CloudBFT: Elastic Byzantine Fault Tolerance

Rodrigo Nogueira, Filipe Araujo, and Raul Barbosa
Faculty of Sciences and Technology
University of Coimbra
3030-290 Coimbra, Portugal
Email: {ran, filipius, rbarbosa}@dei.uc.pt

Abstract—**Cloud computing is increasingly important, with the industry moving towards outsourcing computational resources as a means to reduce investment and management costs, while improving security, dependability and performance. Cloud operators use multi-tenancy, by grouping virtual machines (VMs) into a few physical machines (PMs), to pool computing resources, thus offering elasticity to clients. Although cloud-based fault tolerance schemes impose communication and synchronization overheads, the cloud offers excellent facilities for critical applications, as it can host varying numbers of replicas in independent resources. Given these contradictory forces, determining whether the cloud can host elastic critical services is a major research question. We address this challenge from the perspective of a standard three-tiered system with relational data. We propose to tolerate Byzantine faults using groups of replicas placed on distinct physical machines, as a means to avoid exposing applications to correlated failures. To improve the scalability of our system, we divide data to enable parallel accesses. Using a realistic setup, this setting can reach speedups largely exceeding the number of partitions. Even for a wide variation of the load, the system preserves latency and throughput within reasonable bounds. We believe that the elasticity we observe demonstrates the feasibility of tolerating Byzantine faults in a cloud-based server using a relational database.**

Keywords*—*Distributed systems, fault-tolerant algorithms, Byzantine faults, security, dependability.

I. Introduction

When compared with traditional in-house solutions, clouds offer simplicity, security, dependability and performance at very competitive costs. Indeed, the economies of scale and a large customer base enable cloud providers to acquire and manage large server farms and to accommodate demand peaks. However, this model only works if clients run algorithms that scale, by using more resources under heavier loads and fewer resources otherwise. This has motivated many cloud providers to encourage, through pricing, the use of their non-relational data stores, to enable parallel, unsynchronized accesses.

Unfortunately, these models do not provide all the ACID properties of typical relational databases (Atomicity, Consistency, Isolation and Durability). In fact, relational databases provide a much more powerful model with a large existing software base, finely optimized and deeply understood by programmers. However, using relational databases in a Byzantine fault-tolerant (BFT) algorithm is expensive due to the synchronization needed for replicas and databases. Some researchers argue that the subtleties of replication collide with elasticity, a driving force of the cloud [1]. Here, we experimentally evaluate the opposite point of view. We create a straightforward web server architecture, called "CloudBFT", to evaluate the feasibility of a cloud-based Byzantine fault-tolerant architecture.

Consistency and isolation (the 'C' and 'I' in ACID) are particularly difficult to obtain in replicated databases. The same pair of database operations arriving in different orders at the replicas may result in different views for each operation and will eventually cause inconsistencies as data starts to diverge. Solving this issue in the relational model is not easy, because tables have foreign keys referring to other tables, thus making independent accesses very difficult to achieve. Locking can help here, but fine-grained locks involve low-level programming, because SQL expressions are dealt with by the database, whereas coarse-grained locks, controlled by the middleware, may impair performance, if they are not well done.

Properly isolating resources in the cloud is also a relevant problem. Currently, cloud providers try to concentrate their clients as much as possible, depending on the resources they need, on the internal traffic they generate, or even on the programs they run, to save memory [2], [3], [4], [5]. Despite bringing economical benefits, consolidation affects current fault-tolerant techniques, because these assume independent failures. If redundant processes share the same hardware, they become much more exposed to simultaneous hazards. For this reason, in our solution, we need to ensure the dispersion of replicas by different servers, availability zones, or regions according to the services supported by the cloud provider(s), and, at the same time, promote the consolidation of resources that may run together to control costs.

To keep our approach simple, we resort to well-established practices: a three-tier architecture, with presentation, business and data layers. We need triple replication for BFT and we divide the data for scalability. In our experiments we use the OLTP TPC-C benchmark [6] and partition the data based on the warehouses. This benchmark provides a realistic setting for an online shop, with more clients requiring more warehouses. Since only a small fraction of the operations access more than one warehouse, in most cases (the remaining) we can access warehouses in parallel. To ensure isolation and consistency, we use coarse-grained locks.

To achieve Byzantine fault-tolerance we use the MinBFT algorithm [7]. A distinctive feature of this algorithm is a tamper-proof Trusted Platform Module (TPM) to sign and verify messages. This TPM only signs the same message once, thus preventing attackers from sending different versions of the same message to different receivers. This property enables MinBFT to tolerate a fixed number f of Byzantine failures, with only $2f+1$ replicas, instead of $3f+1$. Existing technology enables multiple VMs to share a single TPM [8], thus opening TPM services to the cloud datacenter.

To deal with the different partitions (warehouses in TPC-C), we create groups of $2f + 1$ VMs (replicas) running on different facilities (different servers, availability zones or regions). This ensures that the service tolerates up to f physical or virtual machine failures. Under light load, one group of VMs may respond to requests involving more than one of the partitions. As the system load increases, the number of partitions per group decreases down to the minimum of 1. The limit for scalability comes precisely from the number of partitions, which, in turn, is bound by the granularity with which we can divide the data.

To make this architecture cost-friendly, we allow the groups to run in the smallest possible number of physical machines, $2f+1$, plus another machine that serves to order requests, for a total of $2f+2$. As the number of clients grows, we increase the number of groups (or VMs), and take the opposite movement when the number of clients shrinks. This ensures elasticity.

Our main contributions are therefore a BFT architecture that is able to scale with respect to the number of VM groups, thus exploiting the elasticity of the cloud. Virtual machines belonging to the same group are placed on distinct physical machines, to avoid common faults, and the relational data model is supported by locking the database whenever necessary, in order to ensure totally ordered execution of requests.

Resorting to a simple analysis we demonstrate the feasibility of the CloudBFT architecture, by observing that large speedups are achievable even with only a few partitions. Our experimental results support this analysis, showing response times below 1 second for 95% of the queries, and excellent scalability of CloudBFT up to 5 groups. Furthermore, increasing the number of the server groups yields a significant improvement of throughput and latency. In our experiments, we observed that for up to 1 000 transactions per minute, adding more groups to a saturated server always resulted in a considerable increase in the capacity of response and on a reduction in the response times observed by the clients. These results support the feasibility of running critical fault-tolerant servers backed by a traditional OLTP database in the cloud.

We organized the remainder of the paper as follows. In Section II we review the related work. In Section III we present our assumptions. In Section IV we describe the CloudBFT architecture. In Section V we show and discuss the results. In Section VI we conclude the paper.

II. Related Work

Software and hardware faults, as well as malicious attacks, are a challenge for online services, as faulty nodes may deviate from the intended behavior. To tolerate such faulty behavior, one may use state machine replication [9] and allow some replicas to fail while the correct ones ensure that the system as a whole provides correct service to its users. One main concern in a cloud infrastructure is that virtual machine placement is either achieved through bin packing (aiming at maximizing the usage of each physical machine) or through placement algorithms that aim to consolidate virtual machines that co-operate. In both cases, if a group of virtual machines co-operates to provide fault-tolerance, the entire group may be placed on a single physical machine. This creates the possibility for common-mode failures, whereby a single fault could bring down more than f replicas [10], [11].

To avoid common-mode failures, major cloud providers like Microsoft Azure [12] and Amazon Elastic Compute Cloud [13] offer a hierarchy of fault-containment solutions: multi-server, multi-availability zones (AZ) and multi-region. Availability zones share a fast, low latency network, but have separate machines and power sources, whereas regions typically have a significant geographic separation. These forms of separation enable clients to set up disaster recovery or high availability (HA) architectures, with different levels of tolerance and costs. Furthermore, availability zones make it possible to prevent physical co-location. Although this is insufficient to tolerate Byzantine faults, it supports our approach, by providing control over physical replica placement.

Nevertheless, most research concerning virtual machine placement addresses the challenge of improving performance rather than dependability or security. Many approaches explore affinity among virtual machines and postulate that virtual machines that communicate with each other should be placed together [2], [3]. Some authors propose to place together virtual machines with a high degree of page sharing [4] or with memory pages that have exactly the same content [5]. Although these approaches improve application performance, we need to consider the risk of correlated faults, as different replicas running the same algorithm are likely to be placed together. In our system, we try to balance performance and fault tolerance, by joining components that can fail together and separating others.

In the most extreme cases, online services might be exposed to faults that cause nodes to exhibit Byzantine behavior [14]. Such faults may cause a node to behave arbitrarily, and require Byzantine fault-tolerant algorithms, to ensure that the remaining replicas are able to cope with erroneous behavior of the affected nodes. Some approaches introduce Byzantine fault tolerance to state machine replication [15], whereas numerous solutions aim to provide Byzantine fault tolerance to systems in general [16], [15], [17]. Byzantine fault-tolerant protocols resist to f faulty servers with $3f + 1$ replicas.

Agreement-based protocols grew in popularity after the Practical Byzantine Fault Tolerance algorithm [15]. In quorum-based protocols, clients must ensure that two different requests (read or write for example) overlap in at least one server, despite of up to f failures. Under low contention, these algorithms tend to be more efficient, because they skip the quadratic costs of agreement [18], [19]. Zyzzyva [16] tries to avoid this cost, by letting servers speculatively execute commands and postponing commit. A promising path that we explore relies on a Trusted Platform Module (TPM) to sign messages, thus requiring only $2f + 1$ replicas to tolerate f Byzantine faults [20]. This cost is comparable to cloud-based HA solutions, as supported by the providers. We can say the same about communication costs, because HA solutions may involve heavy synchronization throughout different AZ or regions.

Given that we target relational databases, one of the key decisions in our architecture is how we replicate the database. This is a deeply studied topic, including for BFT purposes [21], [22], [23]. Replication mechanisms might be single or multi-master. In the single-master case, the master receives all the updates, whereas the slaves serve either for read purposes or for backup alone, e.g., in primary-backup settings for disaster recovery. In multi-master replication, all the replicas may accept updates. This is the case of the HA settings we see in clouds. Independently from the number of masters, replicas might receive updates either synchronously or asynchronously. In the synchronous case, the updates must reach all the nodes before any further operations (*e.g.*, reads), to ensure ACID properties [24], [25]. This model is difficult to implement, because commit of write operations must follow the same order in all the replicas. In the asynchronous case, distribution of the updates might be lazy, and take some time [26]. As a result, applications may be able to observe inconsistent data. On the positive side, the performance penalty of write operations is typically smaller. In this paper, we implement a simple multi-master approach with synchronous updates. However, to enable parallelism and ensure efficiency, we use a logical division of the data by warehouses.

To prevent common-mode failures, we create groups of virtual machines running on different servers, AZ or even regions. Each group responds to a client request in order to guarantee Byzantine fault tolerance. Given that one major advantage of using cloud computing is the elasticity provided by the infrastructure, we design our approach to provide Byzantine fault tolerance without compromising the elasticity. To achieve this, the number of groups responding to client requests may vary according to the load. Our approach is compatible with the relational data model. Although non-relational storage is becoming popular [27] we believe that security requirements are compatible with the relational model. In other words, our approach does not limit the elasticity of a system, allowing the number of Byzantine fault-tolerant groups to grow according to the load on the system. Furthermore, the resulting performance is close to the theoretical limit imposed by conflicting queries in a relational database.

III. System Model

We assume a Byzantine failure model, where faulty nodes or clients may deviate arbitrarily from the correct state. Clients or nodes that deviate from the correct state are said to be *faulty*. We use a Byzantine fault-tolerant algorithm to ensure that even in the presence of *faulty* nodes, the majority of the system can process a client's request correctly, and reply with the correct response. We use MinBFT [7] as the BFT algorithm, because it needs only $2f+1$ replicated nodes to tolerate f failures.

We assume strong adversaries that can delay communication, attack and take control of any node of the cluster. However, adversaries cannot forge cryptographic operations done by the tamperproof component, such as collision-resistant hashes, encryption and digital signatures. Servers and clients must know the private keys necessary to encrypt and sign the messages, in order to ensure authenticity, integrity, non-repudiation and confidentiality.

We assume an asynchronous network that can fail to deliver messages, duplicate them or deliver them out of order. Like

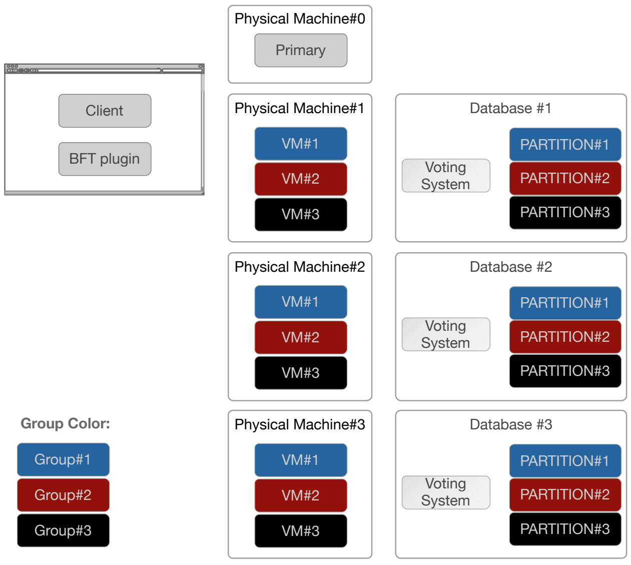

Fig. 1. System architecture.

MinBFT and PBFT, we do not make any assumptions related to liveness, but the network and the internal system must eventually process or send the messages. Furthermore, we assume that faults are independent, *i.e.*, there is no correlation between two failures or the least possible correlation practically achievable. This may be achieved by introducing diversity in the cluster, for example, through the utilization of distinct operating systems, distinct source code, different versions of the programming language used, etc.

Although we use MinBFT as the algorithm to tolerate Byzantine faults, any Byzantine fault tolerant algorithm with some changes, such as PBFT [15] or Zyzzyva [16], is suitable for our system. Thus, our approach may be generally applicable to any Byzantine algorithm.

Since our design is specifically intended for cloud environments, we stipulate that the nodes (VMs) that execute the algorithm must be distributed across $2f+1$ distinct physical machines (in a multi-server or multi-AZ configuration, for example). Thus, if a physical machine fails due to a hardware fault or if multiple VMs are affected by a cross-VM attack, failures will not compromise the majority of the system. In the following section we describe how to achieve this distribution of VMs across physical machines.

IV. CloudBFT

A. Client

Our system model consists of clients accessing an application deployed in the cloud. The application servers that receive and reply to client requests are VMs hosted by a cloud provider. We create groups of VMs such that a separate physical machine (PM) is used for each VM. Virtual machines belonging to distinct groups may share the same PM, as these are never responsible for responding to the same request. We split the application data into separate partitions and associate the VM groups to the different partitions, such that one group

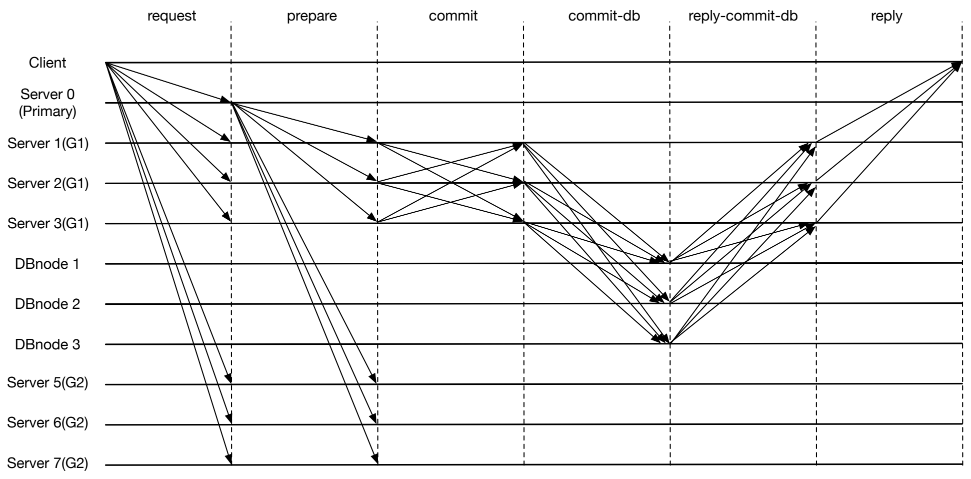

Fig. 2. Normal case execution with 2 groups of size $2f+1$.

may serve one or more partitions, but a partition is never served by more than one group.

We assume that the application has numerous VMs (some of which may execute non-critical functions) and that some of them will be placed in $2f+1$ distinct PMs. This might be ensured through some multi-server or multi-zone agreement with the cloud provider. By using tamper-proof hardware that must be available at each physical node, we are able to identify which VMs are running on which PMs, and create adequate groups using this information. The following subsections describe the protocol in detail.

A client sends requests to the cluster and waits for the resulting response. In our implementation, communication goes through a web service accessible with a special Web browser plugin (see Figure 1). The plugin is a key component, because clients are unable to use the BFT algorithm without it. It manages all clients' requests, ensuring that every request is executed by at least $2f+1$ distinct PMs.

When the client submits a request through the browser, the BFT plugin dispatches it to all elements of the cluster (see Figure 2). All messages sent by the client are signed and encrypted with an *a priori* chosen private key. The BFT plugin will wait for $f+1$ equal responses from the cluster in order to guarantee that the client's request was executed with byzantine fault tolerance.

B. *Primary*

The primary node orchestrates the execution of the BFT algorithm and distributes the work to the groups. After receiving a REQUEST message from a client, it verifies the message integrity and validity. Then, the primary generates a PREPARE message and sends it to the tamper-proof device (as we describe ahead, one such device must be attached to each PM). This device adds a monotonic counter and signs the resulting message, thus preventing the primary from ever sending different versions of the same request.

The primary node selects the Byzantine fault-tolerant group that should process each request, based on the contents of the REQUEST message. The client messages convey information on which partition they refer to (possibly more than one), thus allowing the primary to choose the appropriate group.

Although we assume the general relational data model, data is divided into partitions to enable the system to scale by using parallel execution of BFT groups. As we shall see in the results section, the TPC-C benchmark allows the database to be partitioned, by assigning one group to each warehouse. Requests that require more than one warehouse are handled by serializing all database accesses.

Therefore, the primary node needs to analyze the content of each request to determine which group will process it. Unlike existing approaches, in our design, the primary node does not process any message or response, and is therefore only responsible for ordering requests and sending the signed PREPARE message (see Figure 2). The primary node guarantees totally ordered execution by appending a monotonically increasing counter to all PREPARE messages. One disadvantage is that we need a primary node in addition to all worker VMs, although one primary node co-ordinates multiple groups.

Reducing the load of the primary node is crucial for achieving good system performance. In a realistic scenario, where the system must process a large number of requests, the primary would quickly become the bottleneck if, in addition to signing/ordering messages and issuing PREPARE commands, it would process requests. As the load increases, the system adds more groups and the primary node balances the load.

Since one of our design decisions is to distrust the hypervisors, the primary node must not share the PM with any VM belonging to the groups, *i.e.*, the primary and the groups must be distributed across at least $2f+2$ PMs. Although this requires one additional PM, the cost of the primary becomes less significant as the load increases and the system scales to a larger size.

As soon as the group nodes receive the REQUEST message from a client, these start a timer to detect if the primary is faulty. If the timer expires, because nodes did not get the PREPARE message within the maximum allowable time, they start a view-change operation to elect a new primary. To inform other nodes that it is alive, the primary sends the PREPARE message to all nodes in the cluster, thus canceling their timers accordingly (see Figure 2). A view change is triggered only if the timers of $f+1$ nodes of the same group expire.

C. Groups

The groups are responsible for processing the COMMIT message and for ensuring that the operation is persisted successfully on the database nodes. After receiving the PREPARE message from the primary, the group nodes check if they are responsible for processing the PREPARE message. Since only one group can process a PREPARE message, the others which were not assigned by the primary to process it, will cancel their timers and discard the message (see Figure 2). The chosen group will check the validity and integrity of the PREPARE message by calling a specific function on the TPM for this purpose. If this validation succeeds, the node creates a COMMIT message with a monotonic counter associated. The TPM generates the counter and signs the COMMIT message, ensuring that only another TPM with the same private key will be able to verify successfully the COMMIT message.

As soon as the COMMIT message is created by a group node, it is sent to all others peer nodes of the same group (see Figure 2). The group node will wait for $f+1$ matching COMMIT messages. Each COMMIT message is also verified in the TPM. After receiving $f+1$ matching COMMIT messages, the group member accepts the state and persists it into durable storage. It then creates and sends a COMMIT-DB to all $2f+1$ databases nodes. The COMMIT-DB message contains the operation decided by the group and the order identifier. Each group node signs and encrypts the COMMIT-DB message with an already known key, shared between the database and the group nodes. Each group node needs to wait for $f+1$ matching REPLY-COMMIT-DB messages from the database nodes, guaranteeing the proper processing, even if f database nodes fail. Otherwise, it would not be able to tolerate Byzantine faults in the database nodes and consequently the entire system would be compromised.

Finally, each group node generates the webpage which contains the database response, and then sends it to the client. The page generation could be a heavy stage throughout the system's pipeline, since it needs to parse the database response, generates all data necessary to display the information effectively, such as, HTML, CSS, JavaScript, etc.

Each group has at least one partition associated and most of the time this group is only responsible for executing transactions on this partition (see Figure 1). However, when a transaction needs to access data stored in multiple partitions, a foreign group can access them. As the load increases, the system takes on more groups up to the number of partitions, thus enhancing the computational power and dividing the load across more nodes. The opposite move occurs when the system responds to a lighter load by reducing the size of the cluster. Thus, the system explores the elasticity available in cloud environments, either in response to heavier demands or to save costs during slower periods.

Figure 3 depicts two requests executing in parallel. Two distinct clients send a request to the cluster at the same time. Upon receiving the requests, the primary creates and sends the PREPARE messages (P#1 and P#2) to the groups, in a serial way. After receiving the PREPARE message from the primary, each group starts the COMMIT step. The two groups process the COMMIT messages (C#1 and C#2) in parallel. The database nodes process the COMMIT-DB messages received from the two groups in parallel, since in this example these messages access two distinct partitions. Finally, each group creates the REPLY message in order to fulfil the clients' requests. As the figure shows, different groups can also run this step in parallel.

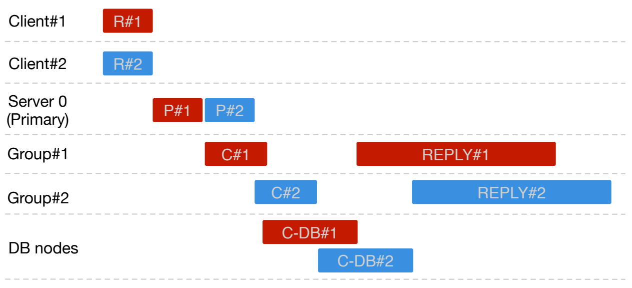

Fig. 3. System architecture.

D. Database nodes

The database nodes are also replicated to ensure the correct operation of the system, even in the presence of faults. To tolerate up to f Byzantine faults, even if hypervisors are not trustworthy, the database nodes must be replicated by $2f+1$ different PMs. To ensure the generality of our system, we do not depart from the relational data model. However, to explore parallelism, we need to split the database into different partitions. A good separation of the database schema may enable different transactions to simultaneously access different partitions, thus being decisive to reduce the contention on the database nodes.

Since the database nodes are physically disconnected from the processing groups, it is vital to ensure that the transactions executed on these nodes were really decided by the groups. The database nodes receive a COMMIT-DB message from the each group node. This message is signed and encrypted by the group node and sent to the database nodes. Although this security mechanism ensures integrity, authenticity, non-repudiation and confidentiality, the database nodes must be sure about which was the correct transaction decided by the group nodes. To circumvent this, every database node must have a voting system for ensuring that only transactions decided by at least $f+1$ group nodes will be able to execute.

As the database nodes are partitioned, we have to consider two types of transactions: *single partition* and *multiple partition*. The former type of transactions only access one partition during the execution, whereas the latter need access to multiple partitions. Distinct *single partition* transactions can execute in parallel, because they will not conflict with one another. However, when a *multiple partition* transaction executes, the voting system (see Figure 1) locks all the database partitions and executes the SQL queries alone in order to guarantee data consistency. One group is associated to at least one database partition. Only that group can execute disjoint transactions (single partition transaction) on that partition. However, a group can execute transactions on several partitions when the operation is not disjoint, requiring access to multiple partitions.

E. Tamper-proof hardware

The purpose of the TPM is to reduce the number of total nodes from $3f+1$ to $2f+1$. The TPM is considered as a tamper-proof component, and therefore, in no scenario could an attacker forge any message signed by the TPM. To reduce the cluster size, in each message produced by the TPM, a monotonic counter is concatenated to the message. As the TPM is a tamper proof component and only the cluster's TPMs know the private key, the signatures ensure that the messages were surely created by the TPM. Furthermore, only another TPM which has the private key will be able to verify the message.

Our system is designed for cloud environments and consequently it is likely multi-tenant, hosting multiple VMs in the same PM. Since we assume that only one TPM will exist per PM, the same TPM must be able to sign and verify operations from distinct group nodes residing in the same PM. Therefore, the TPM must have as many counters as the number of groups in the cluster. Otherwise, the counter sequentiality would be broken, because distinct VMs would increment the same counter. Thus, each group node uses its own counter, thereby guaranteeing proper operation of the system even in a multi-tenant cloud environment.

Each TPM must have a public identifier, known by the other TPMs of the cluster. This identifier enables the nodes running the algorithm to unambiguously determine the source PM of each signed message. Ensuring that messages come from the appropriate process is a more complex problem, but we rely on the same properties and have similar weaknesses as the MinBFT [7] algorithm, concerning access rights to the TPM. In the extreme case where a single process reached the TPM and impersonated nodes from other groups (thus VMs), we would still have a single Byzantine failure, from the point of view of the $2f+1$ relation, as different groups respond to different queries and each group could still conserve a majority of sane nodes. The TPM must provide two functions:

- *createUW(m)* - This function returns a valid unique warrant with a monotonic counter and a TPM identifier concatenated to the message. The monotonic counter ensures the exactly-once and therefore the total order execution. The TPM identifier authenticates the PM creating the warrant.
- *verifyUW(PK,UW,TPM_{id},m)* - This function verifies the validity of the UW certificate previously crated by other TPM. The TPM generates a UW according to the encryption used (RSA [28] or HMAC [29]) by using the private key, PK, the TPM_{id} and the message m. If the UW is equal to the one produced in the TPM, true is returned. Otherwise, the TPM returns false.

Figure 4 depicts the keys used in the system. TPMs must share the same key and all group nodes must have two keys to communicate securely with the client as well as with the database nodes. A session key is enough for guaranteeing a robust communication between client and cluster nodes, but an asymmetric key would be even more robust.

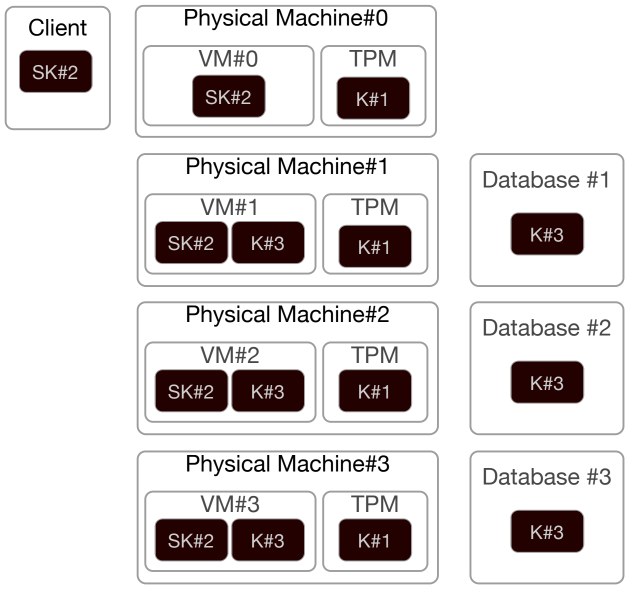

Fig. 4. Keys used in the system.

F. Messages exchanged

To tolerate Byzantine faults in cloud environments, we modify the typical steps made by Byzantine fault-tolerant algorithms, and consequently the content of the messages exchanged during their execution. Almost all messages carry additional information, necessary to ensure proper operation of the system even in a virtualized environment, where one can trust, neither the hypervisor, nor the cloud provider. Before listing these operations, we enumerate their parameters in Table I.

TABLE I. OPERATION LABELS AND THEIR RESPECTIVE MEANINGS.

Label	Meaning
c	Client ID
op	Operation requested by client
t	Timestamp associated to a client request
p_{id}	Primary ID
v	Current view
g_{exec}	Group designated by the primary to process the client request
m	Message containing the client request
$S_{p_{id}}$	TPM sign operation called by the primary
ge_i, ge_j	Group node i and j
S_{ge_i}, S_{ge_j}	TPM sign operation called by the server i and j
o	order id determined by the primary
res	response resulted by the execution of client request

- $\langle \text{REQUEST}, c, op, t\rangle_{\sigma_c}$ — The REQUEST message is sent by the client to all nodes of the cluster, requesting an operation execution. The message must contain the client identifier, the operation to execute and a timestamp.
- $\langle \text{PREPARE}, p_{id}, v, g_{exec}, m, UW_{p_{id}}\rangle$ — The PREPARE message is sent by the primary to all nodes. The message contains the primary identifier, the view number, the identifier of the group which will process the message, the message containing the client request and the unique warrant (UW) created by the TPM.

- $\langle \text{COMMIT}, ge_i, ge_j, v, m, UW_{p_{id}}, UW_{ge_i} \rangle$ — The COMMIT message is broadcast from all elements of a group to all elements of the same group. It must contain the sender (ge_i), the receiver (ge_j), the current view, the message containing the client request, as well as the UW generated by the primary and by the sender.
- $\langle \text{COMMIT-DB}, op, o \rangle_{\sigma_{db_k}}$ — The COMMIT-DB is sent by group nodes to execute operations on database nodes. The message must contain the operations and the order identifier generated by the primary.
- $\langle \text{REPLY-COMMIT-DB}, res \rangle_{\sigma_{ge_i}}$ — The REPLY-COMMIT-DB is sent by the database nodes in reply to the COMMIT-DB message. The message must contain the result of executing the transaction.
- $\langle \text{REPLY}, ge_i, t, res \rangle_{\sigma_{ge_i}}$ — The REPLY message is sent by the the group nodes that processed the request. Therefore, it must contain the sender identifier, the response content and the timestamp sent by the client in the REQUEST message.

V. Experimental Evaluation

To measure the scalability of our system we resorted to experimental evaluation. The CloudBFT clients run in a loop requesting a web page, waiting for the reply and then requesting another page. For each request, the server submits a TPC-C transaction to the database. Upon response from the database, the server generates a web page and replies back. In this setup, TPC-C terminals are replaced by CloudBFT clients sending requests to the cluster. We simulate the page generation step with a sleeping period of 200 ms, since we are not evaluating the client side page rendering. The server must also perform a number of additional operations, like voting, signing and verifying messages using the TPM.

We implemented the CloudBFT system in Java and ran it on a private cluster, under different loads and with varying numbers of VMs. Our cluster runs five Dell PowerEdge R620 rack servers with 4 CPUs and 32 cores, served by a bare metal Xen Hypervisor to support virtualization. The particular configuration of this cluster ensures a strong isolation of the VMs, but gives us little control on the CPU core they use. Thus, there is no physical resource pooling and consequently the system performance is not compromised. Each replica of the group is a single-core Intel Xeon E5405 VM, running at 2.00 GHz , with 1 GB RAM and hosting a Linux 2.6.32 OS.

To enable multiple VMs to access the (single) TPM of their PM, we used a virtualization approach based on TCP sockets. This virtualization scheme accepts requests to sign and verify messages, as we referred in Section IV. The TPM keeps separate monotonic counters for the local VMs, to ensure the sequentiality of these counters for the different VMs. In the verify operation, the TPM compares the expected output against the received data.

To ensure BFT execution of the requests we implemented the BFT plugin using a Java Servlet. This servlet is responsible for handling the clients' requests and dispatching them to all elements of the cluster. Then, it waits for $f + 1$ matching responses from the cluster, before displaying the webpage. The authenticity, integrity, non-repudiation and confidentiality is guaranteed through the utilization of private-public key digital signature and symmetric encryption.

The data tier is composed of $2f + 1$ replicated databases in distinct PMs. We used the MySQL 5.6.17 database management system and the TPC-C as the database schema. The voting system was implemented in Java using the concurrent standard java library to improve performance. The communication with the voting system of the database nodes is done through TPC sockets and all messages are encrypted and digitally signed.

A. *Throughput*

Since our main goal was to build a scalable BFT system for the cloud, capable of responding to increasing loads, we measure the throughput and the latency it obtains against simpler systems and against the theoretical limits. To measure the throughput of CloudBFT, we start by comparing it against a non-replicated non-BFT approach, where each request is processed by a single VM, unlike our own solution, where each request is processed by a group of VMs. In the non-replicated implementation, we varied the number of VMs from 1 to 5. Each of these VMs replies to a different client. Since CloudBFT requires 3 VMs per client, a fair comparison requires 15 VMs, to achieve a comparable number of 5 groups. Figure 5 shows the throughput of the non-replicated solution in grey (the x-axis is the number of VMs) and the throughput of CloudBFT in black (the x-axis is the number of groups), for a system with 15 clients. Since we have 5 partitions (warehouses), the maximum number of partitions per computing group decreases from 5 to 3, 2, 2 and finally 1 partition per group, respectively, for 1, 2, 3, 4 and 5 groups.

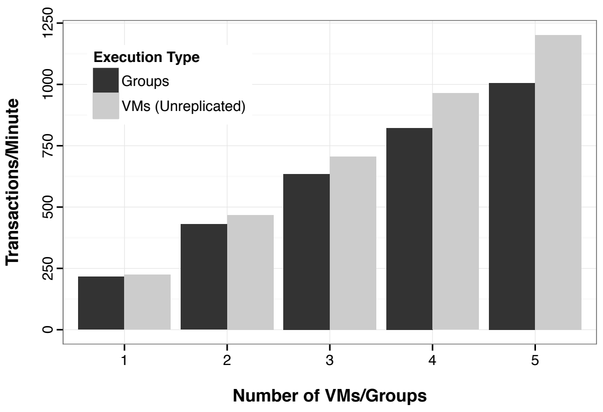

Fig. 5. Throughput with Byzantine fault-tolerant groups *vs.* an unreplicated system.

In these measurements, we can observe that the overhead of our approach ranges between 4%, with 1 group, and 16%, with 5 groups, when compared to 1 VM and 5 VMs of the non-replicated system. The main reasons for this overhead are the additional work performed by all nodes to sign messages, and the way database accesses are performed.

Regarding the work performed when signing messages, the primary node signs each message using the TPM. Each other node must also sign all messages using its local TPM. The primary therefore signs one request at a time. The other nodes are placed in PMs that contain only one TPM. Since different groups may use the same PMs, there is also contention in the local TPM, whenever nodes from different groups simultaneously access this resource.

Regarding the database accesses, CloudBFT locks the entire database externally for requests that must access more than one partition. In the TPC-C benchmark, 10% of the requests are randomly associated to a foreign warehouse, thereby creating write/write conflicts. The non-replicated solution, on the other hand, solves all such conflicts internally using standard transactional isolation. Consequently, the overhead in solving such conflicts is also showing up in the figure.

The relative throughput is also affected by the fact that, at all stages, a BFT group must wait for the slowest node. Since there are normal variations in the response time of each machine, systematically waiting for the slowest node bears an impact on the overhead of our approach.

We measured the amount of time spent at each request processing stage by a group of VMs. Figure 6 shows the time spent by one group of VMs processing requests from a single client. We can observe that generating the HTML of the page is the longest stage, taking about 200 ms. The message signing and verification stages take, combined, 115 ms, and database access takes 71 ms.

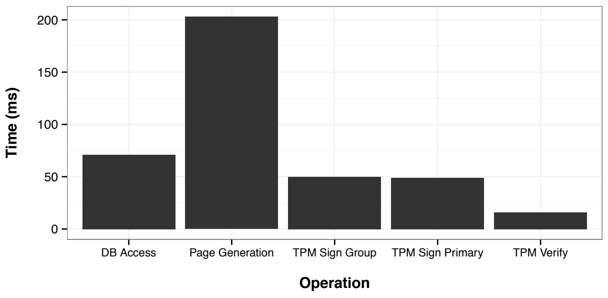

Fig. 6. Time spent at each stage of the BFT pipeline.

Although the HTML page generation stage is the longest, it may be parallelized by adding more groups. Hence, the throughput may be scaled as the number of clients increases, by generating the HTML in parallel groups.

Regarding the load imposed on the primary node, we can observe in Figure 6 that the TPM signature takes 50 ms. This means that the system is limited to a maximum throughput of 1200 transactions per minute. In order to improve this limit, one may use faster tamper-proof hardware or introduce additional modules to sign messages in parallel.

B. Scalability

To evaluate the speedup of our system, we can consider each operation, like signing or verifying a signature, to be a pipeline stage. Then, we consider two distinct cases in the TPC-C benchmark: one out of ten transactions requires a foreign warehouse and the database access must be serialized; the remaining nine out of ten transactions require only the local warehouse and may therefore run in parallel.

In the case of parallelizable requests, only the TPM signature done by the primary is serial, but since the primary could withstand up to 1200 transactions per minute, this effect is not present in these experiments. Hence the speedup bound for nine out of ten transactions should be approximately linear.

In the case of transactions that require foreign warehouses, each group must access more than one partition and, in our approach, the database is locked to serve such transactions. In Figure 6 we can observe that the total time to reply to a request, with one client and one group, is 389 ms. Of this time, the 71 ms corresponding to the database access is the largest time entirely serialized. Therefore, up to the limit of $n = \lfloor 389/71 \rfloor = 5$, we could consider the same coarse-grained linear bound. The overall speedup bound is therefore also linear, i.e., $S(n) = n$, being n the number of processes.

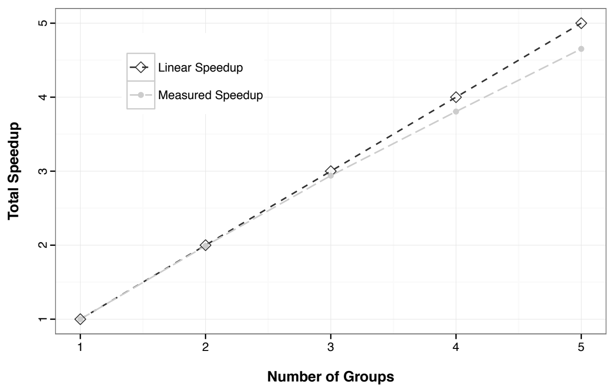

Fig. 7. Total speedup (measured) *vs.* maximum theoretical speedup.

Figure 7 compares the maximum theoretical speedup with the actual speedup, measured with up to 5 groups, using 15 clients. One may observe that our implementation reaches a speedup close to the theoretical maximum. Hence, up to the number of data partitions, the implementation can be said to scale within the limits imposed by the TPC-C benchmark itself. Multiple reasons concur to prevent real solutions from reaching a linear speedup. For example, depending on the load and the number of groups, replicas spend between 2 and 5% of the time waiting on the global lock protecting multi-partition accesses.

It is worthwhile computing the maximum speedup that could ever be achieved as $n \to \infty$. To perform this computation, we assume that 10 serial transactions occur in the beginning and 90 in the end, according to the percentages defined by TPC-C. In steady state, a pipeline where the slowest stage takes 71 ms, can output the 10 requests that occur serially in $10 \times 71 = 710$ ms at least, whereas for 5 partitions, we have the other 90 results in $90/5 \times 71 = 1278$ ms, at least. The system would, therefore, take $710 + 1278 = 1988$ ms for 100 requests. If we compare this to the serial time, which would be $100 \times 389 = 38900$ ms, the maximum speedup we could ever achieve is given by Equation 1. Although this value depends on the time spent in other stages, we can, nevertheless, expect large speedups, quite in excess of the number of partitions.

$$\lim_{n \to \infty} S(n) < \frac{38900}{1988} \approxeq 19.57 \tag{1}$$

C. Elasticity and Latency

We examined the system under an increasing load, to understand how beneficial it can be to add more groups as the number of transactions per minute increases (or the number of clients grows). One of the main advantages of cloud computing is the ability to provision resources on-demand, as the load increases. Our goal is to provide Byzantine fault-tolerance without compromising the elasticity of the cloud.

In Figure 8 we increased the number of clients from 1 to 20 and measured the throughput for systems with up to 5 groups. Each client issues a new request as soon as the preceding reply is received. Due to this, we can observe that a few clients are able to lead the system to its maximum throughput.

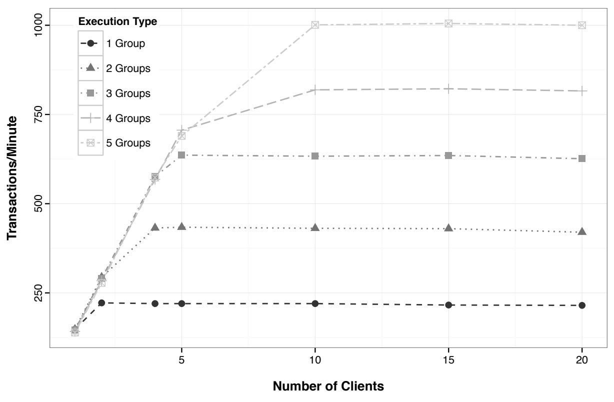

Fig. 8. Throughput for an increasing number of clients, with up to 5 groups.

The results shown in Figure 8 hint to the possibility of making use of the cloud's elasticity to increase the number of Byzantine fault-tolerant groups as the load increases. In other words, one may start the system with one such group and add groups as the load increases. To better analyze this possibility, we measured the latency (in milliseconds) to respond to each request and plotted it against the total load of the system (in transactions per minute). The result is shown in Figure 9.

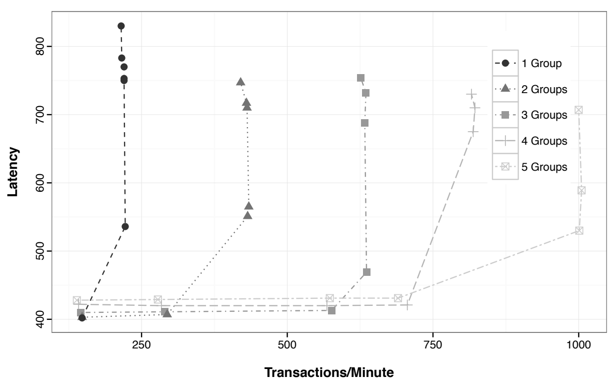

Fig. 9. Latency of requests *vs.* throughput for systems with up to five groups.

In Figure 9 each curve represents a growing number of clients for systems with up to five groups. For instance, the left-most curve shows one Byzantine fault-tolerant group with the number of clients varying between 1 and 20. One may observe that the maximum throughput achieved by one group stabilizes around 200 transactions per minute. At that point the system is saturated and additional requests are queued, leading to increasing latencies.

The cloud's elasticity may be put to use by stipulating a desired maximum latency and determining the number of groups that are necessary to fulfill that requirement under a given load. Figure 9 shows that it is possible to respond to a growing load by elastically adapting the number of fault-tolerant groups.

In fact, the possibility of using more resources to reduce the response time is particularly important, because clients are very sensitive to this parameter [30]. To better understand the response time observed by clients, Figure 10 shows the histogram of latencies, for a system with 11 clients and 5 groups. This configuration was chosen to measure the 5 groups in a high load scenario of 1 000 transactions per minute.

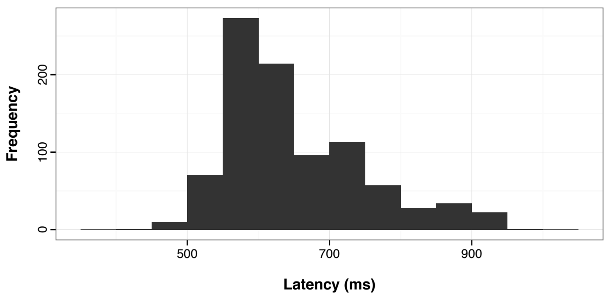

Fig. 10. Distribution of latencies.

We can observe that most requests are replied with a latency around the average of 650 ms. Nevertheless, some requests take 50% above that average. In order to choose an adequate elasticity plan (i.e., choose the number of active groups at each point in time) it is suitable to observe the cumulative distribution function, plotted in Figure 11.

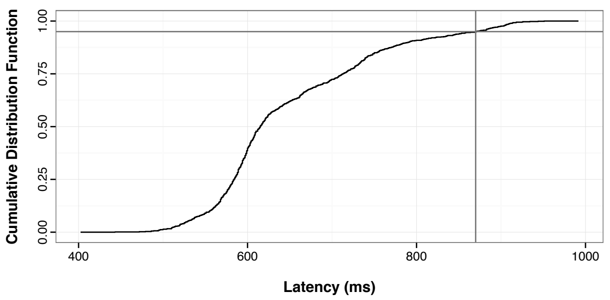

Fig. 11. Cumulative distribution function of latencies.

The cumulative distribution function of latencies, in Figure 11, shows that 95% of the requests are replied within 870 ms (the plotted guidelines). This analysis should be performed online, by using a specified service requirement. For example, if the requirement for an application specifies that the 95th percentile of the clients must receive replies within 800 ms, one would need to startup at least one more group.

VI. Conclusion

This paper described a Byzantine fault-tolerant architecture, designed for cloud applications with critical services that may scale with the number of clients. We create groups of $2f+1$ virtual machines running on distinct physical machines, in order to ensure that a single fault is unable to affect multiple virtual machines belonging to the same group.

The proposed design supports the relational data model. Although this model is well known and frequently used, one must address the fact that two or more groups of virtual machines may require access to the same data items (*i.e.*, the data may not be completely partitioned). Hence, it requires synchronization among different replicas to guarantee totally ordered accesses to every data item.

Using the TPC-C benchmark, our results show that, within reasonably large bounds, elasticity is achievable for cloud-based BFT protocols even under the relational data model. We believe that this may help a wide spectrum of critical services intended to be deployed in the cloud. Further experiments, using larger clusters, will provide a better understanding of the impact on speedup of the number of database partitions, or the percentage of non-parallelizable queries in TPC-C.

Acknowledgment

This work has been supported by the FCT, Fundação para a Ciência e a Tecnologia, in the scope of Programa Operacional Temático Factores de Competitividade (COMPETE) and Fundo Comunitário Europeu FEDER, through project DECAF, An Exploratory Study of Distributed Cloud Application Failures (EXPL/EEI-ESS/2542/2013).

References

[1] K. Birman, G. Chockler, and R. van Renesse, “Toward a cloud computing research agenda,” *SIGACT News*, vol. 40, no. 2, pp. 68–80, 2009.

[2] X. Meng, C. Isci, J. Kephart, L. Zhang, E. Bouillet, and D. Pendarakis, “Efficient resource provisioning in compute clouds via VM multiplexing,” in *Proc. of the 7th International Conference on Autonomic Computing*, ser. ICAC’10. ACM, 2010, pp. 11–20.

[3] J. D. Sonnek, J. B. S. G. Greensky, R. Reutiman, and A. Chandra, “Starling: Minimizing communication overhead in virtualized computing platforms using decentralized affinity-aware migration,” in *ICPP*. IEEE Computer Society, 2010, pp. 228–237.

[4] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. D. Corner, “Memory buddies: Exploiting page sharing for smart colocation in virtualized data centers,” in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE ’09. ACM, 2009, pp. 31–40.

[5] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat, “Difference engine: harnessing memory redundancy in virtual machines,” *Communications of the ACM*, vol. 53, no. 10, pp. 85–93, 2010.

[6] Transaction Processing Council (TPC), *TPC Benchmark C Standard Specification, Revision 5.11*. 777 North First St., Suite 600, San Jose, CA 95112: Transaction Processing Council, Feb. 2010.

[7] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Veríssimo, “Efficient byzantine fault-tolerance,” *IEEE Trans. Computers*, vol. 62, no. 1, pp. 16–30, 2013.

[8] Trusted Computing Group, “TPM main specification,” Trusted Computing Group, Main Specification Version 1.2 rev. 85, Feb. 2005. [Online]. Available: http://www.trustedcomputinggroup.org

[9] F. B. Schneider, “Implementing fault-tolerant services using the state machine approach: A tutorial,” *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[10] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch, “Subvirt: Implementing malware with virtual machines,” in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2006, pp. 314–327. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/SP.2006.38

[11] IBM Corporation, “IBM X-Force 2010Mid-Year Trend and Risk Report,” http://public.dhe.ibm.com/common/ssi/ecm/en/wgl03003usen/WGL03003USEN.PDF.

[12] “Azure: Microsoft’s cloud platform — cloud hosting — cloud services,” http://azure.microsoft.com/en-us/, accessed on April 30^{th}, 2014.

[13] “Aws — amazon elastic compute cloud (ec2) - scalable cloud hosting,” http://aws.amazon.com/ec2/, accessed on April 30^{th}, 2014.

[14] Verizon, “Data breach investigations report,” 2013.

[15] M. Castro and B. Liskov, “Practical byzantine fault tolerance,” in *Proceedings of the Symposium on Operating System Design and Implementation (OSDI)*, Feb. 1999.

[16] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. L. Wong, “Zyzzyva: Speculative byzantine fault tolerance,” *ACM Trans. Comput. Syst*, vol. 27, no. 4, 2009.

[17] T. Wood, R. Singh, A. Venkataramani, P. J. Shenoy, and E. Cecchet, “ZZ and the art of practical BFT execution,” in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys ’11, C. M. Kirsch and G. Heiser, Eds. ACM, 2011, pp. 123–138.

[18] D. Malkhi and M. Reiter, “Byzantine quorum systems,” in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, ser. STOC ’97. New York, NY, USA: ACM, 1997, pp. 569–578.

[19] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, “Fault-scalable byzantine fault-tolerant services,” *SIGOPS Oper. Syst. Rev.*, vol. 39, no. 5, pp. 59–74, Oct. 2005.

[20] M. Correia, G. S. Veronese, and L. C. Lung, “Asynchronous byzantine consensus with 2f+1 processes,” in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC ’10. New York, NY, USA: ACM, 2010, pp. 475–480.

[21] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden, “Tolerating Byzantine Faults in Transaction Processing Systems Using Commit Barrier Scheduling,” in *ACM SOSP*, Stevenson, WA, October 2007.

[22] R. Garcia, R. Rodrigues, and N. Preguiça, “Efficient middleware for byzantine fault tolerant database replication,” in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys ’11. New York, NY, USA: ACM, 2011, pp. 107–122.

[23] S. Elnikety, S. Dropsho, and F. Pedone, “Tashkent: Uniting durability with transaction ordering for high-performance scalable database replication,” in *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, ser. EuroSys ’06. New York, NY, USA: ACM, 2006, pp. 117–130.

[24] I. Akturk, *Replication of Metadata in Distributed Storage Systems: Asynchronous Replication Across Multi-Master Servers*. Germany: LAP Lambert Academic Publishing, 2011.

[25] S. Elnikety, S. Dropsho, E. Cecchet, and W. Zwaenepoel, “Predicting replicated database scalability from standalone database profiling,” in *Proceedings of the 4th ACM European Conference on Computer Systems*, ser. EuroSys ’09. ACM, 2009, pp. 303–316.

[26] S. Almeida, J. a. Leitão, and L. Rodrigues, “Chainreaction: A causal+ consistent datastore based on chain replication,” in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys ’13. New York, NY, USA: ACM, 2013, pp. 85–98.

[27] K. Grolinger, W. Higashino, A. Tiwari, and M. Capretz, “Data management in cloud environments: Nosql and newsql data stores,” *Journal of Cloud Computing: Advances, Systems and Applications*, vol. 22, no. 2, 2013.

[28] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978.

[29] H. Krawczyk, M. Bellare, and R. Canetti, “Hmac: Keyed-hashing for message authentication,” United States, 1997.

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