Machine Fault Tolerance for Reliable Datacenter Systems

Danyang Zhuo*, Qiao Zhang, Dan R. K. Ports, Arvind Krishnamurthy and Thomas Anderson
University of Washington
{danyangz, qiao, drkp, arvind, tom}@cs.washington.edu

ABSTRACT

Although rare in absolute terms, undetected CPU, memory, and disk errors occur often enough at datacenter scale to significantly affect overall system reliability and availability. In this paper, we propose a new failure model, called Machine Fault Tolerance, and a new abstraction, a replicated write-once trusted table, to provide improved resilience to these types of failures. Since most machine failures manifest in application server and operating system code, we assume a Byzantine model for those parts of the system. However, by assuming that the hypervisor and network are trustworthy, we are able to reduce the overhead of machine-fault masking to be close to that of non-Byzantine Paxos.

1. INTRODUCTION

Machine failures are widespread in datacenter environments. Today's largest datacenters are estimated to have up to hundreds of thousands of servers. At this massive scale, hardware faults are an everyday occurrence, ranging from failures of entire machines to DRAM bit errors and undetected disk corruption.

The challenge for system designers is to mask as many of these failures as possible. Replication protocols like Paxos [12] are a mainstay of today's distributed systems, making it possible to build reliable services in the presence of crash failures. However, standard replication protocols are not robust to more complex failures such as those caused by memory errors or non-deterministic software bugs. These types of failures are increasingly a problem for datacenter applications. For example, memory corruptions and software bugs in Google's Paxos library have led to inconsistencies in the Chubby lock database [5], and a single-bit corruption caused Amazon S3 to become unavailable for hours in 2008 [1].

Byzantine fault tolerant (BFT) replication protocols offer an option for these types of failures, ensuring that the system operates correctly even when a fraction of the nodes misbehave in arbitrary or even malicious ways. However, these protocols present significant barriers to adoption. First, they are complex, leading to the question of whether bugs in their implementation will themselves reduce system reliability. Moreover, despite much recent work to reduce the cost of BFT protocols [3, 4, 9, 11], they remain prohibitively expensive. They require higher degrees of replication ($3f+1$ replicas instead of $2f+1$ to tolerate f failures), greater message complexity (sometimes involving all-to-all communication), and expensive cryptographic operations (taking more than 1 μs per message).

As an alternative, we aim to build a set of *machine-fault tolerant* (MFT) protocols that tolerate a class of hardware machine errors. The fault model allows application code to corrupt data or send incorrect responses. Unlike BFT, we assume parts of the infrastructure – namely, a hypervisor running on each machine and the network fabric – are trusted to behave correctly. This model of error is one that allows for an efficient implementation: we are able to provide MFT state machine replication with $2f+1$ replicas and no cryptography, using a protocol whose efficiency approaches that of Paxos.

In this paper, we present a protocol for providing efficient MFT replication. The basis for our protocol is a new primitive implemented in the hypervisor called the *replicated write-once trusted table*, an API that allows applications to register messages with sequence numbers and remotely retrieve messages by their corresponding sequence numbers. Because our model trusts not only the hypervisor but also the communication between hypervisors in the data center, the replicated write-once trusted table can be implemented without the use of cryptography. We show how to leverage this primitive to build a protocol, H-MFT, that provides efficient MFT replication using only $2f+1$ replicas.

2. BACKGROUND

Machine failures beyond simple crash failures, e.g. memory errors and disk corruptions, are increasingly significant in datacenters. Measurements of DRAM errors in Google datacenters [17] show that around 1.29% of their servers experience at least one memory error per year of operation that is not masked by error-correcting codes. Such memory errors can and have resulted in catastrophic consequences [1].

Today's state of the art approach in industry for fault-tolerance is Paxos replication [12], an efficient technique that can mask f crash failures with $2f+1$ replicas. While Paxos has proven effective for tolerating crash failures, it remains vulnerable to other types of machine failures, such

*The first two authors contributed equally to this work.

APSys '14, June 25-26, 2014, Beijing, China

http://dx.doi.org/10.1145/2637166.2637235

as memory errors.

The other well-known replication technique, Byzantine fault tolerance (BFT), can handle any type of non-fail-stop errors, but induces considerable overhead. Traditional approaches to BFT require at least $3f+1$ replicas [16], $O(n^2)$ communication cost, and extensive use of cryptography. Moreover, the complexity of BFT protocols is another barrier for its adoption because deployment of BFT protocols would require significant expertise and administrative effort.

The addition of a trusted component changes the landscape of BFT protocols, making it possible to break the $3f+1$ replica lower bound. For example, A2M [7] uses an attested append-only log, and TrInc [14] uses an attested monotonic counter; both can reduce the number of replicas required to $2f+1$. MinBFT and MinZyzzyva [18] use a similar trusted component to reduce the number of protocol steps. The key theoretical insight is that preventing an adversary from equivocating (sending conflicting messages to multiple replicas) averts many of the most insidious attacks.

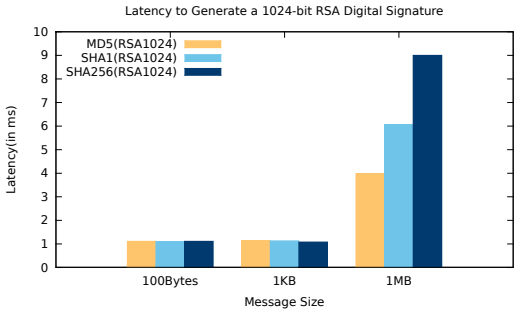
However, preventing equivocation is not sufficient by itself. *Transferable authentication* is also required: each node must be able not only to authenticate a message from another node, but also to forward the message to a third node that will also be able to authenticate it. This requirement was recently formally proven by Clement et al. [8]. What this means in practice for this class of protocols is that their trusted components require expensive digital signatures for attestation (the MAC-based authenticators pioneered in PBFT [4] are not sufficient).

Our work takes a different approach. By relying on the trusted components at each node to communicate with each other over a trusted network fabric, we are able to provide non-equivocation and transferable authentication *without* cryptography. This property is important in the datacenter environment, because, as we see below, cryptographic costs can dwarf network latency as a protocol cost.

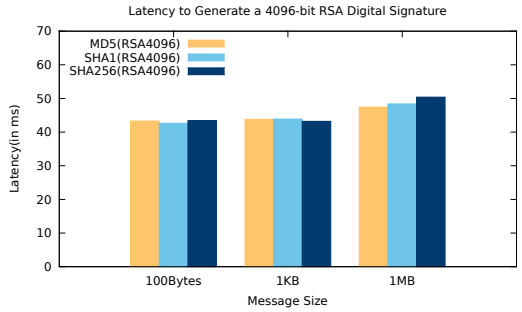
2.1 The Costs of Cryptography

To understand how much cryptographic costs contribute to the protocol, we measure the cost of digital signatures for messages in Fig. 1 and compare to datacenter network latency. Cryptographic costs are measured using the standard Java cryptography library on a Intel i7-2600 3.4GHz 8-core system running Linux 3.8.

The computational cost of digital signatures is substantial, particularly for the small message sizes commonly used in BFT protocols. In particular, it takes at least 1 ms (for 1024-bit RSA) or 40 ms (for 4096-bit RSA) to generate RSA signatures. Secondly, more secure signatures take longer time to generate as the message size grows due the additional computational complexity in more secure hash function. RSA-SHA256 takes 50% (for 1024-bit RSA) more time to generate than RSA-SHA1 when signing a $1MB$ message. More importantly, as we use longer key size, the computational cost of generating signatures is significantly more than the communication cost. For example, when we move from 1024-bit RSA (in Fig. 1a) to 4096-bit RSA (in Fig. 1b), the cost of generating signatures increases 40 times for small control messages.



(a) 1024-bit RSA Digital Signature



(b) 4096-bit RSA Digital Signature

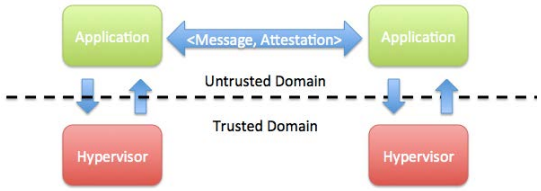
Figure 1: Measurement of cryptography costs

All of these numbers dwarf the network fabric latency in a modern datacenter network, which can be as low as 6 μs [10] – 3–4 orders of magnitude less than a digital signature. What this means is that the latency of BFT protocols in the datacenter is dominated not by the latency of communication but the cost of constructing digital signatures. This disparity seems only likely to increase as advances in cryptography continually necessitate more complex signature algorithms, while trends in network and OS design aim to provide support for ultra-low-latency communication.

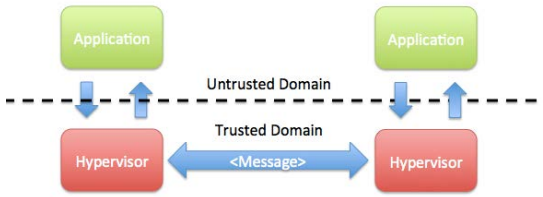
Some protocols (e.g., MinBFT [18]) assume that all nodes have a globally shared secret key that enables a MAC to be used for transferable authentication. Although realizing this assumption introduces problematic key management challenges, it enables far faster cryptographic operations. Even so, it is not a panacea: the cost of computing a SHA256 HMAC remains on the order of 100μs, considerably higher than the network latency, meaning that cryptography costs are still a major component of the protocol cost.

3. MFT MODEL

We present a new fault tolerance model, Machine Fault Tolerance (MFT), that is slightly weaker than BFT yet captures most machine failures including those beyond simple crash failures. The MFT model assumes that the cloud infras-



(a) BFT Model: A2M/TrInc/MinBFT



(b) MFT Model

Figure 2: In (a), a sender proves to a recipient that it does not equivocate on a message by attaching an attestation generated by its trusted hypervisor. In (b), the message is registered in the sender's hypervisor and replicated to the recipient's hypervisor through a trusted channel, and therefore the message is automatically authenticated.

tructure, i.e. a hypervisor running on each machine and the network connecting them, will not fail arbitrarily.

More precisely, our model assumes that the system consists of a set of nodes that run both application code and a trusted hypervisor. The application code may fail arbitrarily; the trusted component implemented in the hypervisor is assumed to fail only by stopping (if the machine crashes). We justify this assumption by observing that hypervisors can have a smaller code base than the full software stack running on a given machine (for example, TrustVisor [15] is implemented in approximately 6000 lines of code, whereas the Linux kernel alone exceeds 15 million lines [2]). Moreover, because hypervisor code is run infrequently compared to the OS and applications and has a smaller memory footprint, it is less prone to machine faults such as memory errors.

We assume an asynchronous network connects the hypervisors, but trust it not to modify or spoof packets. Packets can still be dropped, reordered and duplicated in transmission. We make the standard assumption (required for liveness) that packets between two correct nodes that are repeatedly retransmitted will eventually be delivered. Trusting the network not to modify packets is in some sense unconventional: a traditional view of wide-area fault tolerance would make no assumptions about the behavior of the Internet links connecting nodes, which could potentially be operated by a malicious party. However, we are targeting a datacenter environment, where the network is a trusted infrastructure component. Although the network may suffer outages, the network components (switches, etc) run a less complex software stack and are therefore less likely to suffer complex, arbitrary failures.

The result of this model is that MFT protocols can be implemented more efficiently than BFT protocols. In BFT systems that use trusted components, e.g. A2M and TrInc, applications authenticate a message by generating an attestation from the trusted component and sending both the message and its attestation to the recipient in the untrusted domain, shown in Fig. 2a. Because the data path passes through the untrusted domain, i.e. the application and the network, the attestation must be digitally signed by the trusted component to prevent from being tampered with. In our model, protocol messages are directly sent between hypervisors, and therefore do not cross into untrusted domain, as shown in Fig. 2b. The recipient application can be certain that the messages are authenticated even without the protection of digital signatures. Our model thus enables a much more efficient implementation without using cryptography.

4. REPLICATED WRITE-ONCE TRUSTED TABLE

The fundamental primitive underlying our protocol is the *replicated write-once trusted table*, a simple abstraction that provides equivocation-free communication between user applications. It implements transferable authentication using the trusted network rather than using cryptography.

A replicated write-once trusted table is owned by one of the replicas (the *table owner*), and replicated to a set of other replicas (the *table clients).* The table owner can write messages into the table, but cannot subsequently modify them. Table clients are notified of new messages in the table, and can retrieve them by id. Because the messages are immutable and replicated, a pointer comprising the table id and message id can be used to forward a message to another client while providing authentication.

The replicated write-once trusted table is designed to be a simple API that can be implemented easily in a hypervisor but can be applied to many applications; Sec. 5 shows how to apply it to MFT replication. In this respect, it is similar to the trusted log from A2M [7], with a novel modification to the retrieval mechanism.

4.1 API

The full API of *replicated write-once trusted table* is shown in Figure 3. We describe in detail the semantics of our API below.

Table Creation. To disseminate messages reliably to other replicas, an application invokes `create_table(clientID_list)` to create and own a trusted table. The application obtains a data-center unique table id, which is a tuple consisting of the IP address of the hypervisor and a table id locally unique to the hypervisor.

Message Registration. Having created a trusted table, the application registers messages by invoking `put(T, UI, msg)`.

- `create_table(clientID_list)` → `table_id`
  Creates a new table with the specified list of clients, and returns a unique table id.

- `put(T, UI, message)`
  Writes the specified message into the table with id UI, assuming that T is a valid table ID and UI is available for write. If UI maps to an existing message or has been garbage collected, put instead returns UNAVAILABLE.

(a) API of Table Owner

- `get(T, UI)` → `message`
  Returns the message corresponding to id UI in table T if it is available. Otherwise, returns GARBAGE-COLLECTED if the message has been garbage collected or NOT-WRITTEN-YET if no message with id UI has yet been written.

(b) API of Table Client

- `truncate(T, UI)`
  Verifies that T is a valid table ID. Upon receiving $(size\_of\_clientID\_list + 1)/2$ matching truncate requests, garbage collect all table entries with sequence number smaller than UI.

(c) API common to Table Clients and Table Owner

Figure 3: Replicated Write-Once Trusted Table API

In order to maintain write-once semantics, the hypervisor first verifies that the sequence number does not already map to a message or has been garbage collected. If the sequence number is available for write, the message is stored in the table corresponding to the given sequence number.

Message Retrieval. Rather than providing attestations that untrusted clients can exchange to verify that the messages are stored in the table (as in A2M and TrInc), recipients retrieve messages from the table client in their hypervisor by invoking `get(T, UI)`. The advantage of this retrieval mechanism is that messages are pulled from a trusted table of the message owner to the intended application via only trusted components, i.e. the hypervisors and the trusted communication channel between the hypervisors. As a result, messages arrive at the intended application authenticated without the protection of digital signatures.

Garbage Collection. Without garbage collection of table contents, hypervisor memory usage might grow without bound. At the same time, a table owner cannot unilaterally garbage-collect table contents because this might allow it to remove a message needed by a remote host. Accordingly, $f+1$ `truncate(T, UI)` calls must be made from different hypervisors before the table owner can safely remove messages with ID lower than UI.

4.2 Table Replication

The contents of a table must be replicated, because other replicas may need to refer to a message even if the table owner has crashed. To make this possible, table replicas are created on the hypervisors of all other clients. When the table owner adds a new message to the table, the message is broadcast to the other replicas, which store it in their replica of the table. If a table client calls `get` for a message not available in the local replica, the hypervisor contacts the table owner; if the table owner is non-responsive, it requests the message from any other replica that might have it available.

5. H-MFT

In this section, we describe a Hypervisor-MFT protocol that leverages our table API to prevent server equivocation. Every H-MFT protocol message needs to be stored in the *replicated write-once trusted table* so that the other replicas in the system can see them in the same order. Each replica has one table that stores every protocol message it sends. We replace the attestation mechanism in MinBFT [18] with our replicated table retrieval mechanism, and also modify its garbage collection subprotocol to work with our table API. Despite the similarity with MinBFT, our protocol cannot tolerate Byzantine behaviors in the network and hypervisors. However, H-MFT does work correctly in handling other kinds of non-crash failures, just as MinBFT.

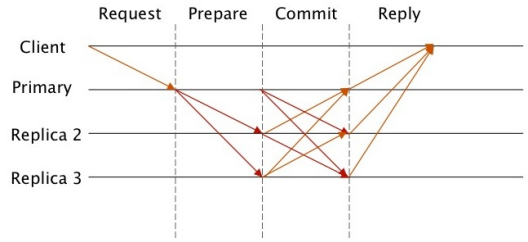


Figure 4: H-MFT common case

5.1 Design

Client. A client sends its request $\langle \text{REQUEST}, o, t, c \rangle_{\sigma_c}$ to the primary and waits for replies, where o is the request opcode, t is the client request identifier (used to ensure that a request is not executed more than once), c is the clientID, and the request is signed by the client's private key σ_c. Each reply $\langle \text{REPLY}, t, v, r \rangle$ received by the client has the client request

identifier t, current view number v, and the application reply r. The client needs to wait for $f+1$ matching replies to be sure that the reply is valid and the request is persisted in the system, because at least one of the replies comes from the correct replica, and since the correct replica's hypervisor has the latest client request, the request will be eventually propagated to all the other correct replicas. As is common practice in today's Internet, the client request is signed at the client side. We therefore do not take into account the latency from generating digital signatures for client requests in our evaluation.

Replica. Each replica maintains a current sequence number, UI, for its own table and the last seen sequence numbers for all the other replicas. In order to send a message, replica s_i advances UI_i by one and puts the message with the new UI_i. In order to receive a message from another replica s_j, replica s_i always uses the next expected sequence number for s_j to retrieve the message.

Upon receiving a client request, the primary replica s_p puts a tuple $(\text{UI}_p, \langle \text{PREPARE}, v, s_p, m \rangle)$ into its table, where UI_p is the current sequence number for primary replica s_p, v is the current view number and m is the original client request. All the other replicas fetch the client request from the primary's table. Upon receiving a PREPARE message, replica s_i verifies the client digital signature to ensure that the request is an authenticated request from the client and checks the client request identifier to ensure that the request does not get executed twice. If the request is valid, replica s_i puts $(\text{UI}_i, \langle \text{COMMIT}, v, s_i, s_p, \text{UI}_p, m \rangle)$ into its table for sequence number UI_i. Replica s_i then waits for $f+1$ matching COMMIT message from the other replicas' tables before executing the client request and sending a reply $\langle \text{REPLY}, t, v, r \rangle$ to the client.

Garbage Collection. We periodically garbage collect the tables. For k consecutive client requests, replica s_i checkpoints its application state and puts $(\text{UI}_i, \langle \text{CHECKPOINT}, s_i, \text{UI}_{latest}, d \rangle)$ where UI_{latest} is the sequence number for the last executed request and d is the digest of the checkpointed state. Upon receiving $f+1$ matching CHECKPOINT messages with identical UI_{latest} and d, replica s_i considers the checkpoint stable. We call the set of references, (s_j, UI_j), to the $f+1$ matching CHECKPOINT table entries a *checkpoint certificate*. Replica s_i then checks if its own checkpoint state matches with that in the checkpoint certificate. If the replica finds its own checkpoint state stale, it initiates a state transfer from a correct replica using the same protocol of PBFT [4]. Otherwise, it helps the other replicas to garbage collect their tables by calling truncate request on all replicas with the appropriate sequence number UI.

View Change. When replica s_i suspects the primary to be faulty, it puts $(\text{UI}_i, \langle \text{REQ\_VIEW\_CHANGE}, s_i, v, v' \rangle)$ where v is the current view number and $v' = v+1$ is the next view number. Upon receiving $f+1$ matching REQ_VIEW_CHANGE requests, replica s_i updates its view to v', and puts $(\text{UI}_i, \langle \text{VIEW\_CHANGE}, s_i, v', C_{latest}, O \rangle)$ where C_{latest} is the latest stable checkpoint certificate, O is the set of all messages sent by the replica s_i since the last checkpoint. Replica s_i then stops responding to messages from the old view v. Upon receiving $f+1$ matching VIEW_CHANGE requests, replica s_i now has a *new view certificate*.

The new primary $s_{p'}$ writes $(\text{UI}_{p'}, \langle \text{NEW\_VIEW}, s_{p'}, v', V_{VC}, S, \text{new UI}_{p'} \rangle)$,where V_{VC} is the *new view certificate*, S is the set of requests accepted since the last checkpoint and new $\text{UI}_{p'}$ is the next sequence number new primary i will write to in the view v'. Once a replica receives the NEW_VIEW message, it checks the *new view certificate* and whether S can be computed from the certificate. If the NEW_VIEW request is valid, it waits every request in S to be executed before accepting requests for v'.

5.2 Discussion

Without garbage collection, H-MFT is a simple translation of MinBFT [18] using our table API. For any message protected by digital signature in MinBFT, we send and retrieve the message through the trusted hypervisor and network. For any certificate composed of $f+1$ digital signatures used in MinBFT, we use $f+1$ references to the replicated tables for authentication. However, a possible issue from using our replicated tables is that references in certificates can be invalid if the original messages referenced are garbage collected. Therefore, to prove the correctness of H-MFT, we only need to show that our garbage collection subprotocol does not violate the correctness of MinBFT.

Whenever a new checkpoint c is stable, we want to safely garbage collect all messages before the previous stable checkpoint $c-1$. We need to ensure that all existing certificates do not reference messages before checkpoint $c-1$. If replica s_j has a certificate with an invalid reference to a message before $c-1$, at least $f+1$ replicas believe that checkpoint c is stable for the truncation to have happened for $c-1$. Since the latest stable checkpoint cannot be garbage collected (f faulty replica are not enough to remove the latest checkpoint certificate), s_j is guaranteed to be able to state transfer to the latest stable checkpoint.

6. EVALUATION

Our design uses the replicated write-once trusted table to provide MFT replication without many of the most expensive components of BFT protocols. In Table 1, we compare the complexity of H-MFT with several existing protocols: PBFT [4], Zyzzyva [11], A2M-PBFT-EA [7], MinBFT [18], and Paxos [12] as a baseline for comparison.

Like other proposals that use a trusted component (A2M-PBFT-EA and MinBFT), H-MFT reduces the replication overhead to $2f+1$ replicas. However, it does so without the use of digital signatures and message authentication codes that we have shown in Section 2.1 represents a significant cost for datacenters. H-MFT provides replication in only four message delays, the optimal achievable for a non-speculative protocol. We anticipate that building a speculative protocol

	PBFT-MAC [4]	Zyzzyva [11]	A2M-PBFT-EA [7]	MinBFT [18]	*H-MFT*	Paxos [12]
replication factor	3f+1	3f+1	**2f+1**	**2f+1**	**2f+1**	**2f+1**
number of message rounds	5	**3**	5	**4**	**4**	**3 or 4**
number of digital signatures	**0**	**0**	4	**0**	**0**	**0**
number of HMAC	2	2	0	2	**0**	**0**
number of messages	O(n^2)	**O(n)**	O(n^2)	O(n^2)	O(n^2)	**O(n)**

Table 1: Performance Comparison of BFT, MFT protocols and Paxos

akin to Zyzzyva using the replicated write-once trusted table abstraction could allow us to further improve on H-MFT's performance, eliminating the $O(n^2)$ message cost.

7. RELATED WORK

Machine Fault Tolerance can be provided by BFT protocols, but at an excessive cost. Traditional BFT protocols require at least $3f+1$ replicas [13]. Prior work [3, 4, 9, 11] has reduced the communication cost of BFT protocols, but cannot reduce the replication cost.

Our work is inspired by recent work that reduces the replication cost to $2f+1$ using a trusted component, including A2M [7], TrInc [14] and MinBFT [18]. Each makes use of a trusted component on each end host machine to attest and order messages sent by the sender to prevent equivocation. A2M uses an attested append-only memory as a trusted log for each BFT protocol message. TrInc and MinBFT reduce the space requirement on the trusted component by using a simple monotonic counter instead. However, neither A2M nor TrInc can avoid using digital signatures, because the messages are exchanged via an untrusted network and require cryptographic attestations to prevent tampering. MinBFT replaces digital signatures with HMACs, reducing the cost of cryptography (though it still exceeds datacenter network latency). However, this requires establishing a globally shared symmetric key.

Clement *et al.* [8] formally prove that both non-equivocation and transferable authentication are required to reduce BFT replication cost to $2f+1$. Transferable authentication is necessary because in a $2f+1$ replica group, in the worst case, only one correct replica has the correct protocol message. The correct replica has to prove to the other replicas that the message is indeed authenticated. Transferable authentication is traditionally provided using digital signatures; our replicated write-once trusted table instead leverages a trusted network to provide transferable authentication without cryptography.

Chun *et al.* [6] propose using a single hypervisor as a trusted base to provide BFT for virtual machines *on the same physical machine*. However, the single hypervisor becomes the single point of failure in the system.

A complementary approach to reducing replication cost is to separate execution from agreement [19, 20]. This approach divides BFT protocols into an agreement phase ordering incoming requests, and an execution phase where the requests are executed by application code. While the agreement phase still requires $3f+1$ replicas, only $2f+1$ replicas need to participate in the execution phase. ZZ [19] takes this approach further, using only $f+1$ execution replicas in the fault-free case, and activating additional replicas when there are failures. This approach is complementary to ours; we anticipate that using a hybrid approach, we can construct a MFT protocol that requires only $f+1$ execution replicas and $2f+1$ agreement replicas.

8. CONCLUSION

The authenticated communication channel between hypervisors in datacenter network provides a new opportunity to rethink the fault tolerance model. We argue that machine faults cover most of the non-crash failures in datacenters, and can be handled efficiently by a new class of MFT protocols. Using our new replicated write-once trusted table abstraction, we can implement an efficient protocol, H-MFT, for building more robust distributed systems.

Acknowledgments

We thank the members of the UW systems and networking research groups for their insightful feedback and lively discussions. This work was supported by the National Science Foundation under awards #0963754 and #1016477, and by the Wilma Bradley and Hacherl Endowed Fellowships.

9. REFERENCES

[1] Amazon S3 availability event: July 20, 2008. `http://status.aws.amazon.com/s3-20080720.html`.

[2] Linux kernel in 2011: 15 million total lines of code and Microsoft is a top contributor. `http://arstechnica.com/business/2012/04/linux-kernel-in-2011-15-million-total-lines-of-code-and-microsoft-is-a-top-contributor/`.

[3] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *SOSP*, 2005.

[4] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *OSDI*, 1999.

[5] T. Chandra, R. Griesemer, and J. Redstone. Paxos made live - an engineering perspective. In *PODC*, 2007.

[6] B.-G. Chun, P. Maniatis, and S. Shenker. Diverse replication for single-machine Byzantine-fault tolerance. In *USENIX ATC*, 2008.

[7] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *SOSP*, 2007.

[8] A. Clement, F. Junqueira, A. Kate, and R. Rodrigues. On the (limited) power of non-equivocation. In *PODC*, 2012.

[9] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. Hq replication: A hybrid quorum protocol for Byzantine fault tolerance. In *OSDI*, 2006.

[10] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable low latency for data center applications. In *SOCC '12*, San Jose, CA, USA, Oct. 2012.

[11] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *SOSP*, 2007.

[12] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 1998.

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

[14] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *NSDI*, 2009.

[15] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *IEEE Symposium on Security and Privacy*, 2010.

[16] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, Apr. 1980.

[17] B. Schroeder, E. Pinheiro, and W.-D. Weber. Dram errors in the wild: A large-scale field study. In *SIGMETRICS*, 2009.

[18] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo. Efficient Byzantine fault-tolerance. *IEEE Transactions on Computers*, 2013.

[19] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet. ZZ and the art of practical BFT execution. In *EuroSys*, 2011.

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