communicate with the collection of clients. Messages from each NIC are placed on separate worker queues.

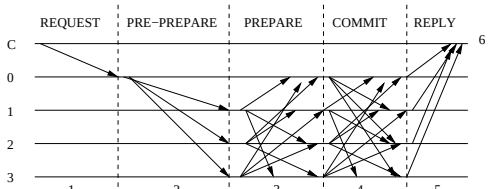

Fig. 4: Basic communication pattern in Aardvark.

degradation due to a faulty primary.

2. Eventual progress is guaranteed when the system is eventually synchronous.

Past protocols 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. Performing view changes regularly introduces 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.

5 Protocol description

Figure 4 shows the agreement phase communication pattern that Aardvark shares with PBFT. Variants of this pattern are employed in other recent BFT RSM protocols [1, 13, 17, 20, 30, 34, 35]. We organize the following discussion around the major steps of the protocol. These steps correspond to the numbered steps in Figure 4.

5.1 Request distribution

The fundamental challenge in the request-distribution phase of Aardvark is transferring requests from the clients to the servers. It is extremely important that, upon receiving a request, every replica is able to make the same decision about the authenticity of the request. We ensure this property by signing requests.

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 for the intended recipient with $\mu_{c,r}$.

Upon receiving a client request, a replica verifies the request. Request verification takes the following sequence of steps as illustrated by Figure 5:

1. **Blacklist check.** If the sender c is not blacklisted, then proceed to step 2. Otherwise discard the message.
2. **MAC check.** If $\mu_{c,p}$ is valid then proceed to step 3. Otherwise discard the message
3. **Sequence check.** Examine the most recent cached reply to c. If the request sequence number s is the next sequence number expected from the client c proceed to step 4. Otherwise

   3a. **Retransmission check.** Replicas utilize an exponential backoff to limit the rate of client reply retransmissions. If a reply has not been sent to c recently, retransmit the last reply sent to c. Otherwise discard the message.
4. **Redundancy check.** Examine the most recent cached request from c. If no request from c with sequence number s has previously been verified, then proceed to step 5. Otherwise

   4a. **Once per view check.** If an identical request has been verified in a previous view, but not processed during the current view, then process the request. Otherwise discard the message.
5. **Signature check.** If σ_c is valid process the request. Otherwise blacklist the node x that authenticated $\mu_{x,p}$ and discard the message.

Primary and non-primary replicas process 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. Non-primary replicas forward each request to the current primary once; replicas discard any requests it has previously received. Note that non-primary replicas will retransmit requests multiple times provided that a view change occurs between retransmissions.

Note that a REQUEST message that is verified as authentic might contain an operation that the underlying replicated service rejects due to an access control list (ACL) or other service-specific security violation. From

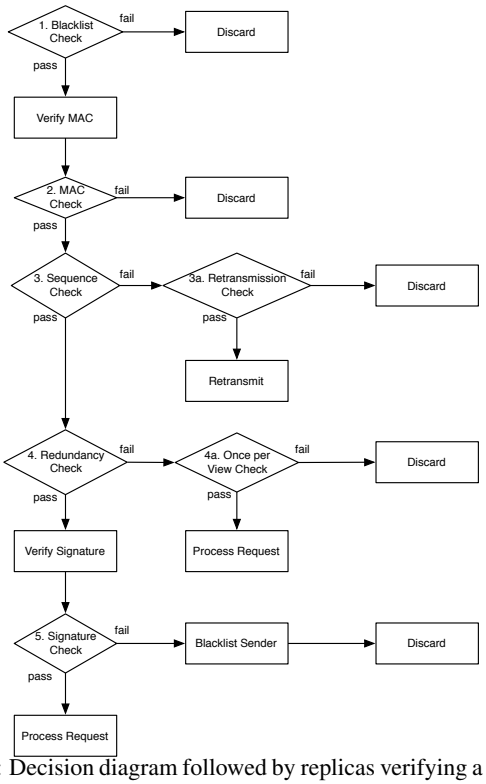

Fig. 5: Decision diagram followed by replicas verifying a client request.

the point of view of Aardvark, such messages are valid and should be executed at a well-defined point in the linearizable execution of the underlying service, which may execute the request by generating an error code, for example.

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 c has suffered a significant fault or 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.

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 seperate worker queues for 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.

We deploy our prototype implementation on dual processor machines. We assign one core to verify client requests and the second to run the replica protocol. This explicit assignment not only allows us to isolate resources but also to take advantage of parallelism to mask the additional costs of signature verification.

Maxi-min analysis: Our focus in applying the restricted maxi-min design principle to Aardvark is ensuring that the performance of Aardvark is good when the network is well behaved. We evaluate the effectiveness of our design by systematically considering the costs a faulty client can impose on the system. Two imporant factors in this analysis are the costs of individual messages and of collections of messages.

There are five actions that can consume significant replica resources: receive message, verify MAC, retransmit cached reply, verify signature, and process request. The costs a replica pays to process a client request increase as the request passes each successive check in the verification process, but the rate at which a client can trigger these costs becomes more restricted at each step.

Starting from the final step of the decision diagram, 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 such requests can trigger expensive signature checks and processing to the maximum rate a correct client can submit such requests. The sequence check (3) 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 (4) prevents repeated signature verifications for the same sequence number by caching each client's most recent request. The exactly once check (4a) permits repeated processing of a request only across different views to ensure progress. Finally, the signature check (5) ensures that only requests that will be accepted by all correct replicas are processed.

Moving up the diagram, replicas respond to retransmission of completed requests paired with valid MACs by retransmitting the most recent reply sent to that client. The retransmission check imposes an exponential backoff on retransmissions, limiting the rate at which the retransmissions actually occur. 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.

MAC verifications occur on every incoming message that claims to have the right format unless the sender is blacklisted, so processing to receive messages and verify MACs are effectively limited only by the network's ability to deliver messages.

Protecting against a client capable of flooding the network is beyond the scope of this paper, but it is an area of active research [32, 25]. Our goal is to ensure that Aardvark's replication protocol does not increase its vulnerability to client attack. Like existing systems, the best that

Aardvark can hope for under network flooding attacks is to maintain throughput in proportion to the percentage of correct client requests that are delivered to the replica. Aardvark achieves that goal through the early discard of messages and explicitly limiting the rate at which client induced retransmissions and expensive verifications occur.

5.2 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 [10]. Aardvark follows the restricted maxi-min design principle by adding explicit resource scheduling to limit a faulty replica's ability to slow progrss. The protocol continues from step 2 of the communication pattern in Figure 4.

> 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. Note that although we show a single request as part of the PRE-PREPARE message, multiple requests can be batched in a single PRE-PREPARE [10, 16, 20, 21].

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

Upon receipt of a $\langle$PRE-PREPARE, $v,n,\langle$REQUEST$,o,s,c\rangle_{\sigma_c}\rangle_{\vec{\mu}_p}$ message from primary p, replica r verifies the authenticity of the message. The verification process r follows to verify the PRE-PREPARE message is similar to the one for verifying requests. If r has already accepted the PRE-PREPARE message, r discards the message pre-emptively. If r has already processed a different PRE-PREPARE message with $n' \neq n$ 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 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 PRE-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 the pre-prepare, and send a REPLY message to the client.

After receipt of $2f+1$ matching $\langle$COMMIT$,v,n,r'\rangle_{\vec{\mu}_{r'}}$ from distinct replicas r', replica r commits and executes the request before sending $\langle$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 [10], but we omit these details here for simplicity. They do not introduce any new issues for our restricted maxi-min analysis.

Resource Scheduling: Aardvark relies on the multiple physical network connections and incoming work queues to isolate traffic from individual replicas. This separation serves two purposes, it allows Aardvark replicas to ensure fairness in handling messages and it allows Aardvark replicas to silence other replicas that are abusing the network.

When there are incoming messages to be processed from multiple replicas, the receiving replica processes those messages in a round robin fashion. The fairness provided by round robin scheduling ensures that at most $n-2$ messages are processed between the time a message is received and when it is processed and prevents messages from a single faulty replica from drowning out messages from other replicas while facilitating the efficient collection of quorums of PREPARE and COMMIT messages.

The separation of network resources also allows replicas to pull the plug on a faulty replica that is overaggressive in network transmission by disabling interrupts on the network card connected to the faulty node. In our prototype, we disable a network connection when a replica's rate of message transmission in the current view is a factor of 20 more than any other replica. After dis-

connecting a replica for flooding, replicas reconnect the replica after 10 minutes, or when another replica is disconnected for flooding.

Maxi-min analysis: We now consider the Aardvark design under the restricted maxi-min design principle by considering the impact that single messages and exceptional message volume from a faulty primary or replica can have on the system.

Non-primary replicas send PREPARE and COMMIT messages to each other, but do not act on received PREPARE or COMMIT messages until there are a quorum of consistent messages. The use of quorums is fundamental for maintaining safety and has a significant beneficial impact on performance. Specifically, since replicas do not act until they have received messages from a quorum of other nodes, a single faulty replica is unable to cause another replica to perform actions it would not take anyway.

Since no individual message can cause undue harm to the system, a faulty replica's best chance to harm Aardvark is based on flooding other replicas with messages to impose additional MAC authentications and system interrupts. The round robin scheduling of incoming messages ensures that a faulty replica will be unable to actively prevent messages from other replicas from being delivered or processed. Incoming messages also impose costs in the form of system interrupts, the negative impact of system interrupts caused by a flooding replica are negated by turning the corresponding NIC off.

The primary has responsibilities that are not shared by other replicas. We discuss Aardvark's response to these additional challenges in Section 5.3.

Catchup messages. State catchup messages are not included in the description above. The state catchup messages are used to bring slow replicas back up to speed; the basic strategy is that a replica advertises its current state, and the other replicas respond with parts of their state that it may be missing. These messages offer chances for faulty replicas to impose significant load on other replicas since they do not require a quorum of messages to trigger work.

Aardvark extends its resource scheduling to explicitly deprioritize the handling of status messages as long as the system is making progress. The only time it is necessary to bring a slow replica up to speed is when the lack of responses prevents the system from making progress. Therefore, there are two scenarios under which Aardvark replicas process infrastructure messages: during view changes and when no other messages are available to process. As a result, processing catchup messages never slows the system.

5.3 View changes

Employing a primary to order requests enables batching [10, 16] and avoids the need to trust clients to obey a backoff protocol [1, 12]. 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 [5, 6] 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.

Hence, rather than designing specific mechanism to defend against each of these threats, past BFT systems [10, 20] 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 utilizes the same view change mechanism described in PBFT [10]; in conjunction with the agreement protocol this is sufficient to ensure eventual progress. Aardvark changes the conditions under which view changes are initiated. Aardvark augments the view changes required to ensure liveness with additional criteria based on the recent current system performance to limit a faulty primary's ability to adversely impact both the overall throughput and fairness.

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. In order to make this assessment, replicas in Aardvark expect two things from the primary: regular progress in the form of PRE-PREPARE messages and high throughput over every checkpoint interval.

Following the completion of a view change, each replica starts a hearbeat 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, Each time a view change is initiated due to the heartbeat timer, a replica doubles the heartbeat timer. Once the timer is reset because a PRE-PREPARE message is received, the replica resets the heartbeat timer back to its initial value. The heartbeat timer is application and environment specific, our implementation utilizes a hearbeat of 40ms. The regular PRE-PREPARE heartbeats ensure that replicas reach the next checkpoint interval promptly.

Replicas keep track of system throughput every checkpoint interval by recording the number of requests ex-

ecuted during that interval and measuring its duration. Replicas keep track of the peak throughput they observe during the previous n views. When a new view starts, replicas establish a required throughput value of 90% of the maximum peak during the previous n views.

At each checkpoint interval, the replica compares the observed throughput to the required throughput; if the observed throughput is less than the required throughput then the replica initiates a view change. After an initial grace period, 5s in our deployment, the replica begins increasing the required throughput by a factor of 0.01 every checkpoint interval. By tightening the screws on the primary, the primary is forced to provide consistently higher levels of service or be replaced. Since there is a limit to the throughput that is possible in the system — either through saturation or reaching the limit of client requests, the primary will eventually fail this check and 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.

Maxi-min analysis: Adaptive view changes limits the damage a faulty primary can impose. The adaptive view change and PRE-PREPARE heartbeats allow faulty primaries two fundamental options: they can provide substandard service and be replaced promptly or remain 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. In order to avoid being replaced due to missing the heartbeat timer, a faulty primary must make consistent process towards the next checkpoint interval. Once the checkpoint interval has been reached, a faulty primary is replaced unless it provides the required throughput, which is at least a fraction of the throughput a correct primary would provide. While the primary can remain just ahead of the required throughput, if it does so as long as possible it will provide the system with 95% of the throughput expected from a correct replica while it is primary.

5.3.2 Fairness

In addition to hurting overall system throughput, primaries are able to control which requests are processed. A faulty primary could consequently be unfair to a specific client (or set of clients) by neglecting to order requests from that client. In order to minimize the magnitude of unfairness in the system, replicas track fairness of request ordering. 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 $k+2c$ 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.

Maxi-min analysis: A faulty primary can delay processing a correct client's request from when the primary initially receives the client's request until the client's retransmission time fires and it sends its request to all replicas. Once $f+1$ replicas forward that client request, the primary must order it within $2c$ sequence numbers or a view change occurs.

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.

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 execution. To assess the loss in throughput incurred by Aardvark during uncivil executions, 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/sec. □

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. □

Theorem 2. *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 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 $\frac{t_{gracePeriod}}{t_{maxViewTimeout}+t_{gracePeriod}}\frac{n-f}{n}$ of that of Lemma 1, and, as α/θ tends to 0, the ratio between the throughput during uncivil and gracious intervals approaches $g\frac{t_{gracePeriod}}{t_{maxViewTimeout}+t_{gracePeriod}}\frac{(n-f)}{n}$. □

7 Evaluation

We evaluate the performance of Aardvark, PBFT, HQ, Q/U and Zyzzyva on a local Emulab cluster [33] located at UT Austin. This cluster consists of machines with dual 3GHz Intel Pentium 4 Xeon processors, 1GB of memory, and 1 Gb/s Ethernet connections.

The codebases used to report our results are provided by the respective systems' authors. James Cowling provided us the December 2007 public release of the PBFT codebase [7] as well as a copy of the HQ codebase. We used version 1.3 of the Q/U codebase, provided to us by Michael Abd-El-Malek in October 2008 [29]. The Zyzzyva codebase is the version used in the SOSP 2007 paper [20]. Whenever feasible, we rely on the existing pre-configurations for each system.

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.

7.1 Aardvark

Aardvark's peak performance is competitive with that of state of the art systems as shown in Figure 7.1. Aardvark's throughput tops out around 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

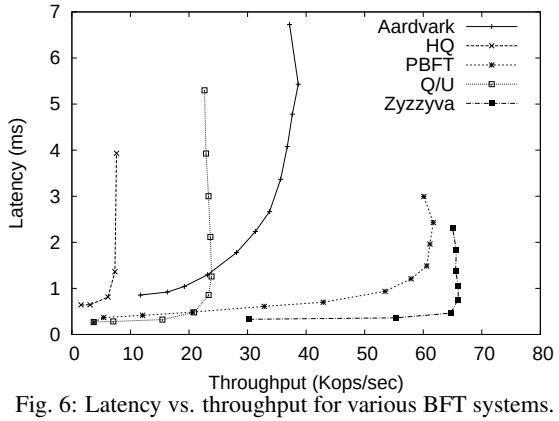

Fig. 6: Latency vs. throughput for various BFT systems.

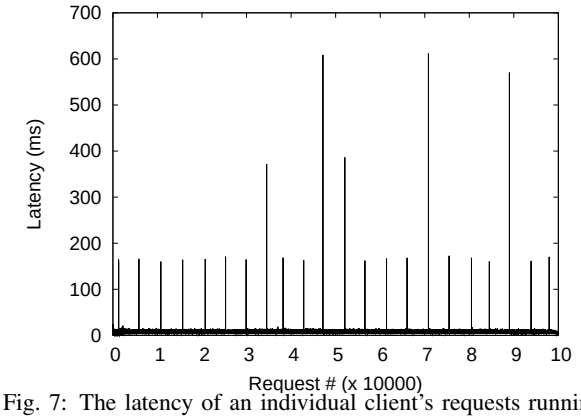

Fig. 7: The latency of an individual client's requests running Aardvark with 210 total clients. The sporadic jumps represent view changes in the protocol.

.

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. Most view changes complete in under 200ms; higher spikes indicate cases in which multiple view changes occurred successively. 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.

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. Figure 9 reports these peak throughputs.

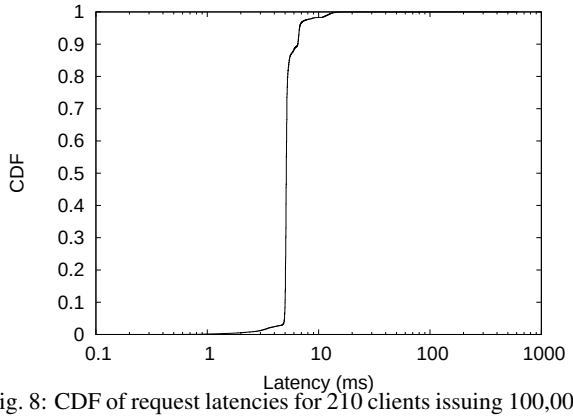

Fig. 8: CDF of request latencies for 210 clients issuing 100,000 requests with Aardvark servers.

System	Peak Performance
Aardvark	38667
PBFT	61710
PBFT w/ client signatures	31777
Aardvark w/o signatures	57405
Aardvark w/o adaptive throughput	39771

Fig. 9: Peak throughput of Aardvark and incremental versions of the Aardvark protocol

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 33% hit to system throughput. Splitting work across queues makes it easy for Aardvark to utilize the second processor in our test bed machines, which reduces the additional costs Aardvark pays for requiring signatures to authenticate client requests.

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 non-faulty, a view change requires the same amount of work as a single instance of consensus. Aardvark views 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 or 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: regular request dissemination and network flooding.

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

In PBFT and Zyzzvya, the clients send requests that are authenticated using MAC authenticators. The faulty client includes an authenticator on requests so that the primary will successfully verify the request, but the verification process will 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. The specified procedures for addressing this problem is a "view change like protocol" in PBFT and a conflict resolution procedure in Zyzzvya that blocks system 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 ([8] 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 systems drops to 0. In Zyzzyva this is because 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. In both PBFT and Zyzzyva, we see the potential for performance degradation due to protocol design. This potential is magnified in practice by the difficulty of implementing all complex corner cases correctly.

In HQ, our intended attack is to have clients send certificates during the WRITE-2 phase of the protocol with inconsistent MACs. The specified response is a signed WRITE-2-REFUSED message. We expect the signature generation to noticeably impact performance. Unfortunately, the HQ implementation is a prototype intended to be used to compare the normal case performance to that of PBFT, and the replica processing necessary to defend system safety against faulty MACs from clients is expressly not implemented.

QU 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 operations, 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 conflicting requests to the replicas. In this setting with only 20 clients, QU provides 0 throughput. QU's focus on performance in the absence of 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.

System	Peak Performance	Network Flooding
PBFT	61710	crash
QU	23850	21197
HQ	7629	0
Zyzzyva	65999	crash
Aardvark	38667	7873

Fig. 10: Observed peak throughput of BFT systems in the fault free case and under heavy client retransmission load.

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 and sending requests with invalid MACs. We implement both attacks and find the results to be comparable. We report the results for requests with invalid MACs. Our focus on the restricted maxi-min design pattern limits our vulnerability to faulty clients.

Network flooding. In Figure 10 we demonstrate the impact of a single faulty client that floods the replicas with messages. In PBFT, Zyzzyva, Aardvark, and QU we instrument a client to repeatedly send 9k byte messages to the replicas. In HQ a client repeatedly requests TCP connections.

HQ, PBFT, and Zyzzyva suffer dramatic performance degradation as their incoming network resources are consumed by the flooding client. In the case of HQ every incoming TCP connection on the replica is allocated to the spamming client. PBFT and Zyzzyva both encounter problems as the incoming client requests drown out the replica communication necessary for the systems to make progress.

QU performs well under the client flooding attack largely because we were unable to implement a mechanism to interfere with the RPC library they use for communication in the available time. We do, however, show a minimal degradation in performance due to increased consumption of network bandwidth.

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

Fig. 11: Throughput during intervals in which the primary delays sending PRE-PREPARE message (or equivalent) by 1, 10, and 100 ms.

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

Fig. 12: Average throughput for a starved client that is shunned by a faulty primary versus the average per-client throughput for any other client.

7.2.2 Faulty Primaries

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

In Figure 11 we show the impact on PBFT, Zyzzyva, and Aardvark of primaries that delay sending PRE-PREPARE messages by 1, 10, and 100 ms respectively. The throughput of both PBFT and Zyzzyva degrades dramatically as the slow primary is not slow enough to trigger their view change conditions. 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. Figure 12 explores the impact that an unfair primary can have on the throughput for a targeted node. In 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 was selected because it keeps 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. Aardvark's fairness detection and periodic view changes limit the impact of the unfair primary. For Zyzzyva, the unfair primary ignores messages from the targeted client entirely. The resulting throughput is 0 because replicas in the Zyzzyva implementation do not forward received requests to the primary.

System	Peak Throughput	Replica Flood
HQ	7629	0
PBFT	61710	251
QU	23850	21197
Zyzzyva	65999	0
Aardvark	38667	11706

Fig. 13: Observed peak throughput and observed throughput when one replica floods the network with 9k byte messages.

7.2.3 Non-Primary Replicas

We implement a faulty replica that blasts network traffic at the other replicas and show the results in Figure 13. The HQ attacker again repeatedly opens TCP connections, consuming all of the incoming connections on the other replicas. In the other four systems, the attacker blasts 9KB messages at the other replicas. PBFT and Zyzzyva 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 of both requests from the clients and also replica messages required to make progress. Aardvark's utilization of separate worker queues ensures that the replicas receive the messages necessary to make progress, though the throughput is lower than expected.

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. [31] show that PBFT [10], Q/U [1], HQ [13], and Zyzzyva [20] are all sensitive to network performance. They provide a thorough examination of the gracious executions of the four canonical systems through a ns2 [27] 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. [5] and Amir et al. [6] 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.

PBFT [10], Q/U [1], HQ [13], and Zyzzyva [20] are recent BFT replication protocols. These systems focus on optimizing performance during gracious executions and collectively demonstrate that BFT replication sys-

tems 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. [18] 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. [26] 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. [14] explore how quickly consensus can be achieved under eventual synchrony.

9 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, for significant improvement in performance during uncivil executions.

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*, 2006.

[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*, 2002.

[4] A. S. Aiyer, L. Alvisi, R. A. Bazzi, and A. Clement. Matrix signatures: From macs to digital signatures in distributed systems. In *DISC*, 2008.

[5] 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.

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

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

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

[9] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. 3rd OSDI*, pages 173–186, Feb. 1999.

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

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

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

[13] 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.

[14] P. Dutta, R. Guerraoui, and L. Lamport. How fast can eventual synchrony lead to consensus? In *DSN*, 2005.

[15] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, 1985.

[16] R. Friedman and R. V. Renesse. Packing messages as a tool for boosting the performance of total ordering protocls. In *HPDC '97*, 1997.

[17] R. Guerraoui, V. Quéma, and M. Vukolic. The next 700 bft protocols. Technical report, Infoscience — Ecole Polytechnique Federale de Lausanne (Switzerland), 2008.

[18] J. Hendricks, G. R. Ganger, and M. K. Reiter. Low-overhead byzantine fault-tolerant storage. In *SOSP*, 2007.

[19] I. Keidar and A. Shraer. Timeliness, failure-detectors, and consensus performance. In *PODC*, 2006.

[20] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative byzantine fault tolerance. In *SOSP*, 2007.

[21] R. Kotla and M. Dahlin. High throughput Byzantine fault tolerance. In *DSN*, June 2004.

[22] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

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

[24] B. W. Lampson. Hints for computer system design. *SIGOPS Oper. Syst. Rev.*, 17(5):33–48, 1983.

[25] A. Mahimkar, J. Dange, V. Shmatikov, H. Vin, and Y. Zhang. dfence: Transparent network-based denial of service mitigation. In *NSDI*, 2007.

[26] D. Malkhi, F. Oprea, and L. Zhou. Omega meets paxos: Leader election and stability without eventual timely links. In *DISC*, 2005.

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

[28] M. Osborne and A. Rubinstein. *A Course in Game Theory*. MIT Press, 1994.

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

[30] M. Serafini, P. Bokor, and N. Suri. Scrooge: Stable speculative byzantine fault tolerance using testifiers. Technical report, Darmstadt University of Technology, Department of Computer Science, September 2008.

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

[32] M. Walfish, M. Vutukuru, H. Balakrishnan, D. Karger, and S. Shenker. Ddos defense by offense. In *SIGCOMM '06*, 2006.

[33] 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*, 2002.

[34] T. Wood, R. Singh, A. Venkataramani, and P. Shenoy. Zz: Cheap practical bft using virtualization. Technical report, University of Massachussets, 2008.

[35] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proc. 19th SOSP*, 2003.