Visigoth Fault Tolerance

Daniel Porto[†], João Leitão[†], Cheng Li[‡], Allen Clement[‡ *], Aniket Kate[♮],
Flavio Junqueira[♯] and Rodrigo Rodrigues[†]

[†]NOVA Univ. Lisbon / NOVA LINCS, [‡]MPI-SWS,[♮]MMCI/Saarland University, [♯]Microsoft Research

Abstract

We present a new technique for designing distributed protocols for building reliable stateful services called Visigoth Fault Tolerance (VFT). VFT introduces the Visigoth model, which makes it possible to calibrate the timing assumptions of a system using a threshold of slow processes or messages, and also to distinguish between non-malicious arbitrary faults and correlated attack scenarios. This enables solutions that leverage the characteristics of data center systems, namely their secure environment and predictable performance, in order to allow replicated systems to be more efficient with respect to the utilization of resources than those designed under asynchrony and Byzantine assumptions, while avoiding the need to make a system synchronous, or to restrict failure modes to silent crashes. We implemented a VFT protocol for a state machine replication library, and ran several benchmarks. Our evaluation shows that VFT has comparable performance to existing schemes and brings significant benefits in terms of the throughput per dollar, i.e., the server cost for sustaining a certain level of request execution.

1. Introduction

Techniques have been proposed over the past few years to make the performance of both data center networks [48, 52] and data center systems [32] more predictable. Predictability is important because systems in data centers often comprise and depend on a number of networked servers and operations require a subset of those servers to be contacted and to exchange messages. Without predictable performance, the quality of the provided service might fall short of the demanding requirements of users of online services [22], and even of offline services such as batch processing [51].

However, despite this trend of increasing predictability in performance within the data center, the design of replication protocols for stateful services that run inside data centers is still making the same pessimistic assumptions regarding timeliness that are commonly used for unpredictable environments like the Internet. For example, systems like Chubby [9], Spanner [14], Megastore [7], or ZooKeeper [25] use at its core the Paxos [30] consensus algorithm and variants [27], which assume an asynchronous system, where all messages and processing events can be arbitrarily slow.

A similar argument to the one made above regarding the pessimistic assumptions on timeliness can also be made regarding non-crash faults. There is increasing evidence that machines and networks fail in unexpected ways that are not captured by the crash fault model, particularly at the scale of a data center, where the unlikely becomes commonplace [3–5, 26]. While this is addressed by Byzantine Fault Tolerance (BFT) techniques, BFT is unnecessarily conservative for data center environments. This is because BFT is designed to cope with coordinated malice, which is unlikely to happen within the security perimeter of the data center. (In fact, this excess of pessimism has been pointed out as one of the obstacles for the adoption of BFT in data center environments [45].)

In this paper, we take the position that it is possible to take advantage of the fact that data centers are more predictable and controllable than an open Internet environment, in order to make stateful services more resource efficient. Furthermore, this can be achieved without having to make assumptions that might be difficult to meet in practice, such as assuming a fully synchronous system where all machines and all messages meet tight deadlines, or that data corruption never occurs. By resource efficient, we mean cutting the replication factors of systems like Paxos, which is an important goal since ultimately this can lead to savings in both infrastructure and energy costs, which represent the vast majority of the costs for operating a data center [24].

To demonstrate this, we present a new technique for designing distributed protocols for reliable stateful services called Visigoth Fault Tolerance (VFT). VFT introduces the

[*] currently working at Google.

EUROSYS '15, April 21-24, 2015, Bordeaux, France.

http://dx.doi.org/10.1145/10.1145/2741948.2741979

Visigoth model, which provides two important knobs that can be tuned independently. The first is between an asynchronous system where delays can be arbitrarily large and a synchronous system where there is a bound on message transmission and processing delays. More precisely, our model assumes that, among the machines that communicate in a distributed protocol, there is a subset that can communicate in a timely manner, and only a limited number of machines are perceived as arbitrarily slow, due to either message transmission or processing delays. Importantly, in our model, when some machines cannot be contacted, we do not require the ability to pinpoint which ones have crashed and which ones are just slow. The second knob is between the crash model, where processes fail by silently halting, and the Byzantine model, where faulty processes can collude to create the worst-case attack scenario. In this respect, while the Visigoth model can be parameterized to handle arbitrary commission faults, just like BFT systems, it places a bound on the number of faulty processes that are coordinated and work together to intentionally defeat the system.

The Visigoth model has relevant theoretical and practical implications. From a theoretical standpoint, this model fills the spectrum between existing techniques, namely between the synchronous and asynchronous models, and the crash and Byzantine models. From a practical standpoint, VFT enables us to design practical systems for predictable environments like the data centers that support Internet services, while taking advantage of that predictability in order to lower replication factors, thus improving the operational costs (and in some cases the performance) of these systems.

We designed a VFT protocol for state machine replication by adapting an existing protocol for the Byzantine model, and in this process we abstracted some generic constructs that can be reused for adapting other protocols. One of the limitations of our initial adaptation is that it can lead the system to violate safety conditions when the assumptions on the level of synchrony are not met. Since there may, in practice, occur scenarios where more than a given threshold of machines behave asynchronously, it is desirable that in those cases the system halts instead of violating safety. We achieve this through an extension to our initial VFT protocol, which reintroduces more resource expensive majority-based quorums, but only for a tiny fraction of protocol steps.

To gain some insight into how to set the parameters used by our model, we measured the message transmission and processing time in a small research cluster and in Amazon EC2. In addition, we conducted an experimental evaluation of our prototype, which shows significant benefits compared to conventional approaches in terms of throughput per dollar, i.e., the cost one has to pay to rent virtual servers for sustaining a certain level of request execution.

The remainder of this paper is organized as follows. In Section 2 we formalize the Visigoth model and discuss its fundamental properties. Section 3 reports on our experience adapting an existing replication protocol to the Visigoth model, and we discuss an extension to the protocol to preserve safety despite arbitrary asynchrony in Section 4. In Section 5 we analyze some measurement data to understand how to parameterize the model. We provide and discuss experimental results in Section 6. Section 7 overviews relevant related work, and we conclude in Section 8.

2. Visigoth Fault Tolerance

In this section, we present the Visigoth model for implementing replicated systems.

2.1 Relaxing the asynchrony assumptions

Our model builds on the observation that data centers typically operate a well provisioned network [44], leading to fast and predictable response times, as observed in previous measurement studies [50]. Furthermore, modern and upcoming network technologies enable the implementation of a network fabric that makes the probability of experiencing arbitrary latencies negligible [48, 52], while various hypervisor and OS-level techniques allow for the mitigation of interference effects, which can increase message processing times [32, 39]. It is therefore reasonable to expect that most messages are delivered and processed in a timely fashion.

We consequently assume that there is a bound on the time the network takes to deliver a message. This assumption brings us close to what the distributed computing literature calls a *synchronous system*. A synchronous model assumes a maximum delay T for transmitting and processing a message. This assumption allows a synchronous algorithm to proceed in rounds with a duration of T, where in each round all processes can communicate with each other. In this case, it is safe for an algorithm to assume that if a message is not received from a given process in a given round, then that process has clearly crashed.

However, the complexity of the software stack running at each server makes it difficult to guarantee that all processes generate or consume messages in a timely fashion. Occasionally, latency spikes can still be introduced by processes themselves due to operating system effects (*e.g.*, scheduling) and programming language features (*e.g.*, garbage collection stalls). In light of this observation, it is unwise to assume that the delivery bound always holds.

To capture these two effects, we assume a bound on the number of processes that are slow simultaneously. Since slowness might be perceived differently by distinct processes, we assume that the set of slow processes towards different processes is possibly different.

More precisely, to quantify the amount of synchrony, the model makes the following assumption: for any process p, at most s processes are "slow but correct" from the point of view of p, such that it takes longer than T time units for p's messages to and from those processes to be transmitted and processed. Note that this formulation requires a fixed set of

s processes to be correct but slow from the point of view of each process throughout the execution. In practice, this can be relaxed to say that this set only needs to be fixed for sufficiently long (e.g., for a consensus instance to conclude).

Note that, in contrast to the synchronous model, there is no possibility to detect if a process has crashed in this model, since among the processes whose messages were not received within the bound T, there is a mix of faulty and up to s slow but correct processes. However, we can use timeouts to determine that some number of processes must have crashed, even if we cannot pinpoint who they were.

2.2 Relaxing the assumptions on non-crash faults

This aspect of the model is motivated by the need to handle non-crash faults, i.e., faults such as bit flips that originate in the hardware, or, more generally, any type of data corruption in memory, storage, or the I/O path. These faults are much more rare than crashes, and, when they occur, it is often hard to map them to the root cause. As such, they are typically not considered part of the fault model. Instead, the implementation of fault tolerant systems often resorts to digests of both state and messages to protect against simple data corruption. However, wrong outputs that are not caught by these digests may lead to severe outages [3].

While BFT offers a principled approach for handling non-crash faults, several practical concerns push back its adoption [45]. First, the complexity and overheads of BFT, particularly in terms of increased replication factor compared to using crash fault tolerance (CFT) and additional administration overheads, such as managing server keys. Second, the fact that BFT targets a worst case scenario where all arbitrary faults are generated by a malicious attacker or by colluding machines aiming at defeating the system. Such an attack scenario is considered unlikely by practitioners, since data centers have several well managed security measures, such as firewalls, reverse proxies to limit exposure, or sandboxing.

Therefore, our objective is to find a model that provides a principled approach for handling non-crash faults that stem from bit flips or other forms of data corruption, while being less pessimistic than the Byzantine model, which allows for arbitrary and correlated faults. This, however, entails an apparent contradiction due to the fact that bit flips and data corruption can also be arbitrary.

The key observation to solve this contradiction is the following: what distinguishes data corruption from an attack scenario is not the behavior of a single faulty process, but the fact that data corruption is unlikely to affect the same part of the state across faulty processes and consequently generate the same incorrect manifestation at all faulty machines. We also observe that a collusion attack where an adversary controls all faulty replicas and forces their outputs to be compatible is, in practice, the worst-case scenario that Byzantine fault tolerance protocols must address.

Thus, the Visigoth model sets a maximum threshold o of correlated faulty behavior. In this case, the total number of arbitrary faults in the system may exceed o, without endangering safety. The question then becomes what do we mean by correlated faults. To define this, we observe that algorithms for replicating stateful services follow a typical pattern of collecting a set of messages from different processes and, upon reaching a threshold (e.g., a quorum), an action is performed. For example, in the PBFT algorithm, the primary for a given view sends a *new view* message after gathering $2f + 1$ valid *view change* messages for that view [11]. Similarly, the ABD crash fault tolerant algorithm [6], which provides a simple read/write storage interface, uses two RPC rounds driven by the client, where the client collects answers for the pending RPC from a majority of replicas to conclude each round. Therefore, the model states that when an algorithm collects a set $\mathcal{S}$ of messages from different processes and, based on those messages, performs an action, it can assume a bound o on the number of correlated messages that are arbitrarily faulty. This implies that at least $|\mathcal{S}| - o$ messages can be assumed to be generated by processes that correctly followed the protocol.

n	total number of processes
u	threshold on the total of faults (arbitrary + crash)
r	threshold on the number of arbitrary faults
s	threshold on size of any set of *slow but correct* procs.
o	threshold on correlated faulty behavior
T	max transmission time between non-slow processes

Table 1. Parameters used by the Visigoth model.

2.3 Visigoth model definition

Following the notation of UpRight [13], we divide the space of faults into two types of manifestations: a process that sends a message that does not follow the protocol is said to have suffered a *commission* fault; and a process that either crashes or fails to send a message it should have sent is said to suffer an *omission* fault. The maximum number of commission faults tolerated is r and the maximum number of total faults is u.

The Visigoth model can be precisely defined as follows.

1. The system consists of a set of n processes. Each pair of processes communicate by sending and receiving messages over a bidirectional link. The network can lose, duplicate, and arbitrarily delay messages (subject to some additional constraints below). Network links are pairwise-authenticated, which guarantees that if process i receives a message m in the incoming link from process j, then process j sent message m to i beforehand.

2. Each process executes a sequence of steps (actions) triggered upon either: receiving a message, an internal timer expiring, receiving an external input, or a condition in the state becoming true.

3. Processes may fail by crashing permanently, or by suffering a commission fault, otherwise they are non-faulty.

Commission-faulty processes may send any number of arbitrary faulty messages throughout the entire execution (subject to the remaining model constraints).

4. In the definition of the actions of a process for implementing a given protocol, some of these actions can be annotated by the protocol writer as "message collection steps". These must be internal actions resulting from a condition over the state becoming true, and the precondition that triggers these actions must be that the process collected, in its internal state, a sufficiently large threshold t of messages that meet a certain condition C from distinct processes. (Both the threshold t and the condition C are protocol-specific.) For example, a message collection step can be triggered by gathering a majority (or a quorum) of messages that have the same type and contain the same answer from distinct processes.

5. There are at most u faulty processes (either crash or commission-faulty), out of which at most o processes may suffer commission faults that are correlated. We say that commission faults are correlated if they lead to sending incorrect messages that can be used to trigger the same message collection step. More precisely, assuming that S is a state variable, that is local to a process and maintained by the protocol, if the precondition that triggers a message collection step is of the form:

$$|\{m \in S \text{ from distinct } p_i : C(m) = \textit{true}\}| \geq t$$

then there are up to o commission faults that can lead to incorrect messages m that belong to the set above.

6. We define that process i is slow with respect to j if one or more messages from i to j or j to i take longer than T time units to be transmitted. (By not taking longer than T to be transmitted, we mean that the time from the execution of the action that sent a message to the corresponding message receive action does not exceed T.) For any process, there is a maximum number s of other processes that are slow with respect to it.

7. Processes and clocks need not be synchronized, but we assume a bounded clock drift, so that timers can be used to safely determine if *T* has elapsed.

For quick reference, the parameters of the Visigoth model are summarized in Table 1.

2.4 Consensus: Sliding scale between existing models

To understand the fundamental benefits of this model, we show that it leads to a tight lower bound on the replication requirements that fills the spectrum between existing models. The lower bound we state here is on finding a solution to the majority consensus problem. This problem has a practical relevance, since it is at the core of state machine replication, a practical replication technique [43]. Such a solution must obey the following safety (S) and liveness (L) properties.

L1 If a correct process p proposes a value, then that process decides a value;

L2 if a correct process p decides a value, then all correct processes eventually decide a value;

S1 if two correct processes decide v and v', then $v = v'$;

S2 if all correct processes propose v and a correct process decides v' then $v = v'$;

Given this formulation, we prove that the following number of replicas is required to solve this problem under the Visigoth model.

THEOREM 1. *In a VFT system following the model presented above, there is no solution for consensus when* $n < u + \textit{min}(u, s) + o + 1$.

We prove this Theorem in a separate technical report [40], and in the next section we show a transformation of an existing algorithm to the Visigoth model that shows that this bound is tight, i.e., that we can solve the majority consensus problem with $n = u + \textit{min}(u, s) + o + 1$.

This replication factor represents a smooth transition between existing models, where the extreme cases correspond to well-known results: at $f = u = o; s = n$, this is equivalent to asynchronous BFT; at $f = u = o$, $s = 0$ to synchronous BFT with signatures; at $f = u$, $s = o = 0$, to synchronous CFT; and at $f = u$, $s = n$, $o = 0$ to asynchronous CFT.

Throughout the remainder of the paper we focus on the case where $u > s$. This not only has the advantage of significantly simplifying the notation and the proofs (since the $u \leq s$ case needs to be handled separately), but also focuses on the interesting case where the Visigoth model is advantageous. Furthermore, the $u \leq s$ case is very similar and leads to the same solutions as in the well-studied asynchronous setting, since one cannot in that case improve the algorithms by inferring how many processes have crashed after a timeout.

2.5 Discussion

The previous result highlights one of the key benefits of VFT, which is that it reduces the replication factor to solve the fundamental consensus problem from $n \geq 2u+r+1$ (or $n \geq 3f+1$ in the traditional formulation) to $n \geq u+s+o+1$ (or $n \geq f + s + o + 1$) compared to an asynchronous BFT system. This benefit comes at a cost of making additional assumptions compared to the asynchronous model, and therefore the overall correctness is at stake when these assumptions are not met. In particular, when a partition splits the set of processes into two halves, the bound on s could be violated, and the two halves could proceed independently assuming the other half has crashed. This is a fundamental point associated with our gains on replication factor, and cannot be circumvented. To understand why this is the case, one should consider that (1) u can be greater or equal to half of the system size (e.g., when $o = 0$), (2) protocols must be able to make progress despite u crashed processes, and (3) a situation with half of the processes crashed is indistinguishable from a network partition that splits the system in half.

The problem with this is that the two sides of the partition could make progress without coordination, possibly leading to a violation of agreement on the current system state, and consequent violations of the safety guarantees of the replicated system. In Section 4, we show how to transform those safety violations into liveness violations, through an extension to our base replication protocol presented in the next section.

3. VFT-SMaRt design

In this section we demonstrate the practicality and the technical challenges of Visigoth fault tolerance by adapting an existing replication protocol and its implementation to the Visigoth model.

3.1 BFT-SMaRt overview

We chose to adapt the BFT-SMaRt state machine replication library[1] because state machine replication is a generic and widely used method for replicating stateful services [7, 9, 14], and because BFT-SMaRt is a stable software library that is regularly maintained by its developers.

BFT-SMaRt follows the BFT state machine replication protocol presented in [10], whose normal case operation is close to the PBFT protocol proposed by Castro and Liskov [11]. BFT-SMaRt uses $n = 3f + 1$ replicas to tolerate f Byzantine faults. From these replicas, there is a distinguished leader replica, elected by an epoch change protocol that moves the system through a sequence of epochs. The leader for each epoch drives the epoch change protocol in order to start a new epoch, transferring the necessary state concerning requests executed in prior epochs. In general, most protocol steps initiate an action after collecting a quorum of $2f + 1$ messages; such quorums have the property that any two quorums intersect in at least one non-Byzantine faulty replica. In particular, within an epoch, the normal case operation for BFT-SMaRt to execute a client request proceeds as follows.

1. The client sends a PROPOSE message to the leader replica.
2. Upon receiving the request, the leader broadcasts a READ message to all replicas.
3. Upon receiving a READ from the current leader, each replica replies to it with a copy of its state in a READ-REPLY. (A common optimization is to run steps 2 and 3 only once at the beginning of each epoch for all future requests that may execute in that epoch [30], and therefore the common case replication protocol starts in the next step.)
4. Upon receiving $2f + 1$ READ-REPLIES, the leader broadcasts a STATE message to every replica.
5. Upon receiving the STATE message from the leader, each replica inspects the set of states and adopts a value v shown to be locked in a previous instance, or, if no such value exists, it adopts the proposal of the leader. Then each replica broadcasts a WRITE message with the adopted value.
6. Upon receiving $2f + 1$ WRITEs, each replica broadcasts an ACCEPT message to all replicas.
7. Upon receiving $2f + 1$ ACCEPTs, each replica executes the client request and sends a REPLY message to the client containing the execution outcome.
8. The client collects $f + 1$ matching REPLY messages and returns that outcome.

[1] `http://github.com/bft-smart/`

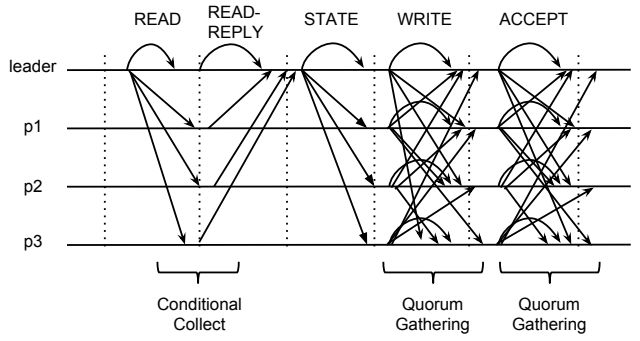


Figure 1. VFT-SMaRt communication pattern.

3.2 VFT protocol design

3.2.1 Overview

BFT-SMaRt was used as a starting point to design a VFT state machine replication protocol: VFT-SMaRt. The protocol uses the set of message exchanges between replicas in Figure 1. This is identical to the message pattern of BFT-SMaRt, which we just described. Note that there are two core message patterns in Figure 1: (1) the first two message steps, corresponding to epoch changes and executed infrequently, where the leader collects information from previous epochs, and forms a certificate consisting of all the information collected and the signatures from the processes that sent it, which can be disseminated to other replicas; and (2) the final sequence of two all-to-all communication steps after the leader relays the client request, which drive the "common case" execution.

To modularize the protocol transformation, we designed a few primitives that can be plugged into different parts of the protocol. These are then composed to form the state machine replication protocol implemented in the VFT-SMaRt library. In particular, the most basic one is the Quorum Gathering Primitive (QGP), which forms the basis for the all to all communication pattern of the normal case operation. The QGP primitive is also used as a building block to construct the conditional collect primitive, which is the primitive that maps to the first core message pattern in Figure 1. For a complete description of these primitives, their composition to form a complete protocol, and all the proofs, we refer the reader to a separate technical report [40].

3.2.2 Quorum gathering primitive

When adapting the protocol to VFT, we spent a large fraction of our time adapting the QGP primitive to work with the Visigoth assumptions. Therefore, instead of trying to describe the entire protocol, we will provide a more thorough treatment of QGP, since it is illustrative of the challenges and the techniques developed to design protocols in the Visigoth model. Considering QGP enables a simpler and more focused discussion of how the new model influences protocol design. We conclude with a brief overview of the remaining main challenges in our protocol design.

Specification. The QGP is initiated by every process asynchronously and consists of every process sending a message to a distinguished gatherer process, allowing this process to collect a quorum of messages. As such, this implements an all to one message pattern, and, if we instantiate a QGP in every process, then we obtain an all to all message pattern such as the one in the last two steps of the protocol in Figure 1. Furthermore, the quorums returned by the gatherer form a dissemination quorum system [34], i.e., a set of subsets (quorums) with the property that any two quorums intersect in at least one correct replica (where correct is defined as non-commission faulty). Another important aspect of this specification is that it is parameterized by a timeout value used by the gatherer (T_{QGP}). To satisfy the intersection property, this timeout must be set in a way that any correct process that satisfies the synchrony bound T is able to convey its message to the gatherer before the timeout. More precisely, the correctness properties of QGP are specified as follows.

DEFINITION 2 (Quorum Gathering Primitive). *A Quorum Gathering Primitive is parameterized by a timeout value T_{QGP}, takes as input at each process p a statement m and outputs at a distinguished gatherer process g a quorum M, consisting of a vector of messages, guaranteeing the following properties:*

- Liveness: if the gatherer is correct, then the primitive eventually returns.
- Safety–Integrity: if the gatherer is correct and delivers M such that some $M[p] \neq$ UNDEFINED and p is correct, then p has statement $M[p]$ as an input.
- Safety–Intersection: if there are two instances of QGP, such that all correct processes that are not crashed and not slow towards the respective gatherer processes initiate the protocol within a maximum δ time window such that $T + \delta < T_{QGP}$, then, if the two correct gatherers p and p' gather M and M' respectively, M and M' intersect in at least one correct replica.

Note that the intersection property is vital for the correctness of any replication protocol that uses QGP, but it is also possibly untenable under the traditional design of quorum-based protocols. This is because a process cannot expect to receive messages from more than $n - u$ processes (since up to u processes may crash), which is not sufficient to ensure the required intersection with other quorums of size $n - u$. Next, we explain how the implementation of QGP in our model overcomes this challenge.

Implementation in the Visigoth model. The first step in our VFT implementation of QGP is simple: the gatherer waits for a quorum of replies and returns that quorum. Given the assumptions of point-to-point authenticated links and by having messages carry a tag that uniquely identify each primitive, this suffices to ensure the integrity property (messages in the returned quorum match the input of the respective processes). However, the central question that needs to be answered by QGP is how large should the quorum size be in order to implement a dissemination quorum system, i.e., ensure the intersection of any two quorums in at least one correct replica.

The initial quorum size that the gatherer tries to collect in QGP is $n - s$. This is sufficient to ensure the intersection property, since any two sets of $n - s$ intersect in more than o replicas, and one of those must be correct. This, however, raises the problem that when there are u faults it may be impossible to gather this quorum size, which would preclude liveness. To address this problem, we can leverage the assumption about the existence of at most s slow processes. This allows us to reduce the quorum size when timeouts ensure that a subset of the replicas that were excluded from the quorum have permanently crashed. In particular, the timing requirements in the specification of the intersection property of QGP allow us to determine that, if a set of x replicas did not reply by the timeout of T_{QGP}, and since at most s of those processes are slow, then at least $x - s$ of those processes must have crashed before the timeout and will not participate in future quorums. Thus, collecting a set of at least $n - u$ replies after the timeout is sufficient to ensure intersection. (If $n - u$ replies are not gathered by the timeout then the gatherer must wait until this number is reached.)

However, the fact that these processes will not participate in future quorums does not mean that they have not participated in a concurrent invocation of QGP, which could lead to safety violations. To understand why this is the case, consider the following example with $u = 2, s = 1, o = 0, n = 4$, and processes a, b, c, d. Suppose that a and b initiate QGPs in parallel, that a is slow towards b and vice-versa, that c crashes after only replying to a, and that d crashes after only replying to b. In this case, the two concurrent quorums that are gathered are $\{a, c\}$ and $\{b, d\}$, thus non-intersecting.

To address this problem, the gatherer contacts all replicas to gather a second (possibly different) quorum with the same rules for the quorum size, but only in the case when less than $n - s$ processes reply by the timeout. In the above example, the second round would reach the quorum of $\{a, b\}$ and the two invocations would return $\{a, b, c\}$ and $\{a, b, d\}$. While this requires an extra round-trip to gather the second quorum,

this is only in the case when more than s replicas are not reachable within the timeout.

Correctness. Even though we present a correctness proof of the entire protocol in a separate technical report [40], we sketch here a correctness proof for the intersection property of this primitive, since this intersection is key to the correctness of not only our protocol, but of most protocols for replicating stateful services.

We focus here on the most challenging case when the two quorums that are output are small quorums of size $n - u \leq Q1, Q2 < n - s$. Since each instance of the primitive goes through two phases of gathering quorums, it must be the case that the first phase of one of the primitives (say $Q1$) finishes before the second phase of the other primitive (say $Q2$) begins. Say that the first phase quorum of $Q1$ has size $Q1 = n - u + a$, and that the final quorum returned by $Q2$ has size $n - u + b$. At the instant when $Q1$ ends the first phase, by the fault model, there are at least $u - a - s$ crashed processes (i.e., all non-responsive ones except s slow processes). This implies that the system size at this time is $n' \leq n-u+a+s$. Given this system size, the intersection in one correct process is guaranteed if $Q1+Q2-n'-o > 0$. This gives:

$Q1 + Q2 - n' > o$
as we know that $n - u + a + s \geq n'$ we can translate to:
$Q1 + Q2 - n + u - a - s > o$
$\Rightarrow n - u + a + n - u + b - n + u - a - s > o$
$\Rightarrow u + o + s + 1 - u + a - u + b + u - a - s > o$
$\Rightarrow 1 + b > 0$

3.2.3 Other challenges in VFT-SMaRt design

Next, we discuss two other challenging points behind the transformation of the original protocol to the Visigoth model. The remaining protocol details are described and proven correct in a separate technical report [40].

Challenge 1: Certificates. The first challenge is related to the use of certificates consisting of quorums of signed messages (or statements), namely in the epoch change protocol. This is a common pattern in BFT protocols, allowing process A holding the certificate to demonstrate to process B that a certain action took place at a dissemination quorum, thus ensuring that no contradictory action could have taken place (since the correct process in the intersection of any two dissemination quorums would not allow for such contradictory action). While these certificates can be produced trivially in traditional models, in VFT there are two factors that complicate this task. These are the fact that quorums do not have a constant size, and the possible presence of commission faults. Because of these, when a certificate uses a size smaller than $n - s$, it is not clear whether the use of a small quorum is legitimate or whether it was due to a commission fault (in case that $o > 0$).

To address this challenge, certificates follow the vector of signatures model only in case when they have size $n - s$, since this allows for intersection with any other quorum irrespectively of the number of crashed processes. When the number of processes in a certificate is smaller than $n - s$, the process putting together the certificate must also collect a statement from the processes that participate in the quorum forming the certificate attesting to the fact that it is legitimate to use a smaller quorum due to the unreachability of the remaining processes.

Challenge 2: Setting timeouts. The second challenge is how to set the timeout parameters that are used by QGP. These timeouts are related to the maximum relative delay that correct and non-slow processes might have when instantiating the primitive. (For processes that are slow this is not problematic since their messages can be arbitrarily delayed anyway.) The problem that is faced when computing this maximum delay is that it is possible that delays accumulate throughout the execution of a protocol that uses QGP. If this is not taken into account, we risk to incorrectly identify certain processes as being slow, which is the same as saying that the premise of QGP that correct and non-slow processes start the protocol in a timely manner is not met, and the properties of QGP do not hold.

To address this, the protocols must set the value of the timer so that, if the maximum length of the chain of messages that lead to a remote process sending a message that we are waiting for is l, then the timer is set to $(l + 1) \cdot T$. While it is clear that this modification works for cases when the maximum delay to move from one phase in the chain to the next is at most T, it is less clear what happens if the timeout expires and the quorum to move from one phase to the next still has not been gathered because we are waiting for slow processes that will take longer than T. This situation turns out not to be problematic because in such cases it is necessarily the case that more than $u - s$ processes have crashed. In that case, we are operating with quorums of at least $n - u$ processes, which are a majority of the non-crashed processes, thereby ensuring quorum intersection irrespectively of the wait time.

4. Trading liveness for safety

The VFT-SMaRt protocol highlights an important advantage of the Visigoth model: it leads to more resource-efficient protocols than previous, more pessimistic models. In particular, when tolerating only machine crashes and compared to the asynchronous model, we are able to cut the replication factor n from $n = 2u + 1$ (e.g., the replication required by the Paxos protocol) to $n = u+s+1$, when $u > s$. However, this resource efficiency comes at a cost. When the bound of slow but correct processes is violated, the VFT-SMaRt protocol may fail to meet its safety conditions. Violating safety is highly undesirable, since it might lead, for instance, to inconsistencies in the state of the system. Practical systems in

fact often prefer to violate liveness instead of safety. For example, it is better to halt the auction system than to declare two users to be the winners of the same auction. (In fact, the Paxos protocol has the advantage of overcoming the FLP impossibility result [21] by always preserving safety but only ensuring liveness under additional synchrony assumptions.)

To overcome this concern, we propose an extension to the VFT-SMaRt protocol that turns potential safety violations when the bound s of slow but correct processes is exceeded into liveness violations. This extension solves the following problem: when the bound s is exceeded, processes may incorrectly assume that a certain number of processes have crashed, and hence they may assume that it is safe to proceed with a small quorum (smaller than $n - s$). The intersection guarantee between two small quorums only holds when the assumptions about crashed processes also holds, otherwise two small quorums might not contain any process in common. This empty intersection enables executions in which we have a "split brain", i.e., that lead to state divergence. To address this problem, we reintroduce majorities to decide which small quorums can be used within a protocol epoch, where the role of the majority is to act as an "oracle" preventing the use of two small quorums without any intersection within a given epoch. Reintroducing majorities raises the question of whether this defeats the original goal of VFT, which is to design resource-efficient protocols. However, majorities are only used sparingly, in key steps of the protocol, and therefore a single majority-based group can be used for a large number of resource-efficient VFT groups.

In more detail, we extend the protocol by adding a logically centralized oracle that is safe in the presence of asynchrony. (In practice, this oracle can be implemented using Paxos, or any other consensus protocol.) The oracle is queried by processes wanting to make use of a small quorum within an epoch: any protocol step that takes an action upon receiving a small quorum now has to wait for an authorization from the oracle (or proof that this authorization was given). The behavior required by the oracle is simple: it only authorizes small quorums within an epoch that intersect with all other authorized quorums for that epoch (or, in the case where commission faults are tolerated, that intersect in $o+1$ processes). In practice, the implementation of the oracle that allows the VFT system to make progress is the following: when the oracle receives the first such request for a given epoch, it authorizes this request, and subsequently it denies all requests that do not meet the intersection property in that epoch (for each VFT-group within the data center).

5. Deployment considerations

We conducted a small measurement study to understand how one can set the parameters of the VFT model, and what are the tradeoffs involved in those choices. This study ran in two environments with the following characteristics.

Research cluster: We used 90 machines of this cluster, each one with 2×6 core Intel Xeon X5650 CPUs at 2.66 GHz, with 48 GB of DDR3 RAM. Each machine had 2 distinct gigabit network interfaces, which are viewed and accessed as a single virtual interface using the Line Aggregation Control Protocol (LACP). These machines are interconnect through 2 redundant switches, each with a maximum throughput of 10 gigabits. All machines were running Linux. No virtualization is in place, and deployed applications were conducting research experiments, such as mining large data sets and searching for bugs.

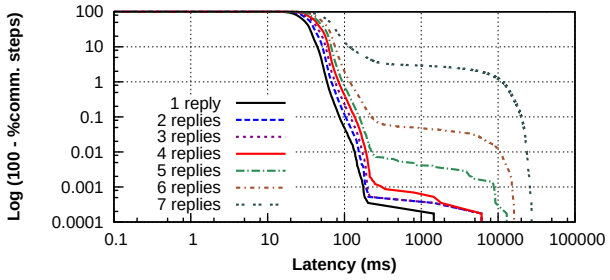
Amazon EC2: Our Amazon EC2 deployment used 15 large instances running a Linux virtual machine, located in the us-east-1 availability zone of Amazon's US East Coast data center. Each instance had 7.5 GB of RAM, and a CPU with 4 EC2 computational units (distributed across 2 virtual cores). We do not know any details concerning the network infrastructure in place at the data center.

To emulate the behavior of an RPC-based distributed protocol, we organized groups of seven processes on different machines/instances. Each process p in each group periodically (every second) initiates a *communication round*, where it measures the time taken by a group RPC to all the processes in the group. The handler of this RPC exercises a combination of various machine resources, similarly to the actions taken by many distributed protocols, namely (1) verifying a MAC, (2) preparing a reply message and a MAC for that reply, and (3) storing both the request and the reply to stable storage. Request and reply message sizes are 50KB in size, similarly to the average workloads observed in production data centers [48].

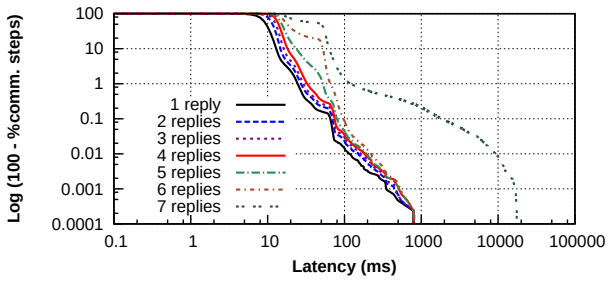
To study how to set s and T in these deployments, we ask the question of how long we need to wait if we want to make sure that the messages from all but the s slowest processes in the group arrive within that time. To answer this question, we depict in Figure 2 the CDFs for the time required for processes to gather the nth reply in the group, $n \in \{1, ..., 7\}$. These results aggregate more than 4.0×10^6 pairs of request and replies for the research cluster and 2.9×10^6 for EC2.

The results show that, despite the fact that processes gather most replies in only a few to tens of milliseconds, in a small fraction of the cases, processes were required to wait for longer periods of time, particularly in the research cluster (Figure 2(a)).

For example, in the less predictable research cluster (Figure 2(a)), all processes gather 6 replies in less than $500ms$ for 99.9% of the communications steps. On Amazon EC2 (Figure 2(b)), in all communication rounds, processes were able to collect 6 replies in less than 1s, which is significantly less time than the worst case latency, which is on the order of tens of seconds. Thus, in both these environments, there are substantial gains when moving from $s = 0$ to $s = 1$, in terms of setting a more aggressive T, and therefore not having to wait for long timeouts.



(a) Research cluster.



(b) Amazon EC2.

Figure 2. CDFs for the time required for processes to gather different numbers of replies.

6. Evaluation

We implemented VFT-SMaRt by extending the publicly available BFT-SMaRt code. In addition to an implementation of our base algorithm, we also implemented the extension described in Section 4 by making VFT-SMaRt act as a client of a ZooKeeper [25] replica group, which implements the oracle logic. This section evaluates these systems experimentally. Our goal is to understand the benefits and costs of employing VFT in place of traditional techniques.

6.1 Experimental setup

We run our experiments on a cluster of 20 6-core 2.67GHz Intel(R) Xeon(R) CPU X5650 machines with 48 GB of RAM, configured with 24 working threads, connected to the network through a bonding interface composed by two 1Gbps NICs. Among these machines, ten were used exclusively to run clients, and the remaining machines were used to run server replicas. Client requests for all experiments are issued in a closed loop. The load on the replicated system is varied by increasing the number of request threads running on client machines. Each data point represents the average latency and throughput of a run with a fixed configuration, where each run lasts for 4 minutes in total and the first minute is considered a warm up period and is excluded from the average.

The performance of VFT-SMaRt under different configurations is compared to (1) asynchronous crash fault tolerant replication (CFT), which was implemented by configuring VFT-SMaRt to tolerate zero commission faults (i.e, $o = 0$) and disabling all timers used in the VFT implementation; (2) asynchronous Byzantine fault tolerant replication (BFT), by comparing with the original BFT-SMaRt codebase.

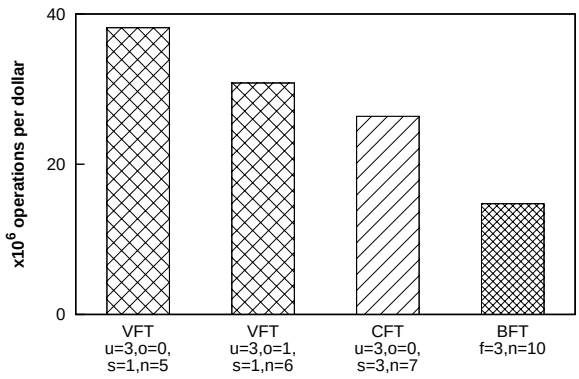


Figure 3. Cost effectiveness of VFT versus traditional approaches using microbenchmarks.

6.2 Cost effectiveness

The main benefit of VFT when compared to alternatives operating under weaker guarantees like synchronous or crash-tolerant algorithms is the fact that it is able to tolerate message or processing delays and arbitrary data corruption. In comparison to traditional options that provide stronger guarantees (namely asynchronous or BFT replication), VFT leads to a more cost effective solution due to a lower replication factor.

In our first set of experiments we try to quantify this cost effectiveness by measuring, for different configurations of VFT, the throughput per dollar of the replicated system, i.e., the server cost for sustaining a certain level of request execution. This is then compared to the cost when deploying asynchronous CFT and BFT systems.

Figure 3 shows, for CFT, BFT, and different configurations of VFT tolerating up to 3 total faults, the achieved throughput per dollar, i.e., the ratio between the maximum throughput that is achieved under each configuration and the cost of renting the virtual servers required to deploy each configuration on Amazon EC2[2]. As expected, when compared to the classical most pessimistic option (asynchronous BFT), the throughput per dollar of VFT is 109% to 159% higher, while still covering to a large extent the relevant class of faults for data center environments. Furthermore, when compared to CFT, VFT is still more cost effective by as much 79%. This is due to being less pessimistic regarding the assumptions concerning asynchrony (i.e., slowness), since we used VFT configurations with $s = 1$.

6.3 Microbenchmarks

Next we try to gain a deeper understanding of the performance of VFT. To this end, we run a microbenchmark where the replicated service only supports a single operation that

[2] With a cost of $0.28 per Hour for Linux instances in US-West availability zone.

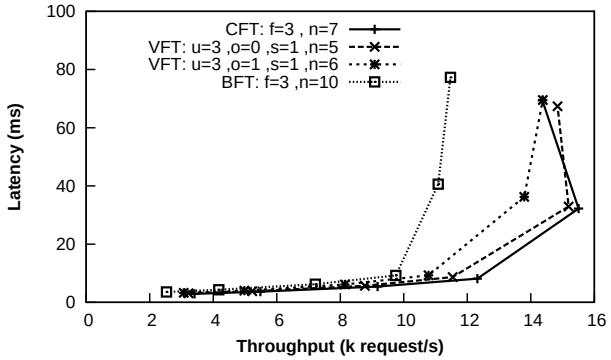


Figure 4. Throughput vs. latency for CFT, BFT, and two different VFT configurations: one tolerating crashes, one slow process, and an arbitrarily faulty process, and another configuration tolerating only crashes and one slow process.

receives as argument 1 KB of data and writes that data to the service state before returning. All clients perform consecutive invocations of this operation. Since this operation modifies the state it cannot take advantage of the traditional optimizations for read-only operations [11].

In these experiments, we vary the number of requests issued by each client so that different levels of load are imposed on the replicated system. For each different load imposed on the system, we compute the average latency and throughput perceived by the set of clients. We then plot these points in a throughput/latency curve.

Figure 4 shows a performance comparison between VFT and traditional approaches (asynchronous CFT and BFT), where all systems are configured to tolerate a total of three faults. In this experiment we used two different VFT configurations: a middle-ground configuration where we tolerate up to one correlated commission fault and up to one slow process, and a crash fault tolerant configuration of VFT with up to one slow process. The results show that the latency values observed across all systems are relatively similar when the system is not saturated. The maximum throughput of VFT is consistently higher than the original BFT-SMaRt code, due to the fact that VFT is using fewer replicas, smaller quorums, and therefore the number of messages being sent and processed is also smaller. When comparing VFT to CFT, there are two opposing effects. On the one hand, the number of VFT replicas is smaller, leading to the effect mentioned above. On the other hand, there is the need to manage timers in the implementation of VFT protocols, which are not used in CFT. When the system is under a high load, these timers are constantly being set and reset, which interferes with the processing of protocol messages. The combination of these two effects explain why CFT achieves a higher throughput than VFT when the total number of replicas differs only by one, and a lower throughput when it differs by two.

In the next set of experiments, we try to understand the sensitivity of the performance of VFT-SMaRt to the parameters of VFT. To this end, we compare the performance of VFT configurations when we fix the replication factor but trade one of the parameters for another. The results in Figure 5 show that trading u for o has almost no effect on system performance, whereas increasing s while decreasing one of the other two parameters has a significant positive impact on the maximum throughput. This happens because, in the common case, the protocol gathers a quorum of $n - s$ messages before moving from one stage of the protocol to the next. As such, increasing s makes it easier to gather quorums in a shorter amount of time, and therefore allows the protocols to move faster from one stage to the next.

6.4 Extended protocol and performance under faults

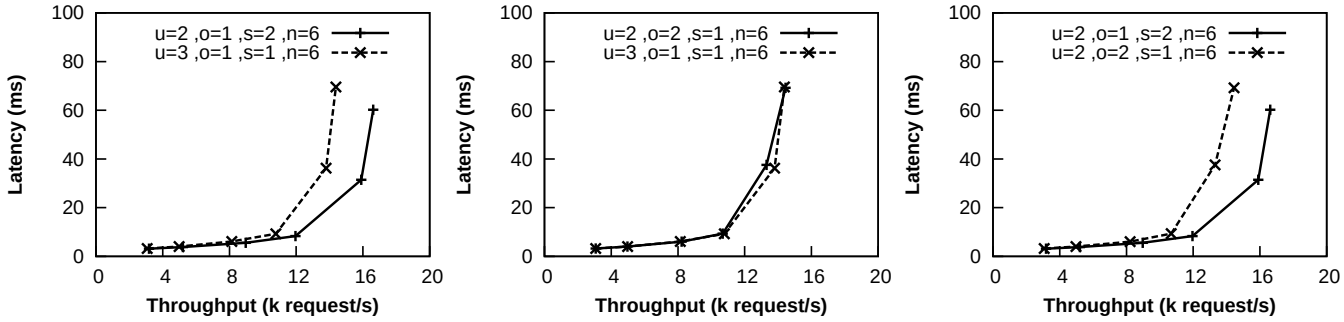
In the previous set of experiments, every replica ran normally during the entire execution. In this section, we show how faults (both permanent and transient) during a run affect the performance of the replicated system. Furthermore, we also evaluate in this experiment the performance of the extended version of our protocols described in Section 4.

To this end, we run three different configurations: a BFT configuration with $f = 2$, hence $n = 7$, our base VFT protocol configured to tolerate the same number of total faults, $u = 2$, but with $s = 1, o = 1$, hence $n = 5$, and the same parameters applied to our extended VFT protocol. In the extended protocol, the oracle that authorizes small quorums is deployed by configuring ZooKeeper to tolerate two crash faults, i.e., using a total of five replicas.

During each run of these experiments, we injected one crash fault at one of the replicas, and 120 seconds later we caused a slowdown of one replica, by forcing it to sleep for 15 seconds.

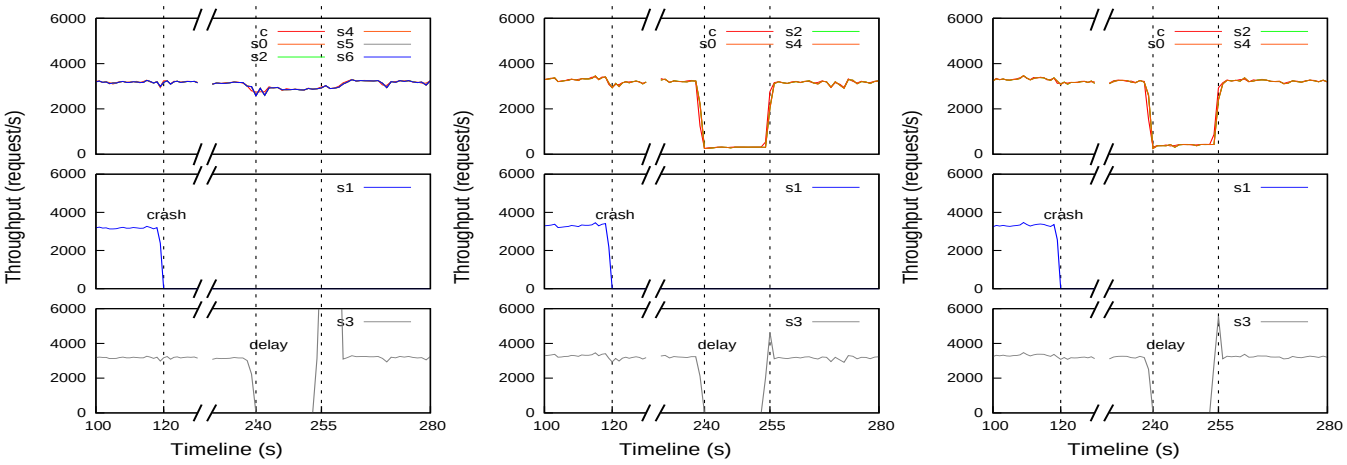
Figure 6 presents, in the y axis, the throughput in number of operations per second. The x axis represents the time into the trace. In these traces, we mark the instants when each of the replica stops, and also when the second replica resumes the operation.

Several observations can be made regarding the results from this experiment. First, none of the systems are noticeably affected by the first fault, since they can still make use of the quorums they were using before that fault. In particular, the VFT systems still use large quorums of $n-s$ after the first crash. Second, when the second replica is suspended, BFT is relatively unaffected since it is still able to gather quorums of $2f + 1$ replicas, whereas VFT has to resort to small quorums of $n-u$, which imply waiting for the timeout before making progress. The consequence of this wait time is a large drop in the throughput of the system. However, this throughput is still above zero, which still gives opportunity to perform a reconfiguration to regain normal case performance. Furthermore, this lower throughput after the second fault could be further improved by increasing the batch size, since a larger number of requests would be executed by each instance of consensus. Finally, the performance of the extended version of our protocols is approximately the same



(a) Trading u for s (b) Trading u for o (c) Trading s for o

Figure 5. Throughput versus average latency for VFT systems configured with $n = 6$ replicas varying u, s, o.



(a) BFT (b) VFT (c) VFT Oracle

Figure 6. System throughput under replica crashes, c stands for clients while s for service replicas.

as that of the base version, which is expected since the oracle only intervenes in a particular point in the execution of the protocol (when small quorums are first put to use). Note that the throughput appears to drop before the instant when the process crashes, but this happens due to the way the throughput is measured over time slots. In particular, the crash affects the throughput measurement over a time interval that has started prior to the crash event.

7. Related work

We survey work that considers timing assumptions as well as non-crash fault behavior in distributed systems.

Timing assumptions. In asynchronous systems, Fischer, Lynch, and Patterson have shown that consensus is impossible to solve when processes can crash [21]. Dwork, Lynch, and Stockmeyer have shown that assuming an unknown bound for message delays or a known bound that holds after some global stabilization time enables solving consensus [19]. Such bounds can equivalently be encapsulated in a failure detection abstraction [12]. Cristian and Fetzer propose the timed-asynchronous model [17] where processes have a bounded drift rate, allowing processes to measure time intervals with a known and bounded error, and to detect untimely events accurately. Our work strikes a different balance between synchrony and asynchrony, as it models bounded asynchrony, which leads to a spectrum of replication solutions that depends on those bounds.

Guerraoui and Schiper introduced a new class failure detectors called Γ-Accurate [23], where the accuracy property of failure detectors is relaxed to focus only on a (fixed) subset of processes in the system Γ. The authors have shown in which conditions these failure detectors allow to solve consensus in a fully asynchronous system where processes can crash. In contrast, we leverage data center properties to build protocols that are more resource-efficient by capturing a continuum between synchrony and asynchrony.

The work by Aguilera, Chen, and Toueg [2] proposes the use of heartbeat-based failure detectors to achieve quiescent reliable communication and solve consensus in partitionable networks. However, the authors assume a fully asynchronous system where network partitions are permanent though not necessarily isolated, which means that given a system with

two partitions say A and B, processes in A might be unable to send messages to processes in B while processes in B can still be able to send messages to processes in A. However, consensus is only achievable if a partition can receive messages from a majority of processes in the system. In contrast, VFT considers both crash and arbitrary faults, while enabling progress to be made even in scenarios where a majority of processes is not available.

Aguilera et al. [1] introduced a new model for partially synchronous systems based on the notion of set timeliness. Different from VFT, their model does not bound the number of slow processes and consequently cannot operate with smaller quorums as we have shown in this work.

Veríssimo and Casimiro proposed the Timely Computing Base model [47]. In their model, they assume a subsystem for transporting regular payload, potentially asynchronous, and a separate subsystem that provides the ability to transmit control messages with timing requirements in a synchronous fashion. This led to a generalization called the wormhole hybrid distributed system model, where there can be secure and timely components in the system [46][16]. In contrast, the Visigoth model does not rely upon a separate subsystem. Furthermore, our model does not require identifying asynchronous and synchronous components.

The communication failure model [42] is tailored to environment like wireless or ad-hoc networks. This model associates faults with communication links instead of processes. In this context an impossibility of solving k-agreement when each process can have more than $n - 2$ links that can lose messages is proven. Our model targets a different environment and as such makes different choices regarding the types of faults and message delays that are allowed.

In the transmission fault model [8, 36] instead of allowing for a set of faulty processes, the messages themselves can be arbitrarily lost or corrupted before their reception. This has the advantage of allowing for a broader range of fault scenarios that are not allowed in other models, e.g., where each process sends a single corrupted message. While this goal is orthogonal to ours, this model has in common with VFT the fact that it allows for reasoning about faults without pinpointing faulty processes.

Handling non-crash faults. Processes that behave arbitrarily are called Byzantine after the seminal paper of Lamport, Shostak, and Pease [31]. Castro and Liskov proposed a practical protocol for Byzantine state machine replication [11]. The UpRight model splits Byzantine faults into commission and omission [13]. The Visigoth model follows the notation of the UpRight formulation but could have also been expressed in terms of the traditional formulation, leading to $n = f + o + s + 1$. Differing from UpRight, VFT introduces new synchrony models and does not limit the absolute number of commission faults in the system to remain safe. Instead, VFT limits the number of correlated faults. ZZ [49] reduces replication requirements for normal case operation by keeping replicas in a dormant state until a fault is detected, which leads these replicas to become active. This feature relies on virtualization for fast state update of dormant replicas, although performance can be affected by the size of the state. CheapBFT[28] leverages trusted devices built with FPGAs to enable resource efficient BFT and reduce the quorum size to $2f + 1$. (In contrast, VFT does not require trusted devices.) Raft [38] and EPaxos [37] are recent agreement protocols. Both were designed assuming CFT. Raft improves understandability while EPaxos leverages application knowledge to deterministically order operations, thus removing the bottleneck at the leader. All these protocols can benefit from VFT, potentially leading to lower replication requirements by capturing arbitrary fault manifestations while not requiring algorithms to worry about the worst-case attack scenario of collusion among commission faulty processes, and also capturing a continuum between synchrony and asynchrony.

Fault detectors have also been explored as a way to design reliable systems in the Byzantine model. In particular the work by Malkhi and Reiter [35] proposes the use of unreliable intrusion detectors to enrich asynchronous systems that can be subject to arbitrary malicious faults. Their fault detector infers arbitrary malicious faults in several ways, which include a process remaining silent when it should transmit a message, or a process transmitting a message that is either malformed, out-of-order, or unjustifiable. Another work shows how to solve consensus in an asynchronous Byzantine system resorting to such fault detectors [29]. Similarly to classical BFT protocols, they assume arbitrary collusion and an asynchronous system, and hence require a replication factor of $3f + 1$.

BFT2F [33] provides weaker than common guarantees in BFT systems when the fault threshold f is exceeded. This is complementary to our work, in that we can also use a similar approach to provide some guarantees if the assumptions in our model are not met.

Several fault models allow for arbitrary corruption of the state of one or several processes, but assume that processes always run the correct code, a set of assumptions that differ from the ones made by the Visigoth model. Self-stabilization [18] assumes that any process in a distributed system can transition to an arbitrary state, and designs distributed algorithms that run in an infinite loop that ensures that the system converges to a correct state in such cases. The Arbitrary State Corruption (ASC) model of Correia *et al.* is a fault model that also focuses on arbitrary state corruption while preserving the protocol code correct [15]. This work proposes a technique called ASC-hardening, which can be used to transform crash-tolerant protocols into ASC-tolerant ones. Different from the Visigoth model, their model focuses on the faults of an individual process, and ASC-hardening requires processes to be implemented following a given structure based on event handlers and messages.

In contrast, a VFT state machine replication protocol relies only upon replicating a deterministic service in a number of machines. On a similar proposal but at a lower level, Schiffel *et al.* proposed a fault detection mechanism targeted at hardware faults [20]. The idea is to develop checks for various operators based on a proposed encoding, and extend a compiler to insert those checks in the application. This is a fundamentally different approach that takes every instruction as potentially operating on corrupted data and validates its results using a redundant local computation. SWIFT is a also a compiler-based technique, based on the Single Event Upset (SEU), in which one bit is flipped throughout the entire program [41]. In contrast to all these proposals, the Visigoth model follows the traditional replication approach for deriving fault-tolerant distributed systems, and it does not assume that the code running on faulty processes is correct.

8. Conclusion

In this paper, we presented Visigoth fault tolerance, a technique for developing robust distributed protocols for data center environments. We believe this paper can spawn a series of interesting research avenues, such as applying VFT to other systems, determining how to parameterize the system to make the assumptions tenable in different environments, or exploring in depth the fundamental properties of VFT.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Thomas Moscibroda, for their feedback. The research of R. Rodrigues is funded by the European Research Council under ERC Starting Grant No. 307732. This work is partially funded by FCT under project PEst-OE/EEI/UI0527/2014.

References

[1] M. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Partial synchrony based on set timeliness. *Distributed Computing*, 25(3):249–260, 2012.

[2] M. K. Aguilera, W. Chen, and S. Toueg. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theoretical Computer Science*, 220(1):3 – 30, 1999.

[3] Amazon. Amazon S3 Availability Event. `http://status.aws.amazon.com/s3-20080720.html`, July 2008.

[4] Amazon. S3 data corruption? `https://forums.aws.amazon.com/thread.jspa?threadID=22709`, June 2008.

[5] Amazon. Possible corruption of small percentage of S3 data. `https://forums.aws.amazon.com/thread.jspa?messageID=262676`, July 2011.

[6] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.

[7] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[8] M. Biely, J. Widder, B. Charron-Bost, A. Gaillard, M. Hutle, and A. Schiper. Tolerating corrupted communication. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 244–253, 2007.

[9] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation (OSDI)*, pages 335–350, 2006.

[10] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (Second Edition)*. Springer, 2011.

[11] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.

[12] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[13] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP)*, pages 277–290, 2009.

[14] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI)*, pages 251–264, 2012.

[15] M. Correia, D. G. Ferro, F. P. Junqueira, and M. Serafini. Practical hardening of crash-tolerant systems. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, pages 453–466, 2012.

[16] M. Correia, N. F. Neves, and P. Verissimo. How to tolerate half less one byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 174–183, 2004.

[17] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, 1999.

[18] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM (CACM)*, 17(11):643–644, 1974.

[19] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[20] C. Fetzer, U. Schiffel, and M. Suesskraut. AN-encoding compiler: Building safety-critical systems with commodity hardware. In *Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security (SAFECOMP)*, pages 283–296, 2009.

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

[22] D. F. Galletta, R. Henry, S. McCoy, and P. Polak. Web site delays: How tolerant are users? *Journal of the Association for Information Systems*, 5:1–28, 2003.

[23] R. Guerraoui and A. Schiper. Gamma accurate failure detectors. In *Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG)*, pages 269–286, 1996.

[24] J. Hamilton. Perspectives - overall data center costs. `http://perspectives.mvdirona.com/2010/09/overall-data-center-costs/`, 2010.

[25] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX annual technical conference (ATC)*, 2010.

[26] M. Isard. Autopilot: automatic data center management. *SIGOPS Operating Systems Review*, 41(2):60–67, 2007.

[27] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. volume 0, pages 245–256, 2011.

[28] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. Cheapbft: Resource-efficient byzantine fault tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*, pages 295–308, 2012.

[29] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Solving consensus in a byzantine environment using an unreliable fault detector. In *Proceedings of the 1997 International conference on Principles Of Distributed Systems (OPODIS)*, pages 61–76, 1997.

[30] L. Lamport. Paxos made simple. *SIGACT News*, 32(4):51–58, 2001.

[31] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[32] J. Leverich and C. Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *European Conference on Computer Systems (EuroSys)*, 2014.

[33] J. Li and D. Mazières. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *Proc. 4th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 131–144, 2007.

[34] D. Malkhi and M. Reiter. Byzantine quorum systems. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing (STOC)*, pages 569–578, 1997.

[35] D. Malkhi and M. Reiter. Unreliable intrusion detection in distributed computations. In *Proceedings of the 10th IEEE Workshop on Computer Security Foundations (CSFW)*, 1997.

[36] Z. Milosevic, M. Hutle, and A. Schiper. Tolerating permanent and transient value faults. *Distributed Computing*, 27(1):55–77, 2014.

[37] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, pages 358–372, 2013.

[38] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (ATC)*, pages 305–319, 2014.

[39] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, 2014.

[40] D. Porto, J. Leitão, C. Li, A. Kate, A. Clement, F. Junqueira, and R. Rodrigues. Lower bound and correctness proofs for consensus in the visigoth model. Technical report, Nova University of Lisbon, 2015.

[41] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August. SWIFT: software implemented fault tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 243–254, 2005.

[42] N. Santoro and P. Widmayer. Time is not a healer. In *Proceedings of the 6th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 304–313, 1989.

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

[44] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Networking data centers randomly. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2012.

[45] Y. J. Song, F. Junqueira, and B. Reed. BFT for the skeptics. Extended abstract for talk at BFTW3: Why? When? Where? Workshop on Theory and Practice of Byzantine Fault Tolerance, 2009.

[46] P. Veríssimo. Uncertainty and predictability: Can they be reconciled? In *Future Directions in Distributed Computing*. 2003.

[47] P. Veríssimo and A. Casimiro. The timely computing base model and architecture. *IEEE Transactions on Computers*, 51(8):916–930, 2002.

[48] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron. Better never than late: meeting deadlines in datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 conference*, pages 50–61, 2011.

[49] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet. Zz and the art of practical bft execution. In *Proceedings of the 6th Conference on Computer Systems (EuroSys)*, pages 123–138, 2011.

[50] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: avoiding long tails in the cloud. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation (NSDI)*, pages 329–342, 2013.

[51] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 29–42, 2008.

[52] K. Zarifis, R. Miao, M. Calder, E. Katz-Bassett, M. Yu, and J. Padhye. Dibs: Just-in-time congestion mitigation for data centers. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*, 2014.