· U C ·

UNIVERSITY OF COIMBRA

DEPARTMENT OF INFORMATICS ENGINEERING
FACULTY OF SCIENCES AND TECHNOLOGY

MASTER THESIS

CLOUDBFT: ELASTIC BYZANTINE FAULT-TOLERANT WEB SERVER

Rodrigo Nogueira
ran@dei.uc.pt

Supervisors:
Raul Barbosa
Filipe Araújo

July 1, 2014

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 dependability and performance. Nevertheless, the migration to cloud environments is a process that has been raising some concerns to many companies, which see the lack of physical control, the physical resources sharing (between distinct clients) and possible security breaches as the biggest barrier to move their systems to cloud environments.

Taking into account these cloud's problems and the resilience, availability and consistency needed in critical applications, we propose CloudBFT: a standard three-tiered system capable of taking advantage of cloud's scalability and elasticity, and simultaneously, being as resilient as possible in order to tolerate a wide range of faults, such as faults caused by intrusions, software and hardware faults, etc. The elasticity and scalability are achieved by scaling out and shrinking the cluster according to the processing requirements. On the other hand, to tolerate a wide range of faults (i.e., Byzantine faults), the system must execute a parallel version of a Byzantine fault-tolerant algorithm, where it is used groups of replicas placed on distinct physical machines, as a means to avoid exposing applications to correlated failures. This challenge becomes even more difficult in a relational model (as we used), where the synchronization and contention is higher.

We believe that the elasticity we observe in our system, as it scales with the load, demonstrates the feasibility of tolerating Byzantine faults in a cloud-based web server using a relational data model. The results show that the system can scale with the load, as well as tolerating Byzantine faults in a cloud-based web server using a relational data model. Therefore, this work indicates that is possible to have a higher level of resilience in cloud environments and tolerating Byzantine faults without compromising the scalability and elasticity.

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

Acknowledgments

Foremost, I would also like to express my sincere gratitude to my advisors Prof. Filipe Araujo and Prof. Raul Barbosa for the continuous support of my M.Sc study and research. They gave me the correct quantity of independence to follow my own research topics, but at the same time, guiding and advising me for following the better way, even when that way seemed too difficult and distant. I would like to thank my family, particularly my parents and my sister for all they have done for me, for supporting me in all decisions, for encouraging me to always achieve more and better. I would like to thank my friends and colleagues that helped me during the last five years of my M.Sc. Without them this path would have been much more complicated.

This work has been partially supported by the project CMU-PT/RNQ/0015/2009 - TRONE (Trustworthy and Resilient Operations in a Network Environment) and by the project DECAF: An Exploratory Study of Distributed Cloud Application Failures (reference EXPL/EEI-ESS/2542/2013). Both projects are funded by the "Fundação para a Ciência e a Tecnologia" (Portuguese Government).

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Motivation

Cloud computing has been consolidating as one of the most successful paradigms of the internet. The need to enhance the processing power and to accommodate peaks in demand and, at the same time, reduce the costs with the infrastructure has led many companies to move their information systems to cloud environments. The elasticity is one of the most important features of the cloud, where the cluster size can grow or shrink, by adding or removing Virtual Machines (VMs) according to the processing power needed at a certain time.

Cloud providers use virtualization for reducing the overall costs with their infrastructures by grouping VMs into the lowest possible number of physical machines (PMs). The costs are reduced because the lower the number of physical machines used is, the less the power consumed by the cluster will be. Using virtualization allows cloud providers to offer services to distinct clients with lowest possible number of PMs, since each client rents VMs instead of PMs. However, the resource sharing might deteriorate the security, since different clients share the same PMs and malicious clients may managed to attack other VMs belonging to the same PM. In addition, clients are only able to access VMs remotely, thereby removing the physical control over these infrastructures and, therefore, increasing the possibility of a physical attack.

The consistency and availability in many applications are crucial, mainly in critical applications that have an utmost need of guaranteeing that the service has enough resilience to work properly without interruption regardless of time or day. Banks are an example, since they must ensure that all transactions are processed properly, otherwise they may lose money, clients may start to complaint about the quality of the service, etc. These guarantees are not provided by cloud providers and because of that, clients that have critical applications are still reluctant about moving their critical system to the cloud.

Cloud infrastructures have been exhibiting a wide range of faults [1], such as hardware faults, software faults, faults caused by an arbitrary behavior of network communication and faults caused by attacks. The Byzantine faults are a result of an arbitrary behavior of the system and might be originated by a wide range of sources, such as attacks, software and hardware faults. For instance, Byzantine faults may be caused by crashes, omissions, timing faults, processing faults, faults in Redundant Array of Independent Disks (RAID) controllers, etc. Systems able to tolerate Byzantine faults are known as Byzantine fault-tolerant and they are becoming increasingly important, since malicious attacks, and software and hardware faults have also become more common.

Building an application capable of taking advantage of cloud's elasticity and scalability is not an easy task, since these applications must be built in accordance to the cloud's computational model. To overcome this barrier, cloud providers have encouraged, through pricing, the use of their non-relational data storages, to enable parallel, unsynchronized accesses.

Unfortunately, these models do not provide all the ACID (Atomicity, Consistency, Isolation and Durability) properties of typical relational databases. 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 algorithm is expensive due to all the synchronization necessary for replicated nodes and databases. In fact, some researchers argue that the subtleties of replication strongly collide with elasticity, a driving force of the cloud [2]. 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 [3, 4, 5, 6].

Currently, there are no cloud service capable to tolerate byzantine faults. Even the main cloud providers, such as Amazon EC2 [7], Rackspace [8], Google Cloud Platform [9] and Microsoft Azure [10] do not have any type of service capable to tolerate byzantine faults, where the security and reliability are deteriorated in favor of the performance and elasticity. A service capable of taking advantage of cloud elasticity by scaling out and shrinking the cluster according to the processing requirements and, simultaneously, capable of tolerating Byzantine faults (intrusions, hardware faults, processing errors, malicious physical access, etc.) would be a great asset for cloud providers and it could attract a larger number of customers.

The ideal system for cloud, mainly for critical applications, is a system capable of taking advantage of the scalability and elasticity (to accommodate peaks in demand) and, at the same time, being as resilient as possible for tolerating byzantine faults. Ensuring all these characteristics at the same time, without harming the latency and throughput is a tremendous challenge and surely a system that has all these features could be applied in practical scenarios.

1.2 Goals

The main goal of this work is to demonstrate that is possible to have a system capable of taking advantage of cloud elasticity for adapting the number of nodes used according to processing power required for processing the clients' requests. In addition, the architecture must be able to tolerate Byzantine faults, thereby ensuring that the system does not deviate from the expected behavior even if a set of nodes has been attacked or a PM has been affected by a hardware fault (compromising the whole PM) or a VM is compromised due to a hardware or software fault, etc. The architecture must be as general as possible, being adaptable to any BFT algorithm with minimal changes. It provides a greater flexibility and enables the user to choose the BFT algorithm that it knows best and relies on the most.

The system must be able to execute on a relational database, guaranteeing both data consistency and scalability by partitioning the database with the lowest data correlation possible. To deal with the different partitions, it must create processing groups (replicas) running on different facilities (different servers, availability zones or regions). The number of partitions may vary according to the load over the system, *i.e.*, 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.

Since this architecture is designed for web applications, like in most of known web frameworks, it must be as decoupled as possible for enabling the programmer for producing modular and maintainable code. Furthermore, to make this architecture cost-friendly, the processing groups must run in the smallest possible number of PMs (it depends on which BFT algorithms the user has chosen), thus taking advantage of virtualization. As the number of clients grows, the architecture increases the number of groups (or VMs), and takes the opposite movement when the number of clients shrinks. This ensures elasticity.

1.3 Contributions

This work provides the following contributions:

- a scalable BFT system that shrink and scales out according to the processing power required for processing the clients' requests;
- it is the first three-tier and cloud-based architecture for tolerating Byzantine faults;
- it is the first BFT system that was demonstrated that is capable to ensure some ACID properties using relational databases in cloud environments;
- it achieves a $10\times$ speedup.

This work led to the following publication: Rodrigo Nogueira, Filipe Araújo, Raul Barbosa, "CloudBFT: Elastic Byzantine Fault Tolerance", to appear in 20th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2014), November 2014.

1.4 Structure

The remaining document is structured as follows. Chapter two introduces the state of the art and the related work, focusing on the cloud computing architecture and byzantine fault-tolerant algorithms. Chapter three presents the CloudBFT architecture. Chapter four describes a CloudBFT implementation using an algorithm that requires $3f + 1$ nodes. Chapter five shows and discusses the results achieved. Chapter six concludes the work.

CHAPTER 2

Related Work

This chapter describes the state of the art and the related work, focusing in two main topics of our investigation: cloud computing and Byzantine fault-tolerant algorithms. Section one describes the main characteristics of cloud computing, its architecture and functioning, as well as its service model. Section two characterizes the fault model, their sources and consequences. Section three discusses the methods to tolerate byzantine faults and depicts the functioning of some algorithms in the literature. Section four compares the Byzantine fault-tolerant algorithms presented in the previous section. Section five details the functioning of the Trusted Platform Module and their features. Section six describes the main benchmarks found in the market.

2.1 Cloud Computing

Cloud computing is a computational model that is dynamically scalable (elastic), easily configurable, of rapid deployment, convenient, virtualized and ubiquitous [11]. According to Foster et. al. [12], cloud computing can be defined as:

"A large-scale distributed computing paradigm that is driven by economies of scale, in which a pool of abstracted, virtualized, dynamically-scalable, managed computing power, storage, platforms, and services are delivered on demand to external customers over the Internet."

2.1.1 Architecture

Virtualization is a crucial part of the cloud computing architecture, since it allows that physical machines with a huge amount of memory and dozens of cores to share their resources between distinct Virtual Machines (VMs). The resource sharing guarantees the isolation between VMs and, therefore, one VM cannot access the resource previously allocated for others. Furthermore, if a VM has been compromised, the other VMs are not automatically affected, but the risk of attack increases and, consequently the risk of the whole physical machines being compromised increases as well. According to the National Institute of Standards and Technology (NIST) [13], the five key cloud's characteristics are

On-demand Service - The customer can configure the service to ensure that the resource management is done transparently and automatically (without directly interacting with the cloud provider). The management is done based on metrics such as disk utilization, memory and CPU used, etc.

Resource Pooling - It allows different customers to share the same software (multi-tenant model), the same operating system (OS) and the same hardware. The resources are dynamically assigned and are adapted according to the customer's needs (more or less processing power, more or less disk space, etc.). The disks are partitioned, giving the illusion that the service is running alone.

Rapid Elasticity - The cluster's computational power can be adjusted quickly and efficiently, conforming to the processing requirements needed at certain time: adding, removing and migrating VMs.

Broad network Access - All resource can be accessed remotely, allowing clients, both thin and thick, to be able to access the service.

Measured Service - The cloud provider automatically and transparently controls and monitors the client's resources. In addition to the fixed price that is charged by the rent of the resources (VMs), the client is also charged conforming utilization of these resources (network traffic, CPU processing, etc.).

Virtualization has a direct influence on most of the cloud's characteristics referenced above. Virtualization, considered a key aspect of the cloud services, allows cloud providers to increase the resource utilization of the cluster, minimizing power consumption (more VMs sharing the same physical machine) and, consequently, reducing the maintenance cost of the cluster.

The cloud environments must have an additional layer for managing the VMs. The hypervisor or Virtual Machine Monitor (VMM) (see figure 2.1) is a component which, in cloud environments, runs over the hardware and is responsible for creating, destroying and managing the execution of all VMs. It is the hypervisor which controls

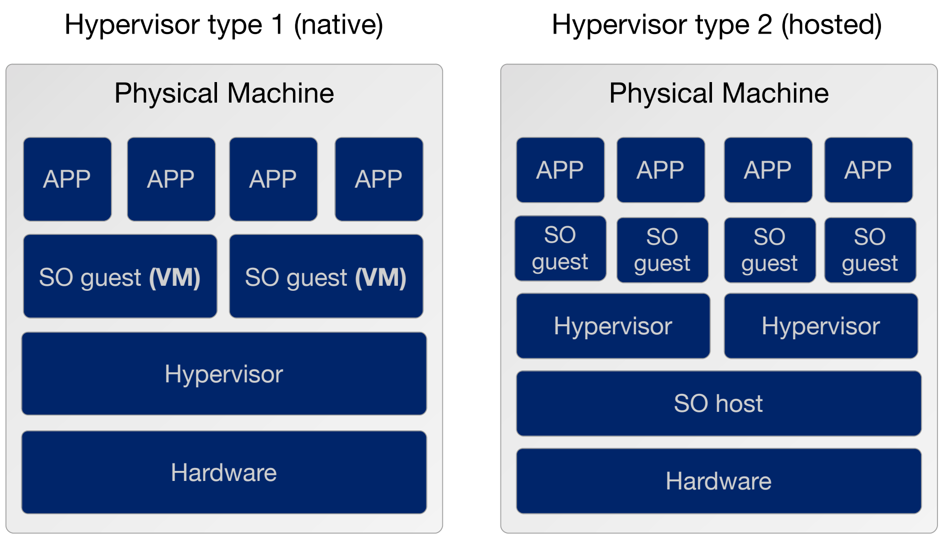


Figure 2.1: Virtualized Architecture and distinct types of Hypervisors

the amount of memory that can be used by the OS guest, the disk partition to store the data, the network adapter for communication, etc. Guaranteeing the isolation between the OS is one of the most important hypervisor roles, since it is essential for guaranteeing the fully isolation between distinct VMs, ensuring that distinct VMs do not conflict during concurrent accesses to resources of the same physical machine (PM). Hence, each VM works as if it were executing over an independent PM.

There are two types of hypervisors (see figure 2.1): bare metal (type 1), that runs directly over the hardware and the hosted (type 2), that works as an OS application, connecting the OS guest with the hardware, through the OS host.

Studies [14] [15] [16] show that the overhead of sharing physical resources between distinct VMs and the addition of the hypervisor for managing the access to this resources is low, especially when the bare metal hypervisor is used, thus guaranteeing a good performance and at the same time maximizing the physical resource utilization. Nonetheless, other studies [17] [18] show that the hypervisor is not 100% secure, and it has been a target of several successful attacks, which managed to break the isolation between VMs, thereby executing a cross-VM attack (see Figure 2.2). According to the IBM X-Force [18], 35% of the vulnerabilities created by the virtualization may compromise the hypervisor, as well as the isolation between the VMs. The bare metal hypervisors (type 1) are considered more secure because they execute directly over the hardware (without any host OS). On the the other hand, adding a hypervisor on top of a host OS (hosted hypervisor) adds more complexity and more vulnerabilities to the host [19]. The hypervisor must be as robust and secure as possible because the proper functioning of VMs depends on this.

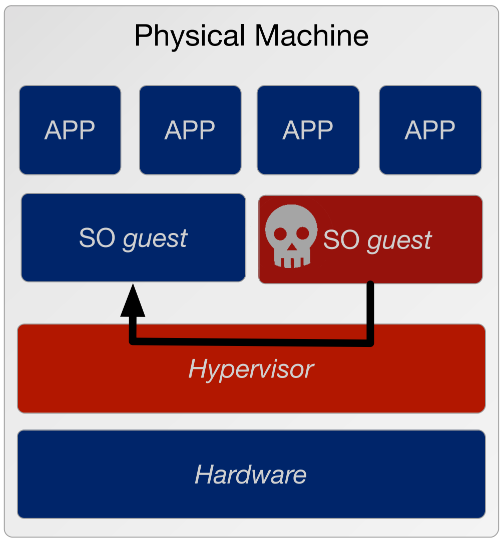


Figure 2.2: Example of a cross-VM attack, where the hypervisor cannot ensure the isolation between VMs.

2.1.2 Service Models

Cloud computing has different service models, which vary according to the abstraction level intended by the customer. The higher the abstraction level is, the lower the access level and infrastructure control will be (see figure 2.3). The service models are:

Infrastructure as a Service (IaaS) - In this model, the cloud provider offers servers, storage, network, and OSes according to the customer needs (on-demand). The restrictions imposed by the IaaS providers are minimal, since the customer can control and configure almost all components over the infrastructure, thereby being able to deploy or execute any type of software, such as applications or operating systems. The user is billed only by the CPU processing, data transferred over the network (download and upload) and disk space used. The infrastructure can scale out or shrink dynamically according to resources that are necessary, thereby reducing the cost when the load is low and guaranteeing computational power when needed. The IaaS big players are Amazon EC2 [7], Rackspace [8], Microsoft Azure [10] and Google Cloud Platform[9].

Platform as a Service (PaaS) - In this model, the cloud provider offers a high level ready-to-use development platform for the client for compiling, testing and deploying its own applications. To provide these services, the PaaS provider imposes some restrictions on the consumer and limits some characteristics, such as supported programming language, available libraries and services, the disk access mechanism, etc. The customer does not have access or control of infrastructure resources (like in IaaS), but the cloud providers ensure that the platform where a customer's application is running, fulfills the needed requirements. The PaaS big players are: Microsoft Azure Services [10], Force.com platform [20], Heroku [21] and Google App Engine [22].

Software as a Service (SaaS) - In this model, the cloud provider is responsible for managing the complete infrastructure and, therefore, ensuring that the service works properly without any downtime or at least as minimal as possible. This involves guaranteeing the appropriate infrastructure operations (IaaS) and properly deploying and operating the customers' applications (PaaS), as well as many crucial features such as scalability, reliability, compatibility, confidentiality, etc. The client just uses the software and does not need to worry about server configuration, OS, network, storage. The access is done by a browser or by a thin client designed especially for this purpose. The SaaS providers charge the customers conforming to the service utilization, for example, subscriptions for a certain period (monthly, annually, etc.). The SaaS big players are: Salesforce.com [23], leading provider of Customer Relationship Management (CRM) online and Google Apps [24] which provides e-mail, calendar, document, and management for business and Workday [25].

2.1.3 Deployment Models

The cloud computing has two key deployment models, which define how the client accesses the resources. One is the public cloud that is the most popular and widely used, where the whole infrastructure and their resources are provided as a service that can be accessed over the Internet. In this model, the cluster's resources are shared by distinct customers (multi-tenancy), and the customers are charged depending on service utilization. The infrastructure is managed by the cloud provider, but the customer can automatically adapt the resources in a certain period, in order

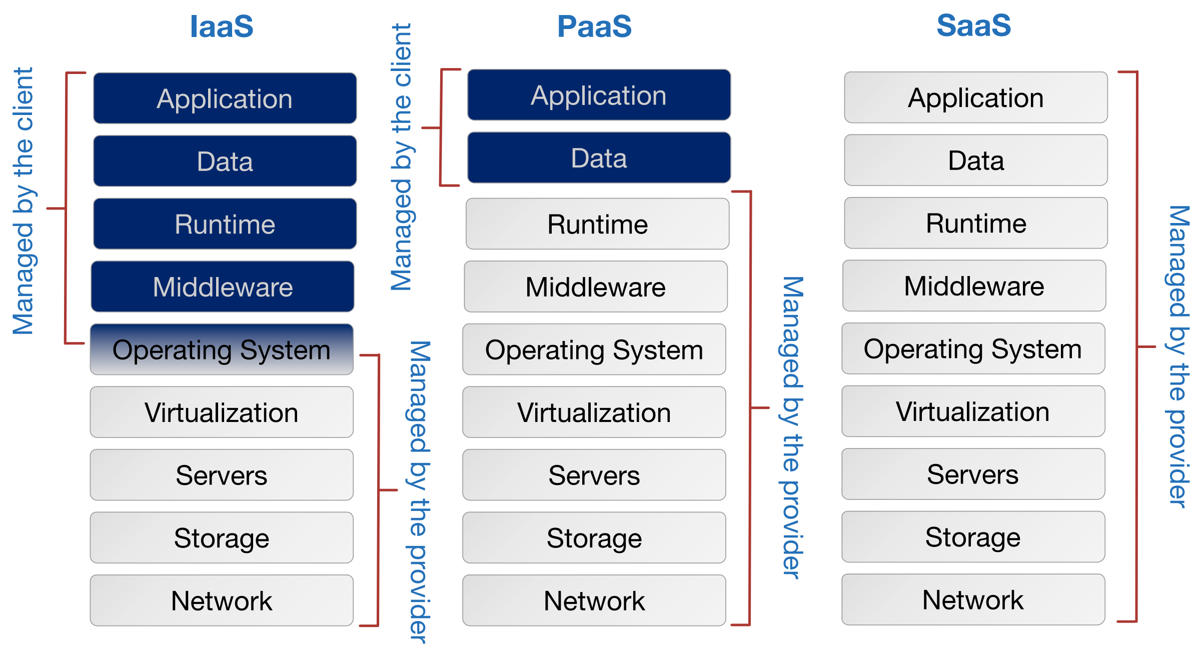


Figure 2.3: Cloud service models: IaaS, PaaS and SaaS

to satisfy their needs, such as processing capacity, space in disk, etc. The main public cloud providers are Amazon [7], Windows Azure [10], Rackspace [8] and Google Cloud Platform [9]. The private cloud is another deployment model, where the whole infrastructure is used by only one organization and there is no kind of hardware sharing between customers (through virtualization), thus reducing the risk of a possible cross-VM attack, since all VMs belong to the same owner. The resource management can be done by the owner, by other companies or by both.

Besides the private and public cloud deployment models, there are two others that are based on these models. The hybrid cloud allows an organization to share their resources between a public cloud and a private cloud, thereby aiming to do as much processing as possible in private cloud and only use the public cloud when the private cloud does not suit to the requirements needed. The community cloud deployment model allows customers to share their resources among a restricted set of organizations, which usually have the same purposes, such as security, interoperability, confidentiality, etc.

2.2 Fault Model

Faults, errors and failures are terms used in the field of dependability, with different meanings, often exchanged erroneously. These terms are extremely important for understanding the following sections well. The terminology used is based on Gray et. al. [26], Laprie [27], and endorsed by IFIP Working Group (IFIP WG 10.4) and the IEEE Technical Committee on Fault-tolerant Computing.

A system can be composed by one or multiple modules that can recursively have sub-modules. Each system has a *specified behavior*, which is the behavior expected in the absence of anomalies, and the *actual behavior*, which is the current state that the system is in.

A fault creates one or more *latent errors*, becoming effective when it is activated. When the error is activated, it might create more errors, that in turn may affect the service, thereby generating a failure (see Figure 2.4), and therefore, affecting proper functioning of the system and exhibiting that to the client. Therefore, failures happen when the actual behavior diverges from the specified behavior, errors are defects in a module and the cause of an error is a fault.

Faults can be caused by physical components (hardware faults), such as a hard drive disk, memory, processor, network interface, as well as by software faults, such as programming mistakes, exceptions, etc. A fault generates a latent error, which, might never be activated, for example, a piece of software containing a programming mistake that is never executed. The errors emerge when, because of a fault, a module no longer works properly, thereby making part of the system or making the whole system inconsistent. Errors may generate a failure, that in turn affects the proper execution of the system, where data may be corrupted or lost, processing errors may occur, hardware may not work properly and, consequently, exhibiting the erroneous behavior to the client.

One of the main goals of this work, is to ensure that errors do not result in failures,*i.e.*, the system must be able to tolerate faults, errors and failures, working properly even in these scenarios.

Although software faults occur more often, hardware faults have a greater impact on the system, mainly if the system uses virtualization, because all VMs belonging to the PM will be affected. For instance, a memory fault may compromise the proper

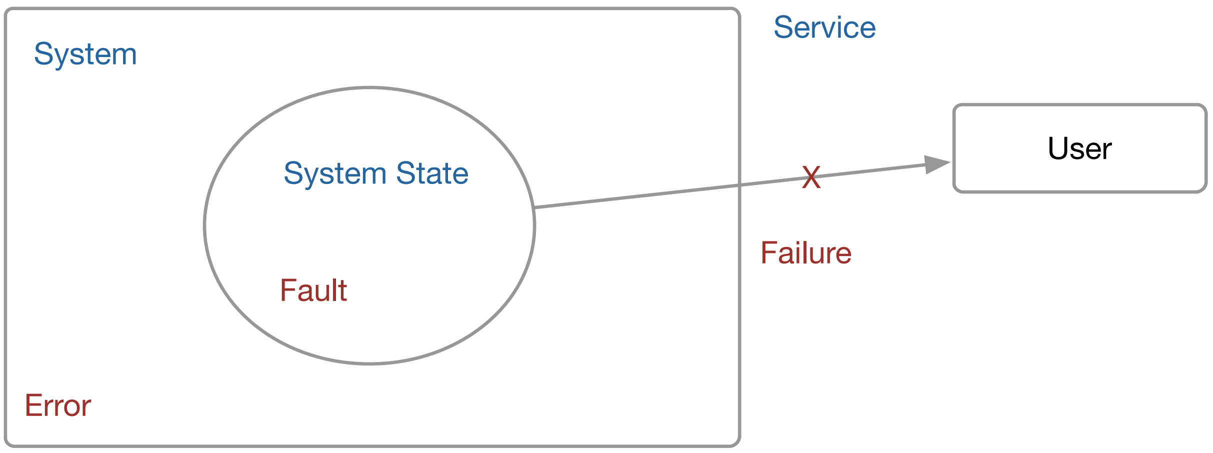


Figure 2.4: The representation of the flow of faults, errors and failures.

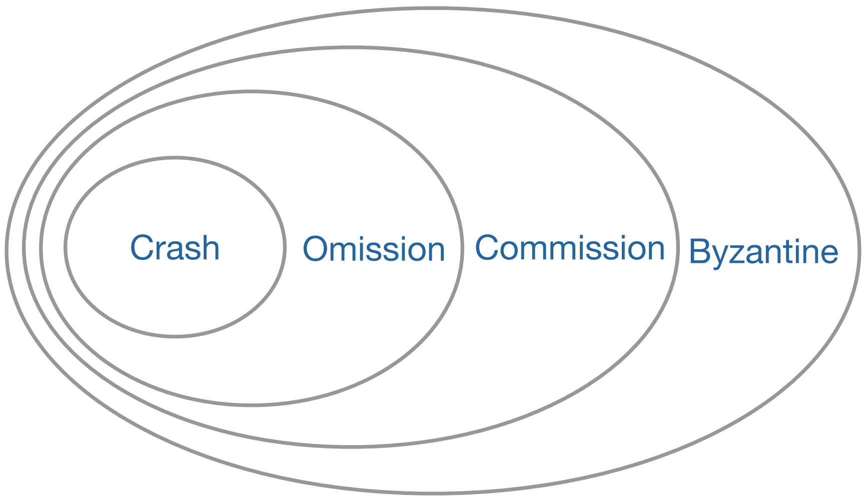


Figure 2.5: Spectrum of faults representing crash, omission, timing and Byzantine faults

functioning of the whole system, since for the correct functioning of the software, the hardware must work well too. The disk and processor faults can have the same effect, since they are crucial components for the correct working of any OS, and consequently, any application.

In cloud environments, tolerating hardware faults is even more important, since the cloud architecture is based on virtualization and, therefore, a PM provides VMs for several clients. Thus, a hardware fault may affect all VMs belonging to the PM, precluding the clients from accessing their VMs or corrupting data stored in these VMs. Cloud providers, such as Amazon, offer High Availability Zones (Multi-AZ), where the service and database are replicated and automatically synchronized across different availability zones. Although the replication achieved by Multi-AZ enhances the system's availability, since the service is replicated across distinct PMs around the world, this is not enough for ensuring the proper functioning of the system in all scenarios, because the replicated nodes may deviate arbitrarily from the specified behavior, thereby being able for corrupting all other replicated nodes. In critical systems, it is necessary a higher level of dependability than the achieved by a typical replicated system, since critical system must ensure that even arbitrary faults do not compromise the service.

The replication achieved by Multi-AZ guarantees that the system is able to tolerate crash faults and omission faults (*e.g.*, failing to receive a request or failing to send a response) (See Figure 2.5, however there are more complex faults, such as commission faults (*e.g.* process the request erroneously, corrupting the system state, sending inconsistent responses to a request) and Byzantine faults. Byzantine faults (see Figure 2.6) are the most complex faults, since the system may deviate from the specified behavior arbitrarily and, therefore, it is extremely difficult to predict the system's behavior and the consequences of the erroneous actions. A system that is capable of tolerating Byzantine faults is known as Byzantine fault-tolerant (BFT). In addition to software faults and hardware faults, the BFT systems must also tolerate physical or remote attacks. Therefore, they must guarantee that if an attacker performs a successful attack, a set of replicated nodes remain following the specified behavior, thereby guaranteeing the correct service functioning, even if a set of nodes has been exhibiting arbitrary behavior. The BFT systems do not make any assumptions about the source of the faults, since they must be able to tolerate faults from any source, including they are hardware or software faults.

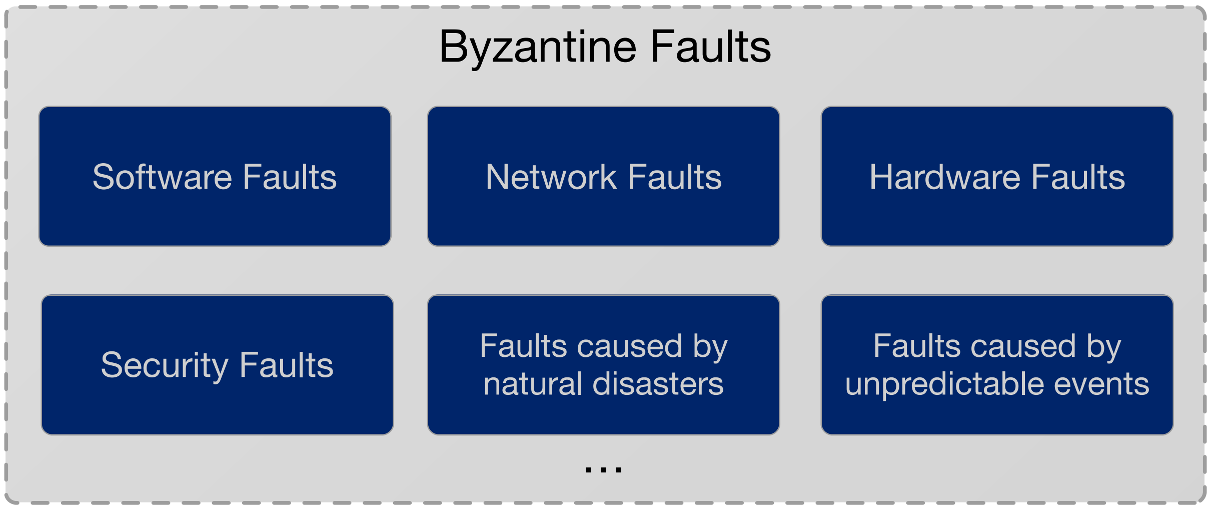


Figure 2.6: Examples of faults that cloud be the source of a Byzantine fault.

2.3 State Machine Replication

The state machine replication is a method for implementing a fault-tolerant service by replicating the initial state of a machine across distinct nodes of the system. Thus, all nodes of the service start with the same state and as the requests arrive on the servers and the nodes start processing them, thereby moving to other states. It is expected that after the initial state, all nodes follow the same sequence of states. The replicas, which deviate from the correct state sequence, are considered *faulty*. The SMR has been used for building Byzantine fault-tolerant algorithms by replicating the service state across several replicas, thereby guaranteeing consistency and availability even when a set of replicas deviates arbitrarily from the expected behavior. In addition to the SMR, it is necessary to incorporate a consensus algorithm to the SMR technique, ensuring that, even when a set of replicas deviate from the correct state sequence, the others (the correct ones) are able to agree on which is the correct state. The consistency is ensured since all replicas execute the same request in the same order (the non-faulty servers) and the availability is guaranteed by using the redundancy achieved through the state replication.

2.3.1 PBFT - Practical Byzantine Fault Tolerance

Practical Byzantine Fault Tolerance (PBFT) is an algorithm proposed by Miguel Castro and Barbara Liskov [28] for tolerating Byzantine faults with the main purpose of guaranteeing a good performance, therefore making it possible to apply it in practice. The PBFT assumes an asynchronous (or have a weak synchrony [29]), for making it capable to work on asynchronous environments such as the Internet. The service is replicated across several nodes ($3f + 1$) and only a set of replicas (f) can exhibit an arbitrary behavior. The algorithm moves through a sequence of configurations (views), where in each view a primary and a set of replicas is defined. The view changes when the replicas suspect that the primary has failed or if it has been demonstrating Byzantine behavior. The algorithm steps (see figure 2.7) are:

1) The client sends a request to all replicas and to the primary, which starts a three-phase agreement protocol (PRE-PREPARE, PREPARE and COMMIT) by sending a

message of PREPARE to all other $3f$ replicas.

2) After receiving the PRE-PREPARE message, the replicas validate the message and then each one sends a PREPARE message to all other nodes $3f$ of the cluster.

3) When each node (replicas and primary) receives $2f+1$ PREPARE messages, they verify the message authenticity and then send a COMMIT message to all other nodes of cluster.

4) After receiving $2f+1$ valid COMMIT messages, each node sends a REPLY message to the client containing the response of the client's request.

5) After receiving $2f+1$ REPLY valid messages, the client accepts it and can act on it.

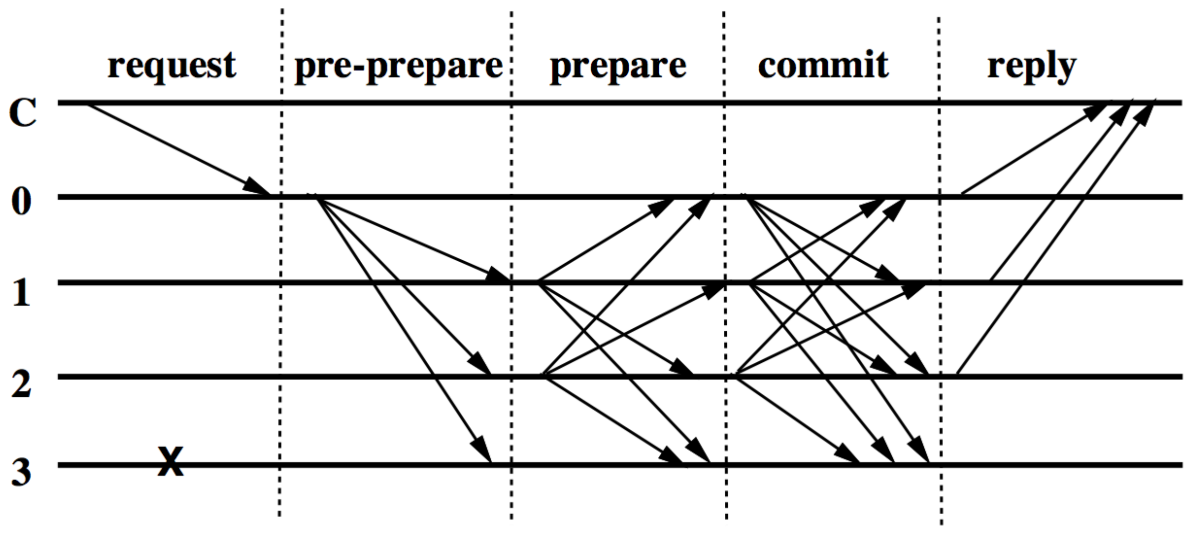


Figure 2.7: PBFT algorithm execution with $3f+1$ nodes. Source: [28]

The algorithm has a checkpoint mechanism, for guaranteeing that the current state of each node is correct, as well as reducing the overhead of a future view-change operation. This overhead is amortized, because the checkpoint operation is executed regularly, thereby ensuring when a view-change operation is needed, the non-synchronized states between replicas are as few as possible. Therefore, after receiving $2f+1$ CHECKPOINT messages with the same state, each node accepts that all processing done before is correct and then it erases the previous message logs and updates the latest checkpoint to the agreed one.

Thus, after receiving two CHECKPOINT messages with the same state, each node accepts that all processing done before is correct and then it erases the previous message logs and updates the latest checkpoint to the agreed one.

The view change is necessary when a replica suspects that the primary has failed. After receiving the client's request and if during a determined interval the primary does not send the PREPARE message, the replicas suspect that the primary has failed and, consequently, start a view-change operation.

2.3.2 Zyzzyva - Speculative Byzantine Fault Tolerance

Zyzzyva is a Byzantine fault-tolerant protocol proposed by Kotla et. al. [30], which uses an optimistic approach (speculative) to reduce the total number of steps done in the normal case execution (without fail). This approach allows Zyzzyva to improve the algorithm's performance in scenarios without the presence failures or Byzantine nodes. However, when there are failures the algorithm must execute an additional step for ensuring the response consistency. This additional step deteriorates the algorithm's performance in scenarios where the nodes are more likely to fail.

In Zyzzyva, throughout the algorithm execution (see figure 2.9), the replicas trust on the primary by executing the requests by the order imposed by primary. Therefore, there is no need to execute a further step to determine the order in which the requests must be executed (generally heavy computationally). When the replicas suspect that the primary has failed, they start a view-change operation. The client has a crucial role, since it is responsible for detecting if there are problems with the response generated by each server. Hence, the client waits for $3f + 1$ matching responses (same content) and if it receives this exact number of correct responses, it accepts the response as being correct and acts on it. Otherwise, if the client receives between $2f + 1$ and $3f$ equal responses, it starts a supplementary step, sending a commit certificate to all replicas, thereby ensuring that at least $2f + 1$ distinct replicas agreed on the order that the request was executed. If the client receives less than $2f + 1$ equal responses after sending the commit certificate, the request is retransmitted to all replicas and then the client waits during a determined period (t) by a correct response (this implies a correct ordering by the primary). Finally, if the client does not receive the response before t expires, the replicas suspect that the primary has failed and a view-change operation is triggered.

Zyzzyva also has a checkpoint mechanism for reducing the overhead of the view-change operations, since the nodes only need to synchronize the states (throughout

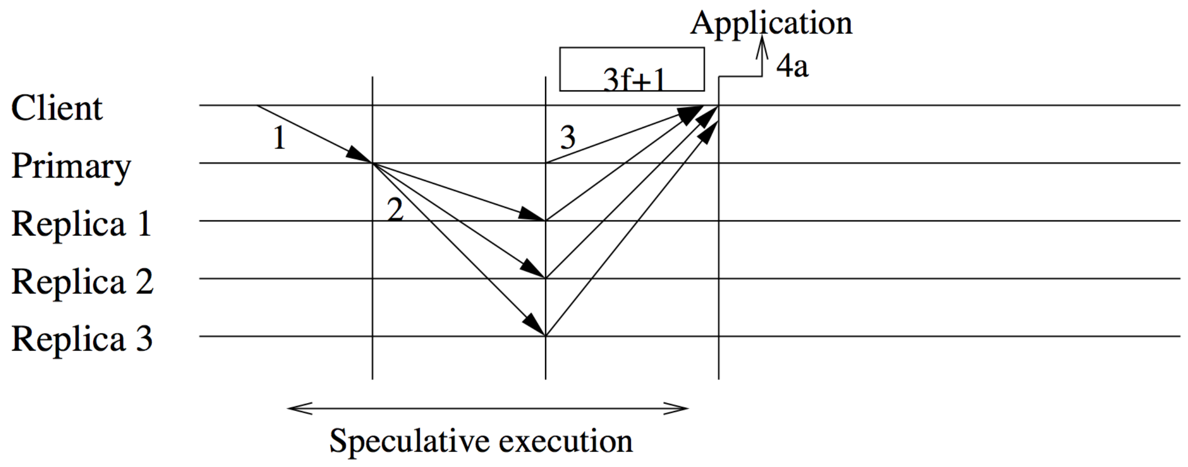


Figure 2.8: Optimistic execution of the Zyzzyva algorithm with $3f+1$ nodes. Source: [30]

the view-change operation) that were executed after the last successful checkpoint. Furthermore, it guarantees that after a checkpoint at least $2f+1$ nodes are in consistent state and that they have executed all requests properly up until present moment.

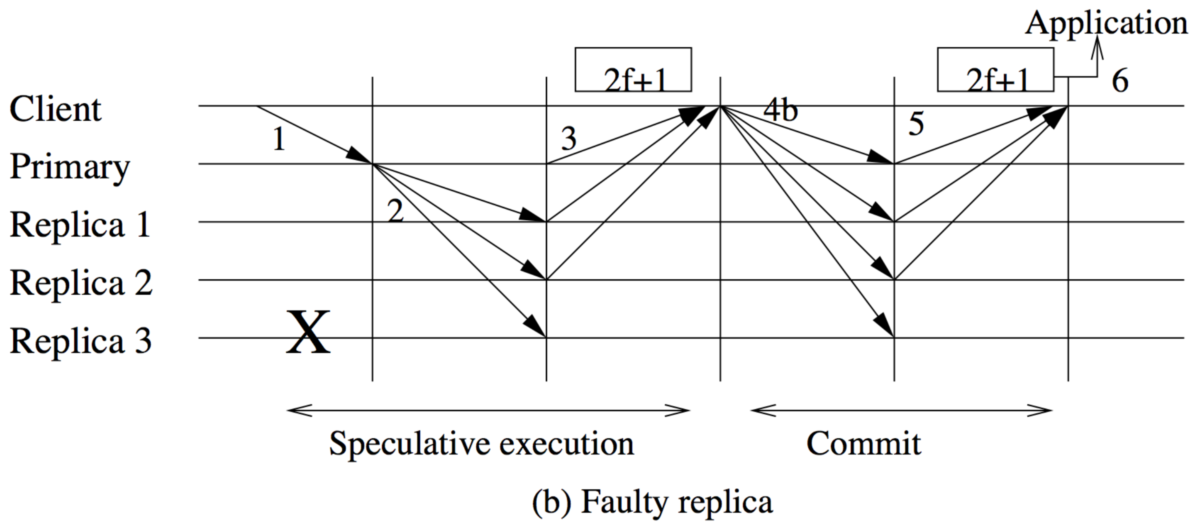


Figure 2.9: Zyzzyva algorithm execution with Byzantine nodes. Source: [30]

2.3.3 ZZ - Cheap Practical BFT using Virtualization

ZZ [31] is an algorithm for tolerating Byzantine faults, that takes advantage of the virtualization for reducing costs and resources used. The ZZ divides the agreement nodes (they are responsible for guaranteeing the execution order) and the execution nodes (they keep the system state and process the clients' requests) into distinct groups (see figure 2.10).

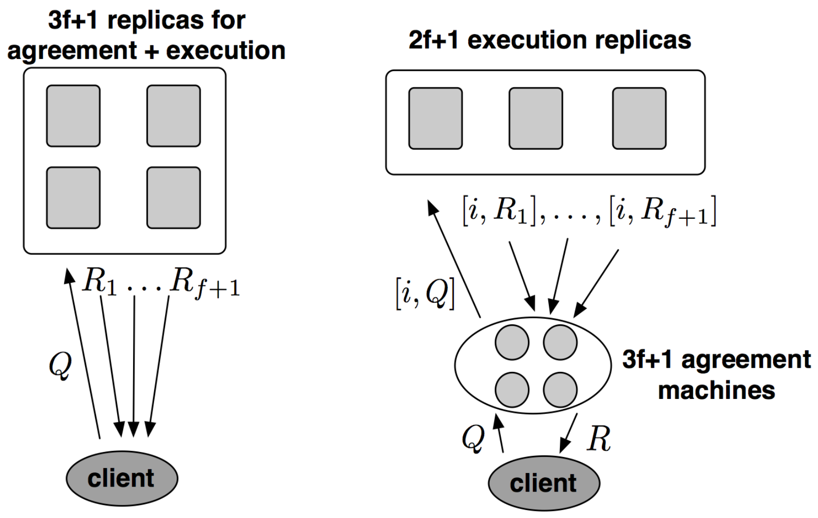
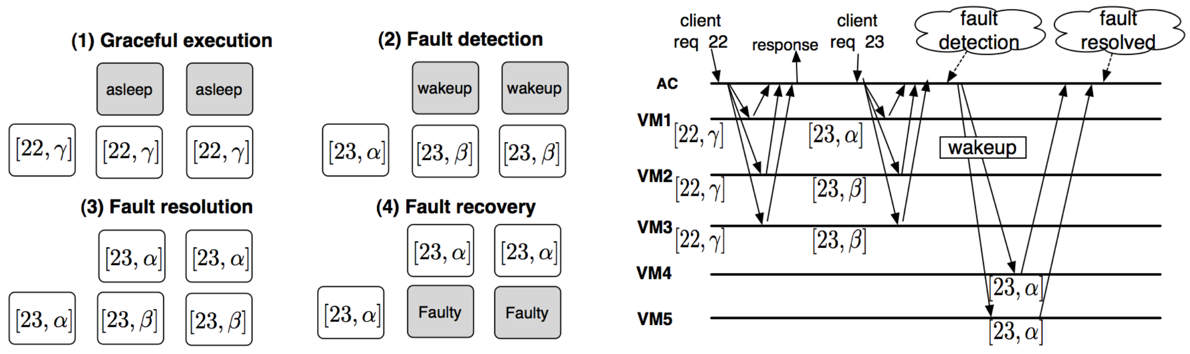


Figure 2.10: Functioning groups division implemented at ZZ: agreement group and execution group. Source: [31]

Like in PBFT [28] and Zyzzyva [30], for assuring the operations' execution are required $3f + 1$ nodes. In this case, as the nodes are responsible for executing only the agreement protocol and the algorithm allows the usage of virtualized environments, it assume that the nodes can be low-cost VMs with very limited resources.

The ZZ allows that the minimum number of nodes for executing the clients' request (execution group), are by far lower than other approaches ($f + 1$). The initial group size is $f + 1$ nodes, and as the conflict on the responses are detected (see figure 2.11), more VMs must be added into the processing group in order to guarantee that there is a majority of non-BFT nodes for being able to decide properly the response. Thus, in a scenario where all servers operate correctly, it only needs $f + 1$ VMs for processing the clients' requests, substantially reducing the number of messages exchanged and, therefore, enhancing the overall performance. In the ZZ, the client also has a crucial role, because it is accountable for verifying that at least $f + 1$ received messages are equal.

The ZZ has a checkpoint mechanism as well, but only the execution group needs to periodically synchronize the state between the nodes.



(a) Protocol for f=2 (b) Fault and Recovery Timeline

Figure 2.11: ZZ algorithm execution. Source: [31]

2.3.4 MinBFT

MinBFT [32] a is Byzantine fault-tolerant algorithm developed by Veronese et. al., which is based on PBFT [28]. The key advantage of this algorithm is the fact of only needs $2f + 1$ nodes. The reducing from $3f + 1$ to $2f + 1$ is achieved by adding a tamper-proof trusted component, which together with a monotonic counter, allows the algorithm to assign a sequence number to each client's request. Thus, it is possible to ensure that the processing order is the correct, since it was decided a by the trusted component. The total order during the processing is ensured by associating the counter to each message, so that the replicas only processing the message with counter value equal to $c + 1$, after processed the c. Since that counter is associated to the message by the trusted and tamper-proof component, it is possible to ensure that correct replicas process the message in the same order and the others, that do not process the message in the correct order, are considered as faulty. The trusted component must be able to work properly, even if it resides has been compromised. The Trusted Platform Module (TPM) (See 2.5) is the trusted component used by the MinBFT, since it is a secure cryptographic co-processor, which provides all necessary resources for guaranteeing the authenticity of the messages and the validity of the respective assigned counter. Like in PBFT; there is a primary and the other nodes are replicas, which processing the requests conforming to the order defined by the primary. The algorithms steps are:

1) The client sends a REQUEST message to all nodes.

2) Upon receiving the REQUEST message from the client, the primary uses the TPM for getting the counter value and assigns it to the PREPARE, which is sent to all replicas. After receiving the REQUEST message from the client, the replicas start a timer that, if it expires, triggers a view-change operation.

3) The replicas verify the authenticity of the PREPARE message and, if the message counter is correct, a COMMIT message is created and sent to all nodes.

4) After receiving $2f+1$ equal COMMIT messages, the nodes send a REPLY message to the client, which contains the response of its request.

5) Upon receiving $f+1$ equal messages, the client accepts this response and acts on it.

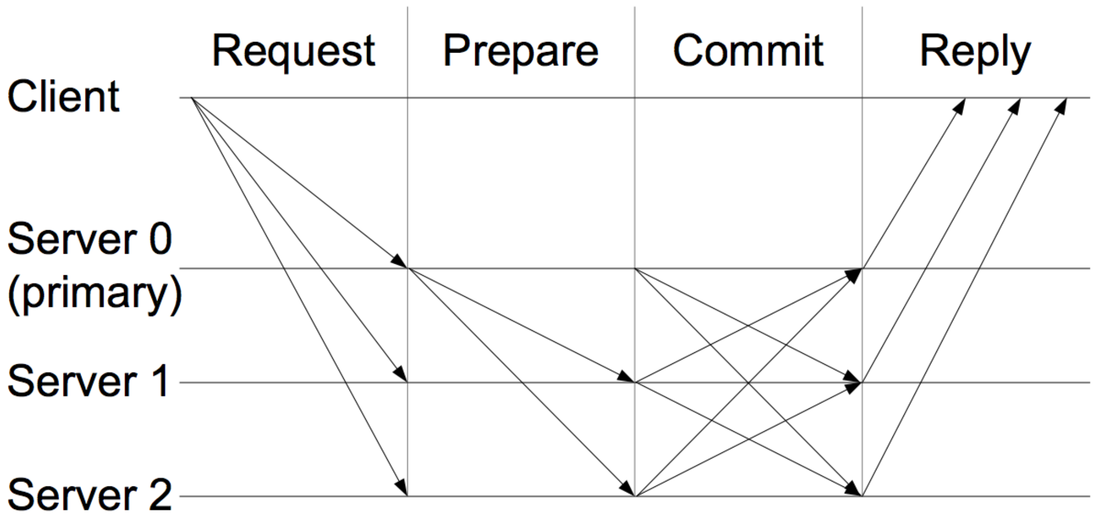


Figure 2.12: MinBFT algorithm execution. Source: [32]

According to Veronese et. al. [32], the main advantages of the algorithm are: 1) Reducing the number of nodes from $3f+1$ to $2f+1$, that reduces the system cost considerably. 2) Reducing the number of necessary steps during the algorithm execution, compared to previous approaches such as [28] [30]. 3) It uses the TPM as trusted component, that is an approach that has never used before in BFT algorithms [33] [34].

2.4 Comparison of BFT algorithms

The state machine replication has been the most used technique for building system to tolerate Byzantine faults [28, 35, 30, 33, 32, 31]. The state replication aims to ensure a higher level of resilience, thereby even when a set of nodes (f) have been attacked or have failed, other groups of nodes can operate correctly, replying to the clients' requests properly.

The PBFT is an agreement-based protocol for tolerating Byzantine faults that requires $3f + 1$ nodes. The PBFT does not use any tamper-proof component and, therefore, it needs an additional step for ensuring the total order. In this additional step (PRE-PREPARE) is necessary for synchronizing the replicas and for the primary assures which nodes are ready for executing the client's request. Since the PBFT was proposed more than 10 years ago, it does not have any virtualization mechanism for taking advantage of cloud environments.

Zyzzyva is a Byzantine fault-tolerant algorithm that uses a speculative approach for reducing the communication steps in the absence of faulty nodes. Both latency and throughput in these scenarios are tremendously improved, since the communication steps are close to the non-replicated approach (the nodes receive the client's request, process and reply). Though the Zyzzyva does not use any tamper-proof component to reduce the required number of nodes, some works have already been conducted in this direction, such as MinZyzzyva [32], thus decreasing the minimum number of nodes from $3f + 1$ to $2f + 1$ to tolerate Byzantine faults. Until now, any Zyzzyva based protocol did not exploit the virtualization and, therefore, these algorithms are not adequate for cloud environments.

MinBFT [32] is a Byzantine fault-tolerant algorithm that requires the lowest number of nodes ($2f + 1$), for tolerating f faulty nodes. This is achieved by adding a tamper-proof component for signing and verifying all messages sent throughout the protocol's execution. Though the tamper-proof component reduces the total number of nodes from $3f + 1$ to $2f + 1$, it adds a slower component that mandatorily has to sign and verify all the messages, thus compromising the system's scalability (the tamper-proof component can be quickly become the system's bottleneck). MinBFT has the lowest number of communication steps compared to other approaches, since all messages are signed and verified by a tamper-proof component, the number of nodes can be reduce, and, consequently, the number of communication steps can also be reduced from 4 (Like in PBFT) to 3. The figure 2.7 and 2.12, depict that the MinBFT removes the PRE-PREPARE communication. This step is no longer necessary, because in MinBFT all replicas only accept messages signed by the tamper-proof component with the respective monotonic counter attached, thus guaranteeing the total order execution without a further step.

ZZ uses a distinct approach for tolerating Byzantine faults. It splits the cluster in two distinct and functional groups of nodes: agreement and execution groups. Furthermore, ZZ uses low-spec virtualized nodes to reduce the overall cost of the cluster, aiming to be feasible the use of the algorithm in practice. The agreement group must have $3f+1$ nodes for tolerating f faults and these nodes can be VMs of low performance and cost, since it does not make any processing (they only make processing for agreeing on the message's order execution). The processing groups start with only $f+1$ nodes and if the client does not receive $f+1$ equal responses, it notifies the cluster and more nodes are added for guaranteeing a majority of correct responses *i.e.* a consensus of at least $f+1$ identical responses.

		Algorithm			
		PBFT [28]	**Zyzzyva** [30]	**ZZ** [31]	**MinBFT** [32]
Tamper-proof component		No	No	No	Yes
Speculative		No	Yes	No	No
Virtualized		No	No	Yes	No
Communication Steps		4	2 (Speculative) 4 (Non-Speculative)	5	3
Cost	Minimum	$3f+1$	$3f+1$	$3f+1$ (Agreement) $f+1$ (Execution)	$2f+1$
	Maximum	$3f+1$	$3f+1$	$3f+1$ (Agreement) $2f+1$ (Execution)	$2f+1$

Figure 2.13: BFT algorithms comparasion

2.5 Trusted Platform Module

The Trusted Computing Group (TCG) is an organization created in 2003 by big IT companies such as IBM, Intel, AMD, Infineon, Microsoft, Hewlett-Pack and Sun Microsystems with one purpose: to improve the security of the computers and the communication networks. The TCG is accountable for creating several specifications, describing how a trusted system should work (trusted computing). The main specification created by TCG is the Trusted Platform Module (TPM): a security chip with its own co-processor, memory, firmware and software.

2.5.1 Architecture

The TPM is composed by distinct and independent units, that together provide all functionalities of this chip. The architecture (see figure 2.14) and its respective units are:

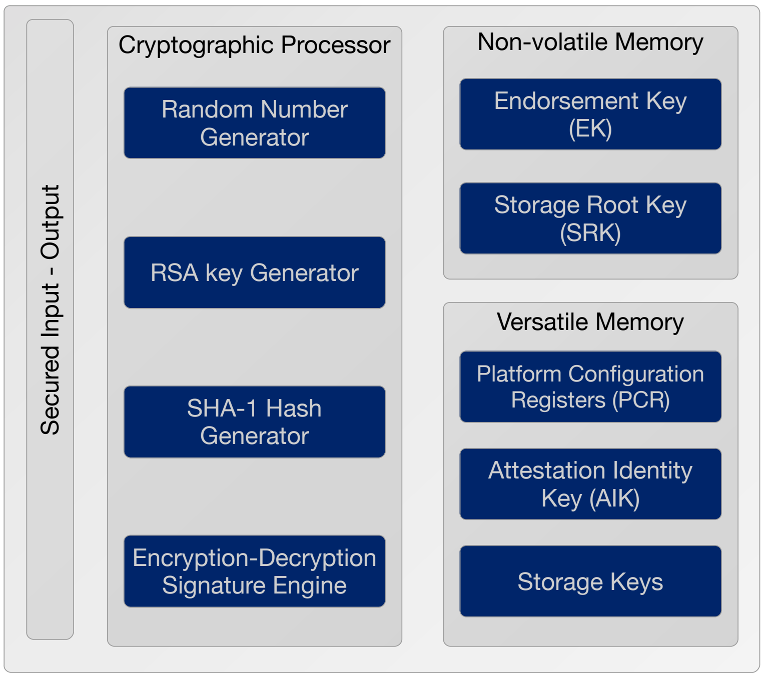


Figure 2.14: Trusted Platform Module architecture

Cryptographic co-processor and functional units - This component is responsible for executing all cryptographic operations that can be executed by the TPM. The TPM has a high quality random number generator (RNG), which is capable to generate non-reproducible and non-periodic numbers. The generated numbers can be used afterwards for internal or external applications. There is also a RSA key generator capable to generate keys with sizes higher than 2048 bits, and it is possible to encrypt, decrypt and sign messages both for internal and external applications. The TPM provides a dedicated unit for generating hash functions (SHA1 - Secure Hash Algorithm 1) that are very important and widely used by several applications to provide some security features.

Volatile Memory - The TPM's volatile memory is accountable for storing the Platform Configuration Registers (PCRs), the Attestation Identity Key (AIK) and the Storage Keys (SK). Since it is a volatile memory, all the registers are discarded after the machine has been shut down. The PCRs are responsible for storing the systems component measurements, such as Basic Input/Output System (BIOS), hypervisor and firmware. The TPM only allows to extend[1] a previous PCR value, so each new component measurement is extended (combined) from the previous PCR value and, therefore, at the end, the PCR will contain the combined value for the whole system component stack. Since the extension operation is not commutative, the order that operations are done is extremely important. The AIK are a key pair used only to sign the structures belonging to the TPM. These keys allow to verify the authenticity of a message generated by a TPM. The SK is a unity for storing other key types, such as registers and necessary information to the proper operations of other specific applications.

Non-volatile Memory - The non-volatile memory is responsible for storing the Endorsement Keys (EK), the Storage Root Keys (SRK), counters and all sorts of data that needs to be persisted. The EK are a RSA key pair of 2048 bits that are incorporated to the TPM during the manufacturing process. The EK private part is used to identify unequivocally the TPM chip. For security and privacy reasons, the EK private part is only used to encrypt structures that will not leave the TPM and to decrypt external or internal structures. The non-volatile memory is also able to store other data structures, such as monotonic counters, that together with the RSA encryption unit and hash functions are crucial for implementing our architecture.

2.5.2 TPM features

The TPM allows to create different applications capable to take advantage of its features, such as the random number generator of high quality, asymmetric keys generator, the ability to create more secure hashes, etc. Nevertheless, the TCG defines [36] that the TPM must provide at least three main features:

Platform Integrity - The TPM must be able to verify the system's state during the

[1]The "extend" operation consists in combining a previous value with the new through a particular operation, replacing the old value by the produced.

boot time by verifying it through reliable metrics (trusted conditions) and extend this reliability until the OS is completely booted and working. For providing this feature, the TPM together with the BIOS, uses the Platform Configuration Registers (PCR), that act as sort of a container by storing the measurements of each system's component. The TPM computes the hash of each measurement (each system component) for assessing if the system is trusted or not. The measurement result is stored in a PCR (this process is called "Extending the PCR"). The PCRs can be extended many times until the final value has been calculated. After measuring all system's components and extending them to the TPM, this value will represent an accumulation of all executed code until the system is fully booted. Thus, the TPM can verify the system's final state by checking the final hash stored into the PCRs and, consequently, it can decide if the system can be trusted or not.

Disk Encryption - The TPM allows that the data stored in a computer are encrypted and associated to the specific platform configurations. This association is done by using PCR values and asymmetric keys, which cannot be accessed or extracted from the TPM. Thus, for decrypting the data previously encrypted, the platform configurations must be the same that were used previously, as well as the same TPM with the same private key. The Sealed Storage is the main data protection method specified by TCG. In addition to this method, there are two more related to data protection: Binding and Sealed-Signing. The first is similar to the Sealed Store, but in the binding operation, the TPM can choose if the used key for encrypting the hard disk data can be transferred to another TPM or if it only belongs to the TPM that created the key. On the other hand, the Sealed-Signing allows the algorithm to add specific PCRs, to ensure that the system that is verifying the signature fulfills all required configurations imposed by the TPM that previously has signed the content.

Integrity Report - It enables to determine the hardware and software current configurations to remotely attest the platform's state, thus avoiding unexpected alterations. To be able to do remote attestations properly, the PCR values must be signed with the Attestation Identity Keys (AIK), thereby ensuring the signature authenticity, since each TPM has its own unique AIK. For determining that an AIK belongs to a trusted platform without revealing the TPM's identity, algorithms for Direct Anonymous Attestation [37] [38] [39] can be used.

2.6 Database Benchmarks

A benchmarking is used to measure the performance of a process, software or system to compare the results achieved against to the best practiced by the industries. In the field of databases, benchmarking is used to measure and compare the performance of distinct Database Management Systems (DBMSs) or NoSQL databases, thus determining which one is more suitable for a particular case. However, there are several benchmarks available and some of them do not trustworthily represent a real scenario and do not have the necessary complexity to be a standard benchmark. To avoid this, the Transaction Processing Performance Council (TPPC) was created for defining and standardizing the relational database benchmarks.

Since then, the TPPC has been creating several benchmarks, that are now the standard for measuring and comparing the relational databases' performance. The most popular benchmarks are:

TPC-C - TPC-C is industry standard for measuring the performance of Online Transaction Processing databases. It simulates a retailer company, where clients can order products and distinct warehouses provide the products for clients of a determined district. Therefore, this benchmarking depicts a widely used business model, mainly by the IT companies that sell goods and products through the Internet. The TPC-C has five main queries of different types, such as read-only, write-only and read-write. This mix of transactions allows to test the major part of possible operations done by the data-schema provided by the TPC-C. For example, the new-order transaction (the client buys something, and, therefore, it must create a new-order for processing the client's order) and the payment transaction for computing the client's payment related to a certain order. To approximate the TPC-C even closer to a real system, the TCG defined that 10% of the items of a new-order transaction must be provided by a remote warehouse (a warehouse of another district). Therefore, it simulates a run out of stock of some of items belonging to a new-order transaction, thereby forcing the system to resort to a remote warehouse to process the client's order and, therefore, enhancing the query complexity (more complex join operation) and concurrency between transactions. The metric used by the TPC-C for measuring the databases' performance is the transactions per minute (tpmC).

TPC-H - TPC-C is a standard benchmark benchmarking designed for decision-making systems for measuring the performance in Online Analytical Processing (OLAP) databases. Since it is designed for an OLAP databases, the queries are business oriented and ad-hoc, that were built for running over large volumes of data. The queries have a high degree of complexity for illustrating a real and complex decision making system (business intelligence). For example, some queries analyze the company's profit obtained in the last year, others analyze what the market share of the company is, etc. The performance metric used by TPC-H is called the TPC-H Composite Query-per-Hour Performance Metric (QphH@Size). This metric is composed by several aspects related to the databases' performance, such as database size, query-processing power, throughput by multiple concurrent users, etc.

Since this system is designated for cloud environments, the TPC-C is more adequate for measuring the system performance and scalability.

CHAPTER 3

CloudBFT

This chapter describes the CloudBFT architecture by depicting the functioning of all components. Section one the system model and the assumptions made. Section two gives an overview of how the system works and which are the main components. Section three describes how the system is organized for assuring maintainability, as well as for being as close as possible to the web frameworks found in the market. Section four characterizes the client functioning and its relevance during the algorithm's execution. Section five details the primary's role and how it is important for the proper functioning of the entire system. Section six describes the processing groups, as well as the main decisions to make this architecture elastic and scalable. Section seven illustrates the functioning of the database nodes and how they ensure data consistency without compromising the system's performance. Section eight characterizes the working of the trusted component (TPM) and the operations that it must implement. Section nine describes the messages exchanged throughout the BFT algorithm execution and the content of each message. Section ten depicts the TPC-C data schema as well as why it is a relevant benchmark for OLTP databases.

3.1 System Model

In this work a Byzantine failure model is assumed, 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*. A Byzantine fault-tolerant algorithm is used 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. MinBFT [32] was used as the BFT algorithm, because it needs only $2f + 1$ replicated nodes to tolerate f failures. Since one of cloud computing characteristics is to be cost efficient, we choose a BFT algorithm that can provide performance and elasticity with minimum number of nodes possible, thus reducing the total overall cost for tolerating Byzantine faults and, therefore, approximating our approach to the practical environments.

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 tamper-proof component, such as collision-resistant hashes, encryption and digital signatures. The adversaries can take control of an entire physical machine and, consequently, controlling all VMs of its physical machine. The system must tolerate these type of physical attacks, since it was designated for cloud computing environments, where the VMs are accessed only remotely and, therefore, we cannot make any assumptions related to the physical access control of such VMs. In spite of the cloud providers can offer a Service-level agreements (where are specified several assurances for the client, such as performance measurement, problem management, warranties, disaster recovery, etc.), for tolerating Byzantine faults, these assurances are insufficient. Furthermore, both 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 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, it is assumed 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 the MinBFT is used as the algorithm to tolerate Byzantine faults, any Byzantine fault tolerant algorithm with some changes, such as PBFT [28] or Zyzzyva [30], is suitable for the system presented in this work. Thus, this approach is so general and adaptable that is able to work with any Byzantine algorithm.

Since the system's design is specifically intended for cloud environments, it was stipulated that the nodes (VMs) that execute the algorithm must be distributed across $2f+1$ or $3f+1$ distinct physical machines (in a multi-server or multi Availability Zone configuration, for example). Thus, if a physical machine is compromised due to a hardware fault or if multiple VMs are affected by a cross-VM attack, the faults will not compromise the system and, therefore a majority of machines ($2f+1-f$ ou

$3f + 1 - f$) remains intact (running the algorithm properly).

3.2 Architecture Overview

The system presented in this work is designed for cloud environments for taking advantage of elasticity and scalability, but at the same time, for being highly resilient to be able to tolerate Byzantine faults. This work has several challenges because for tolerating Byzantine faults is necessary to build a tremendously complex distributed system, to ensure that hardware and software faults, physical and remote attacks and natural disasters do not compromise the system. Thus, the system remains consistent and able to tolerate any type of faults, replying properly the clients' requests even in the worst scenarios. In addition, the system must be able to scale according to processing power required and, therefore, it must be elastic and scalable as well as Byzantine fault-tolerant. The biggest challenge of this work is to find a balance between these challenges in order to assure one without compromising another.

The virtualization is the key component of the system's architecture, since it is through this technique that is possible to parallelize the processing of the requests and the accesses to the data. Thus, we create processing groups (See Figure 3.1), that are distributed across distinct physical machines and each group processes requests independently of the others, thereby enhancing the system's performance by parallelizing the processing and reducing the resource consumption through the use of virtualization. The in-depth description of how CloudBFT parallelizes processing of the clients' requests, how the group division is made, as well as how the database nodes are partitioned are described in more detail in the next sections.

The MinBFT [32] was chosen to be adapted and inserted into the logic tier for tolerating Byzantine faults because, besides it has already been proven that works properly with a tamper-proof component, it only needs to replicate the state across $2f + 1$ distinct PMs. The amount of resources needed for orchestrating and running the BFT algorithm is crucial, since both IaaS and PaaS (the most widely used models by cloud customers) are service models that charge their customers according to the number of VMs rented and by the respective processing and network traffic performed by each VMs. However, the MinBFT must be adapted, enhanced and optimized for making it capable to work properly in cloud environments, for ensuring that all requirements of the architecture presented in this work are fulfilled. BFT algorithms are composed by several components that are crucial for the proper working during its execution. These components are described in more detail in the next sections:

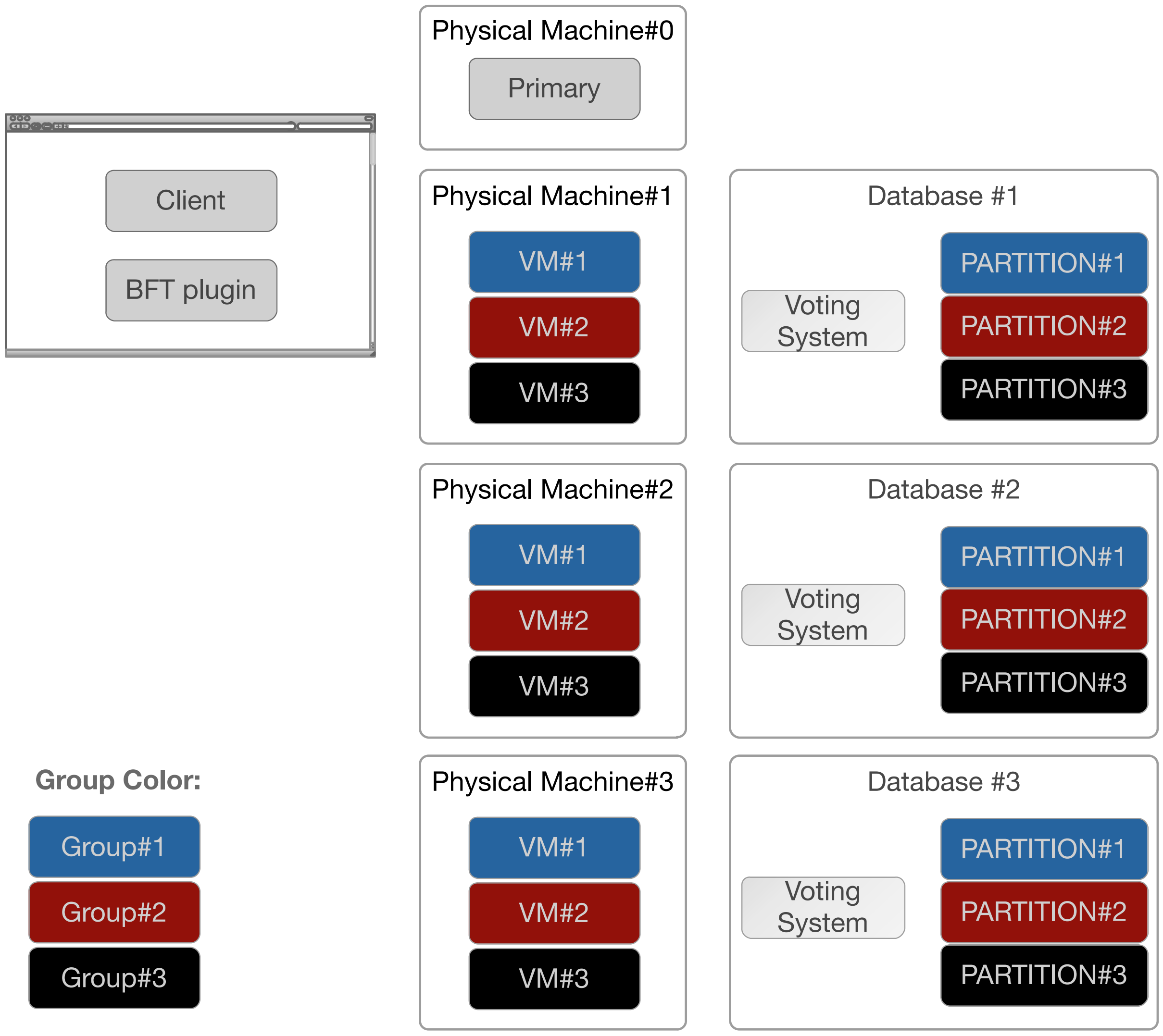


Figure 3.1: System Architecture

3.3 3-tier Architecture

The architecture presented in this work was designed aiming applications that use the HyperText Transfer Protocol (HTTP) to establish the communication between clients and servers. The HTTP is widely used on the Internet by web applications, however since it belongs to the application layer, it is necessary to have another protocol in the transport layer for handling lower level operations. The Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP) are the most commonly used options. Like in most well defined, designed and implemented web applications, our architecture follows a three-tier model. This division achieved through this model allows creating application with greater flexibility and loosely coupled, thereby reducing the cost of future changes, testing more easily the application and improving the maintainability. The main web frameworks, such as ASP.NET, Django, Ruby on Rails, Grails, Spring MVC, use the same paradigm and have been having great adoption by the companies[40]. Cloud providers have invested a lot in web frameworks to be the basis of one its main services: Platform as a Service (PaaS). Besides offering greater flexibility, the PaaS services have been increasingly adopted by many programmers because it is considerably easier and fast to deploy applications in these services. The web frameworks have direct influence on PaaS's features and, surely, the PaaS's future will depend on the web frameworks.

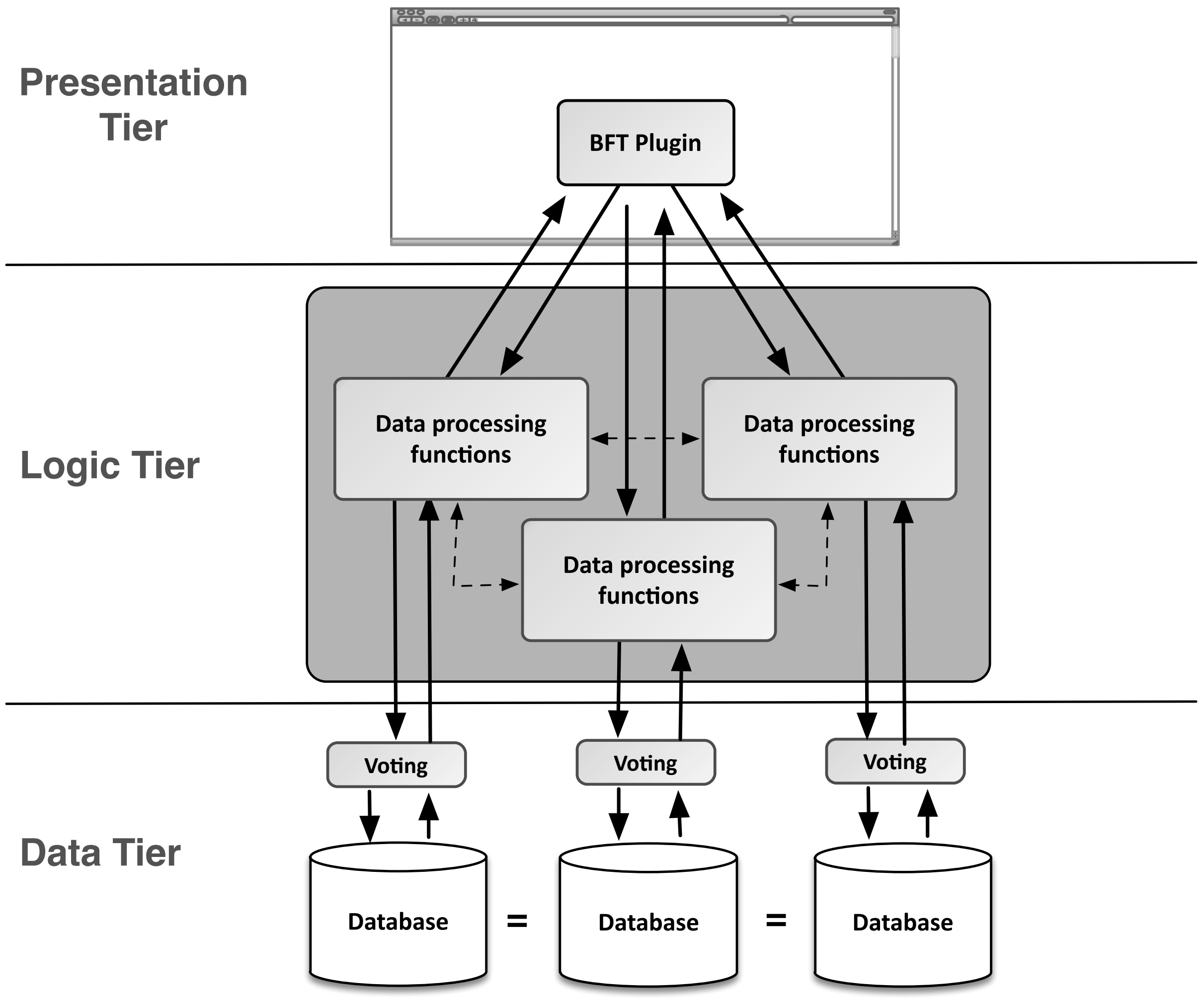


Figure 3.2: System Architecture

The 3-tier architecture decomposes the system into three distinct and functional layers: presentation, logic, and data. The presentation tier is responsible for displaying properly and effectively the information to the client, which, usually accesses this information through a webpage. The logic tier is responsible for processing the clients' requests by doing the needed computation, requesting/updating/persisting data from/to the lower tier (data tier). The data tier, in turn, is accountable for managing the data, replying requests from the logic tier related to the data management (consulting, updating and removing, etc.), as well as managing the concurrency between simulateneous access, thereby guaranteeing data consistency even in scenarios of high concurrency in the data tier.

In our approach, the presentation tier besides displaying the information to the client, is also responsible for managing the sending and receiving of the clients' requests by guaranteeing the Byzantine fault tolerance through a plugin installed into the browser (See Figure 3.2). The Byzantine fault-tolerant algorithm is inserted into the logic tier, being responsible for receiving the clients' requests and for executing the agreement protocol, accessing/persisting data to the data tier and, finally, sending the response to the client. Unlike typical web frameworks, our system needs some additional components, such as the primary and processing groups, both being they crucial for guaranteeing Byzantine fault tolerance.

The use of relational databases obligates the data tier for guaranteeing the consistency between the replicated nodes and, simultaneously, tolerating Byzantine faults during the data processing. Like in typical web frameworks, the data tier is only responsible for managing the data without any information, both about logic tier and about presentation tier, thereby remaining loosely coupled and modulate. In addition to replicated database nodes, the data tier must have a voting system attached to each database node. It is extremely important to design a system as close as possible to what is currently used in cloud environments, because the closer the architecture is to what is found in cloud computing, the less changes the client need to perform for adding Byzantine fault tolerance in its system. Achieving this likeness increases tremendously the possibility of a future adoption of our architecture.

3.4 Client

The client is accountable for sending the requests containing the operation to be executed by the cluster. The communication with the cluster goes through a web service accessible with a special Web browser plugin (see Figure 3.1). The plugin is a crucial component, because without it, clients are unable to use the BFT algorithm, since the plugin is responsible for coordinating the sending of the clients' requests for the correct nodes, as well as the receiving of equal responses from at least $f+1$ proper nodes. Before sending the message, the client must encrypt the message with a session key (SK) previously exchanged and then, it must create the message's digital signature by signing the message attached to its hash value, thereby creating a proof for future integrity verifications. Therefore, only the nodes intended for receiving the client's request will be able to decrypt and read the message's content. To exchange the private keys, algorithms such as the Diffie-Hellman [41] or Menezes–Qu–Vanstone [42] could be used.

The BFT plugin must also manage the receiving of responses from the nodes of the cluster. Therefore, after sending the request to the cluster's nodes, the plugin waits for, at least, $f+1$ correct equal messages. Upon receiving the $f+1$ equal messages, the BFT plugin accepts these messages as being correct, since it surely was decided by the majority of nodes belonging to the cluster. If after a certain interval (t), the client has not received any response from the cluster, the plugin resends the request. To assure the confidentiality of the messages, the cluster's nodes encrypt the content of the response with the same SK previously used by the client for encrypting the request. Furthermore, a digital signature is created and embedded into the message, thus assuring also in the reply messages: integrity, non-repudiation and authenticity.

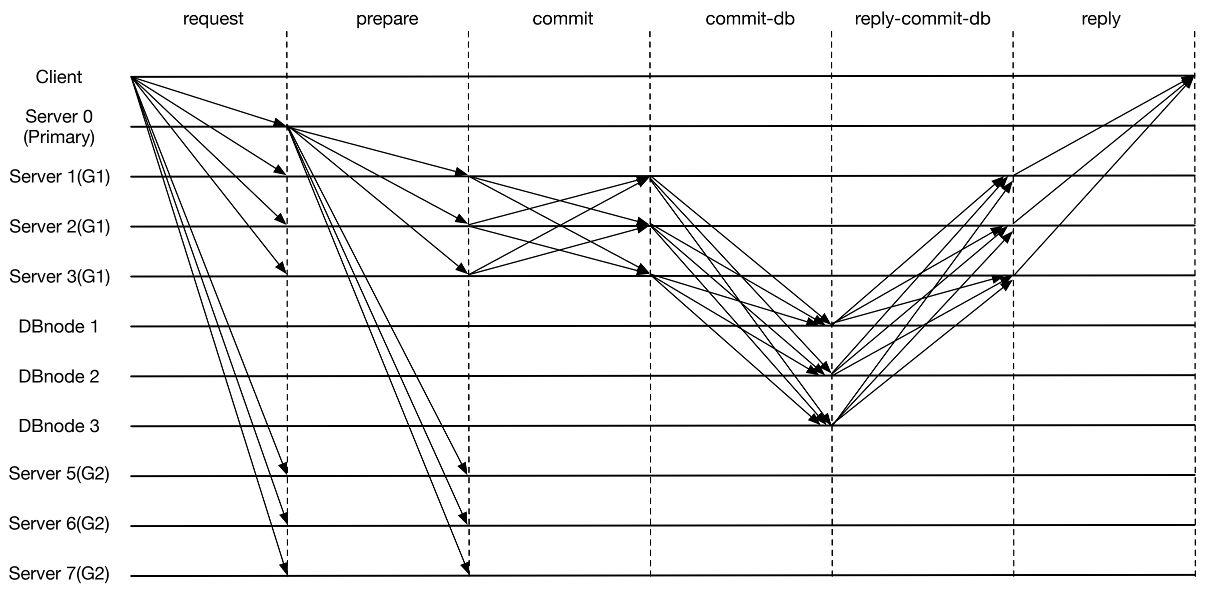


Figure 3.3: Normal case execution with 2 groups with size of $2f+1$

3.5 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 specific partition they refer to (possibly more than one), thus letting the primary direct the message to the appropriate group in charge.

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 chapter, 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 3.3). The primary node guarantees totally ordered execution by associating a monotonically increasing counter to all PREPARE messages. A disadvantage of this approach is that we need a primary node in addition to all worker VMs, although one primary node is sufficient to co-ordinate 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 need to process requests. Thus, 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

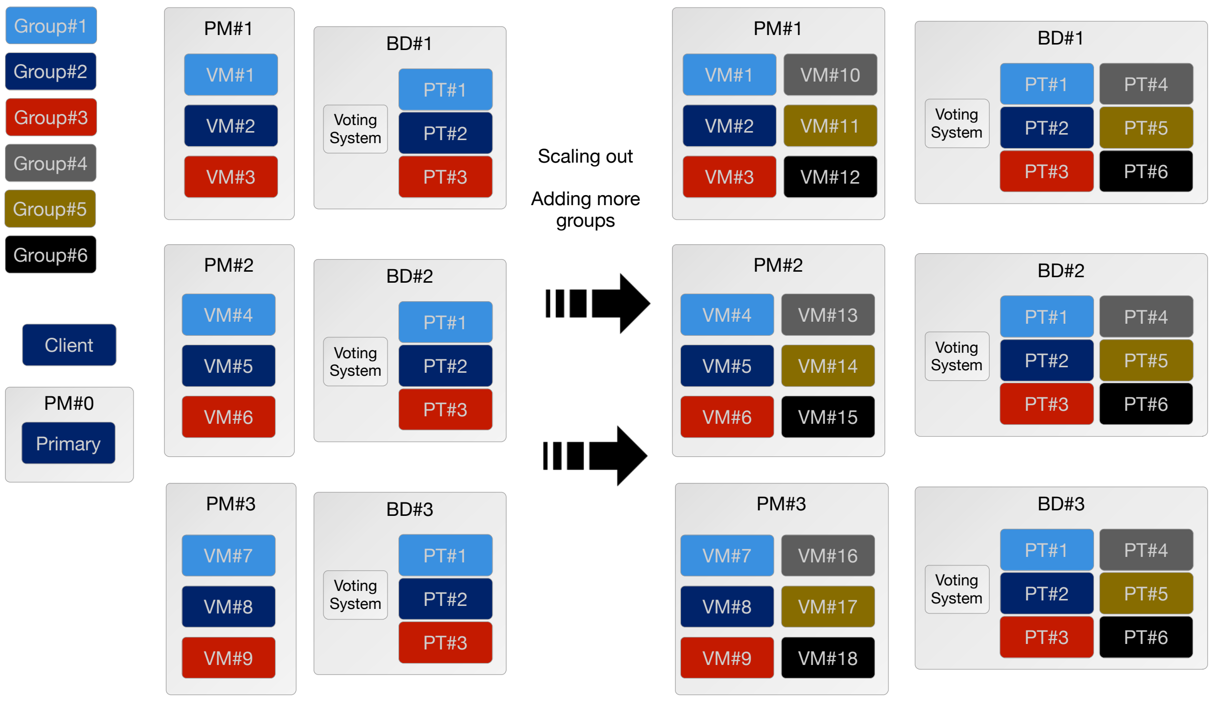


Figure 3.4: Scaling out the system *i.e* adding more groups for meeting the processing requirements.

must not share the PMs 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, as the system load increases and the system scales out, the addition of the primary becomes less significant.

As soon as the group nodes receive the REQUEST message from the client, they start a timer to detect if the primary is faulty. If the timer goes off, 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 the other nodes that it is alive, the primary broadcasts the PREPARE message to all elements of the cluster, which should cancel their timers in response (see Figure 3.3). The view-change is triggered only if the timers of $f + 1$ nodes of the same group expire.

3.6 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 3.3). 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 other peer nodes of the same group (see Figure 3.3). 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 node fails. 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 times this group is only responsible for executing transactions on this partition (see Figure 3.1). However, when a transactions 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 computer power and dividing the load across more computational 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.

3.7 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 3.1) 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. Therefore, only this group can execute disjoint transactions (single partition transaction) on that partition. However, a group can execute transactions on more than one partition when the operation is not disjoint, requiring access to multiple partitions.

3.8 Tamper-proof component

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 [32] 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 cer-

tificate previously crated by other TPM. The TPM generates a UW according to the encryption used (RSA [41] or HMAC [43]) 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.

The Figure 3.5 depicts the keys used in the system. Each TPM 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 it would be even more robust if a asymmetric key would be used together.

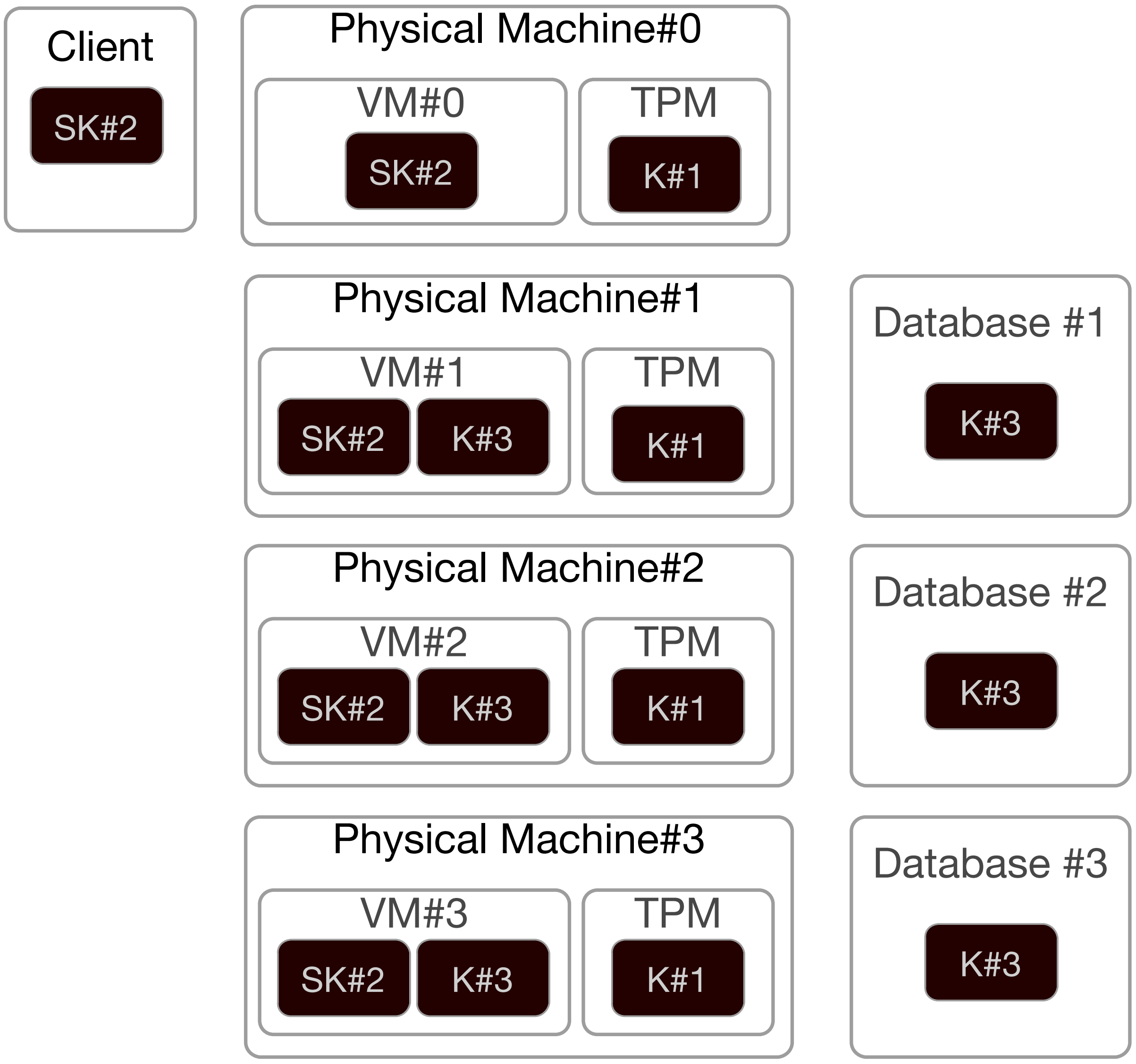


Figure 3.5: Keys used in the system

3.9 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 the execution of the algorithm. Thus, 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 3.1.

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

Table 3.1: Operation's labels and their respective meanings

- $\langle \text{REQUEST}, c, op, t\rangle_{\sigma_c}$ — 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 of the cluster. The message must contain the primary identifier, the view number, the group identifier, which is charged to process the message, the message containing the client request and the unique warrant (UW) created by the TPM.

- $\langle$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.

- $\langle$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.

- $\langle$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.

- $\langle$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.

3.10 TPC-C properties

TPC-C [44] is the main industry benchmark for measuring the performance and scalability of Online Transaction Processing (OLTP) databases. It was designed by the Transaction Processing Performance Council (TPC) and has distinct type of transactions, such as read-only, write-only and write and read, thereby allowing to measure the entire data model consistently and reliably. Despite the TPC-C is not a data model for any specific market, it represents a kind of industry that aims to sell, manage and deliver a service or product. This model is similar to a wide range of companies, whether they be large, medium-sized or small.

The company represented by the TPC-C is a typical retailer, that has lots of *warehouses* geographically distributed across distinct *districts* (see Figure 3.6), in order to serve a higher number of clients efficiently and as the company grows, more *warehouses* are added. Each *warehouse* supplies 10 *districts* and each *district* has an average of $3,000$ clients. The company has more than $100,000$ *items* that are kept in *stock* by all *warehouses*. Clients can create a new *new-order* or check of an existing one (*order*). Each *order* has an average of 10 distinct items (*order-line*) and the client can check his order *history*.

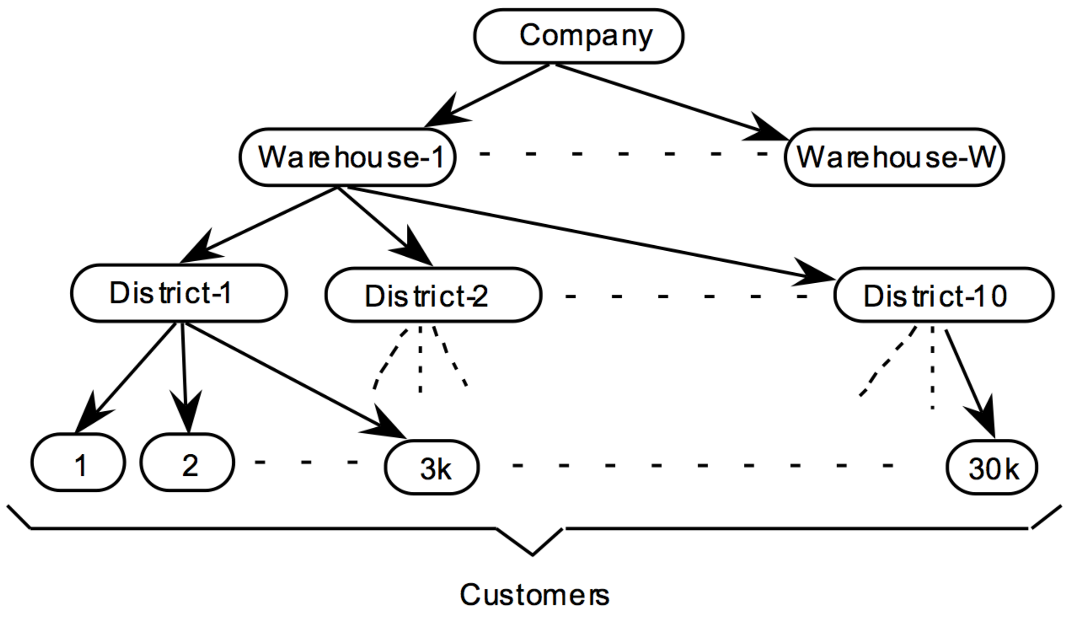


Figure 3.6: Hierarchical representation of TPC-C. Source: [44]

The company's system is also used for processing the clients' payments, creating and managing orders, deliveries and for analyzing the warehouses' stocks for checking if there are products out of stock or close to it. 99% of all existing items of an order (*order-line*) are in stock in the same district warehouse. However there another 1% of items that are not available in the warehouses belonging to client's district and, consequently, must be supplied by a remote warehouse (of another district). This allows creating orders with higher complexity (items from distinct warehouses), since the database needs to manage transactions with higher complexity, thus simulating an environment closer to the real functioning of a company that sells and delivers goods and services.

The diagram 3.7 depicts the TPC-C Entity-Relationship model, that has the characteristics described above. The TPC-C must be partitioned for fulfilling the architecture requirements and, therefore, achieving a better performance by removing the contention on the database. Hence, distinct groups process operations related to distinct partitions (distinct warehouses), thereby removing the concurrency between transactions that need to access distinct partitions. To remove that contention in TPC-C, the biggest (*line-item*) table must be partitioned by warehouse, and, therefore, each processing group will be in charge for processing queries of a set of warehouses. As the system scales, each group works with a lower number of warehouses, until the point of one group processes transactions of only one warehouse, and therefore of one partition.

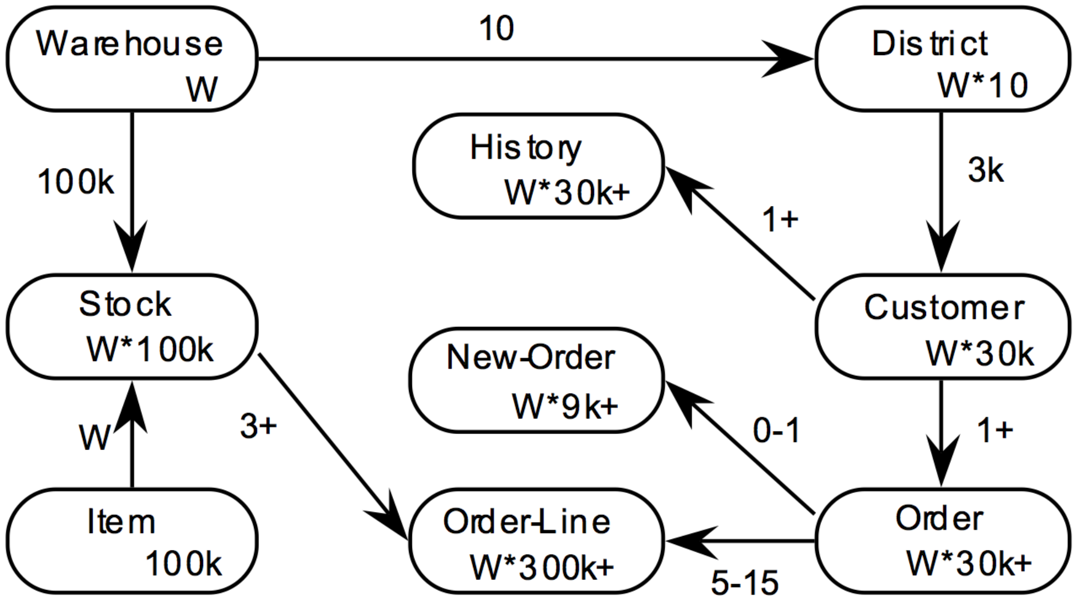


Figure 3.7: Entity-Relationship model of TPC-C. Source: [44]

Amazon Store [45], the world's largest online retailer, uses a similar data schema to TPC-C, where it has several warehouses scattered throughout the world and each warehouse is accountable for supplying a particular region and, consequently, a set of clients that belongs to this region. The Amazon business model reinforces the TPC-C's relevance in the OLTP database benchmark field. Therefore, since our system uses a OLTP database, the TPC-C is the best option for proving that our system is feasible and that might be applied in practice.

CHAPTER 4

Cloud BFT implementation using a $3f + 1$ BFT algorithm

To demonstrate that CloudBFT can run with any well-defined BFT algorithm, we also implemented a version of CloudBFT using a BFT algorithm that requires $3f + 1$ nodes for tolerating f Byzantine faults. Although an algorithm that uses a lower number of nodes ($2f + 1$) is better for cloud environments, because the lower the number of nodes is, the lower the cost with the cluster will be. We intended to demonstrate that our architecture is as general as possible, thereby being able to work with any BFT algorithm with minimal changes. The execution of the BFT algorithm is the heaviest step of the architecture and, therefore choosing the right algorithm is a very important decision for assuring a good performance.

There are some implementations of BFT algorithms that need $3f + 1$ nodes to tolerate f faults. Most of these implementations are based on PBFT and do not use any kind of speculation for reducing the total number of nodes, like is used in Zyzzyva. We used BFT-SMaRt [46] as the $3f + 1$ BFT algorithm, because it is a robust and flexible implementation.

Most of the changes made for making the $3f + 1$ algorithm capable to fulfill the CloudBFT's requirements are similar to the changes made in the $2f + 1$ algorithm (see Chapter 3). However, there are important differences that are crucial to ensure the Byzantine fault tolerance, without compromising the scalability and elasticity and vice-versa. Since BFT-SMaRt is an implementation based on PBFT, in this chapter, we will use the nomenclature used by PBFT.

To implement this version of CloudBFT, we based on the same assumptions (see Section 3.1) that were assumed in the $2f + 1$ version. MinBFT reduced the number of nodes from $3f + 1$ to $2f + 1$ by adding a tamper-proof component that is responsible for signing and verifying all messages throughout the algorithm's execution. The absence of the tamper-proof component and consequently a higher number of nodes are the main difference between the CloudBFT's implementation presented in the previous chapter ($2f+1$) and this one ($3f+1$).

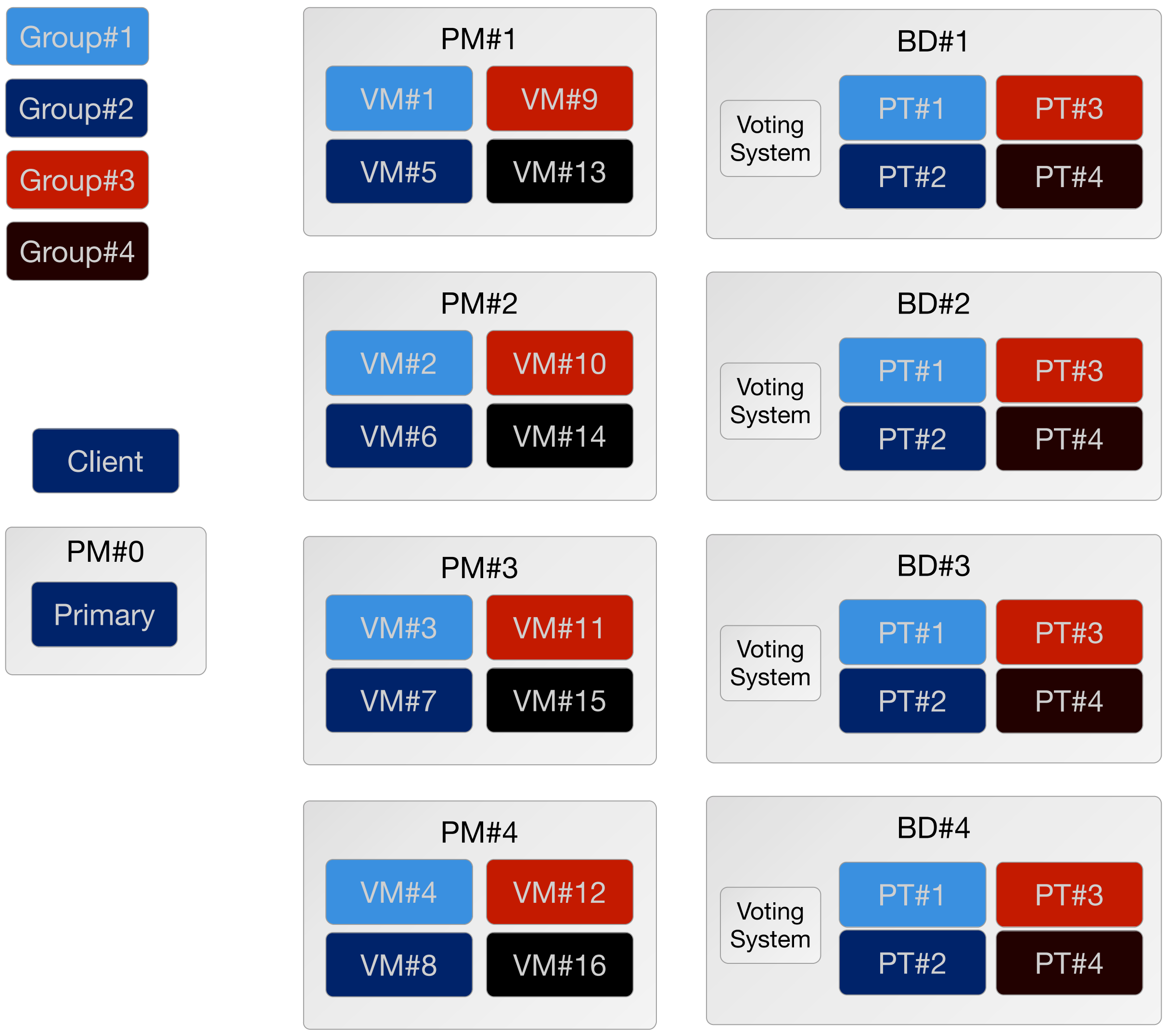


Figure 4.1: System architecture using a $3f+1$ BFT algorithm implementation

Like in the previous implementation, the client must have a BFT plugin installed into its browser to be able to execute request with Byzantine fault tolerance. The BFT plugin must know which are the BFT algorithm used by CloudBFT, since the number of nodes may change ($2f+1$ or $3f+1$) between distinct implementations. Therefore, in this implementation, the BFT plugin sends the REQUEST to all nodes belonging to the cluster, then it waits for $3f$ matching responses (REPLY messages).

The primary is responsible for orchestrating the execution by determining the order that the operations must be processed. However, unlike the $2f+1$ algorithm, it is done in two steps (see Figure 4.1), where the first one is for synchronizing the nodes (PRE-PREPARE), and the second is in fact to determine the order that the operations must be processed (PREPARE). Therefore, upon receiving a REQUEST from the client, the primary creates the PRE-PREPARE message and send it to all cluster's nodes and waits for the PREPARE messageAs it is depicted in figure 4.1, the primary must be physically isolated from the other nodes of the cluster and, therefore, the system must need at least $3f+2$ PMs to be able to tolerate f faults. Nevertheless, as the load increases and more groups are added, this additional PMs becomes less significant.

In this implementation, the groups must be distributed across $3f+1$ PMs and, therefore, it necessary to pay a higher price to have elasticity and to accommodate peaks in demand (by adding more groups). It is the main disadvantage of having an algorithm that needs more replicas to tolerate the same number of faults (f). However, this CloudBFT version could be easier to implement in cloud environments, since it does not need to have a tamper-proof component in each PM.

The primary chooses the group to process the clients' requests based on the request's content. This CloudBFT implementation also uses TPC-C as data schema and each group is responsible for processing requests related to a each partition, however there are requests that must access more than one partition. Upon receiving the PRE-PREPARE message, the group nodes chosen by the primary to process the request sent a PREPARE to other nodes belonging to the same group, thereby ensuring that they are ready to start the execution of a new request.

After receiving the PREPARE message, the group chosen by the primary to process this requests, starts the agreement protocol by exchanging the COMMIT messages (see Figure) to decide which is correct state and, consequently, the correct operation that must be sent to the database nodes. In this version, each group node sends $3f$ messages and receives the same number. Thus, at the end of the agreement protocol, it is possible to ensure that the correct state was decided correctly by a majority of proper nodes ($3f+1-f$). After deciding the system state, each group node sends a COMMIT-DB message to $2f+1$ distinct database nodes containing operation and the message order. Then, each group node waits for $2f$ equal responses from the database nodes. With this response, the group node will generate the HTML, CSS and JavaSript necessary to build the webpage that will be displayed to the client.

In this version, the database nodes are also responsible for ensuring that the operations are executed successfully. It also has a voting system (see Figure 4.1) to

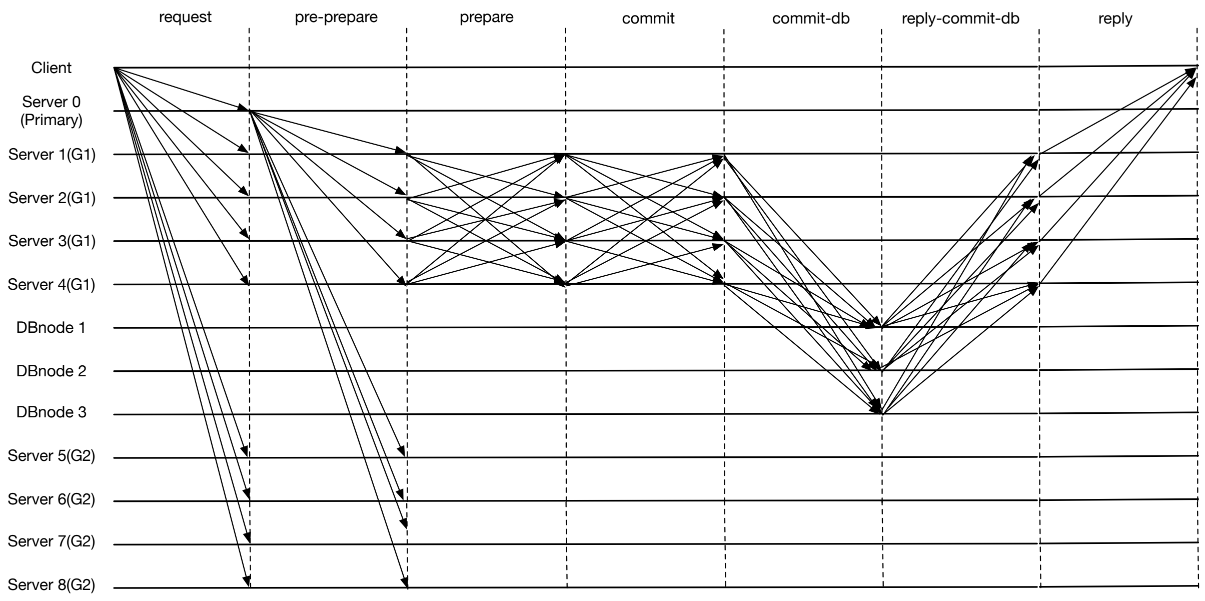


Figure 4.2: CloudBFT execution using a $3f+1$ BFT algorithm implementation

guarantee that all operations executed by database nodes were decided by at least $3f$ proper nodes. Furthermore, the voting system manages the concurrency between operations that needs to access more than one partition. Since the voting system executes the operations according to the order specified in the COMMIT-DB message (the order was previously decided by the primary and group nodes), the data tier must be composed by only $2f+1$ nodes. Upon executing the transaction on the database, each database node sends the response to the group node that previously has requested the operation execution.

CHAPTER 5

Evaluation

This chapter shows and discusses the results achieved by running an experimental setup of CloudBFT. Section one describes the experimental used for executing the tests. Section two discusses and compares the throughput achieved by CloudBFT and an unreplicated system, as well as the time spent in each CloudBFT's component. Section three discusses about the scalability achieved by CloudBFT and compares the maximum theoretical speedup against to the achieved by our system. Section four examines the results achieved by CloudBFT taking into account the elasticity and latency.

5.1 Setup

To measure the scalability of the system we resorted to experimental evaluation. The CloudBFT clients run in a loop requesting a web page, waiting for the reply and then returning to the beginning and requesting another page. For each page request, the server submits a TPC-C query to the database. Upon response from the database, the server generates a web page and replies back. We can simulate the page generation step with a sleeping period of 200 ms, because we do not care for the client side page rendering. In addition to these steps, the server must perform a number of extra operations, like voting, signing and verifying messages, using the TPM and the voting system.

CloudBFT system was implemented in Java and ran it on a private cluster, under different loads and with varying numbers of VMs. The private cluster sports 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 Chapter 3. 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 (See Figure 5.1). 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.

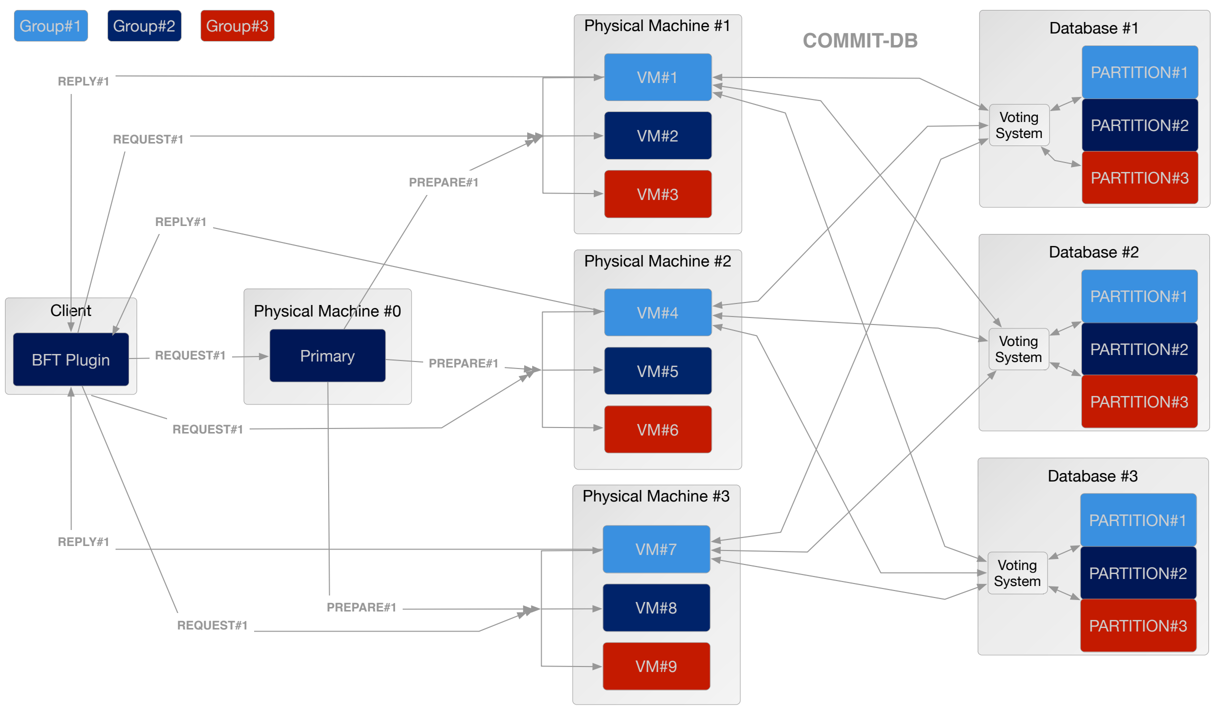


Figure 5.1: BFT execution of a request. In this figure the agreement step is not depicted for sake of simplicity.

The data tier is composed by $2f+1$ replicated databases in distinct PMs (See Figure 5.1). 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 the performance. The communication with the voting system of the database nodes is done through TPC sockets and all messages are encrypted and digitally signed.

5.2 Throughput

Since the main goal of this work was to build a scalable BFT system for the cloud, which is able to respond to increasing loads, we measure the throughput and the latency it obtains against simpler systems and against the theoretical limits.

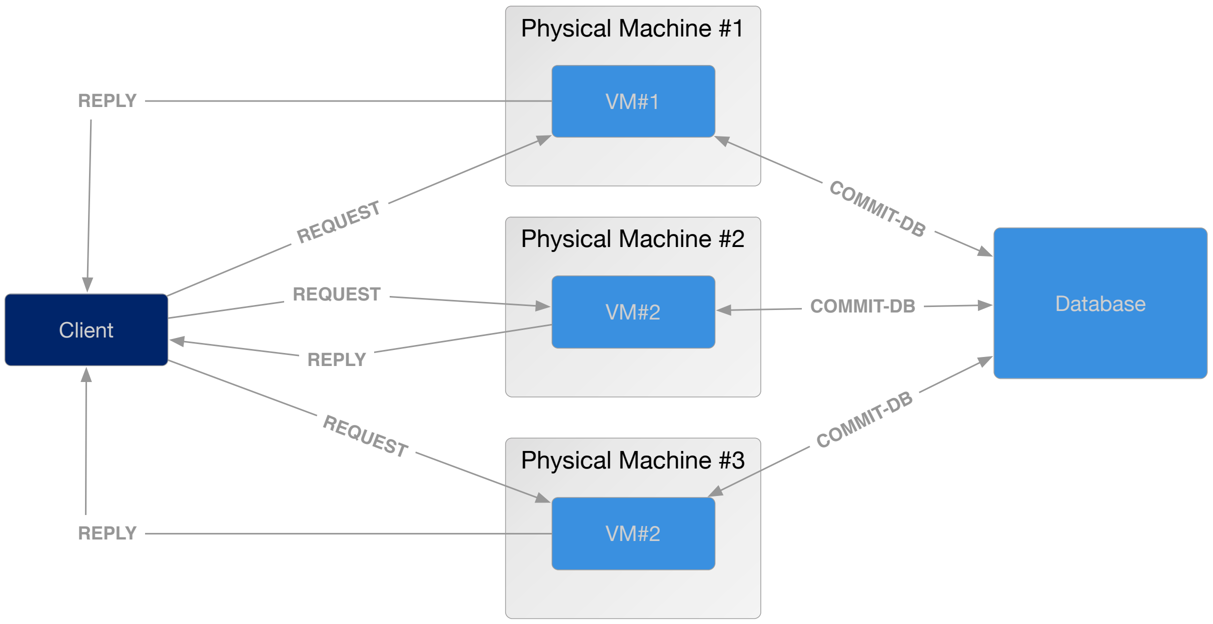


Figure 5.2: Execution of the unreplicated system with 3 replicated VMs.

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 (see Figure 5.2), 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 5.3 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.

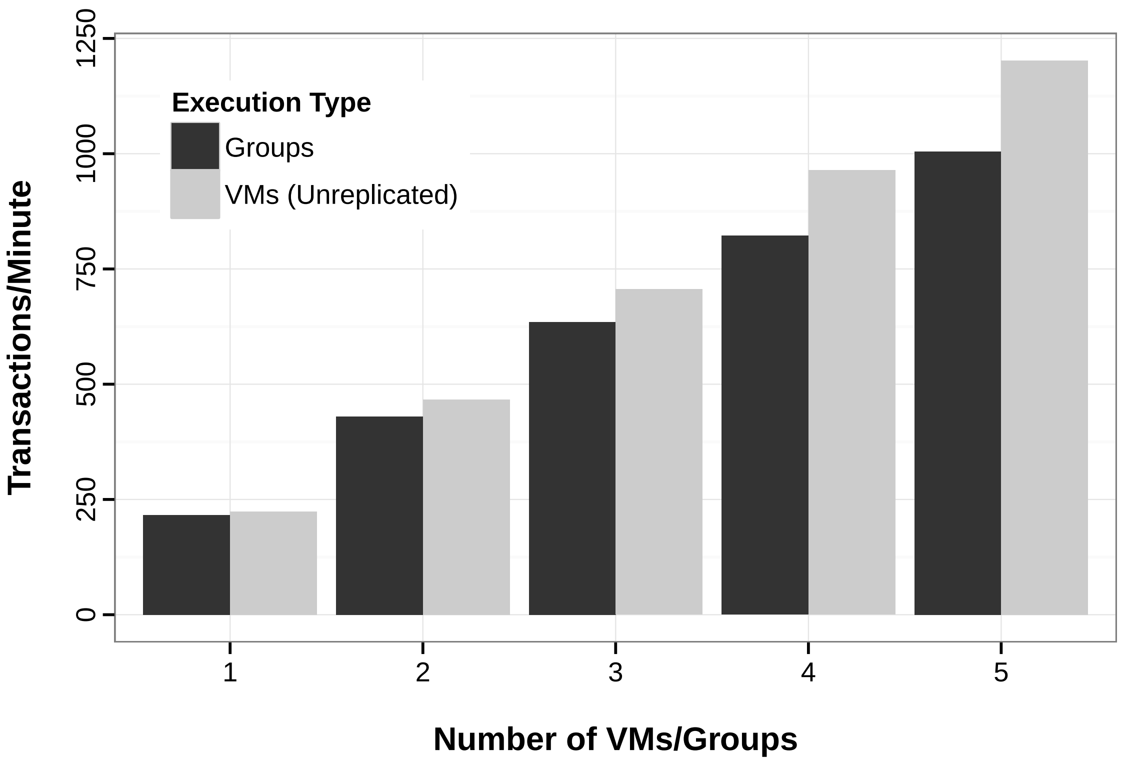


Figure 5.3: Throughput with Byzantine fault-tolerant groups *vs.* an unreplicated system.

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 5.4 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.

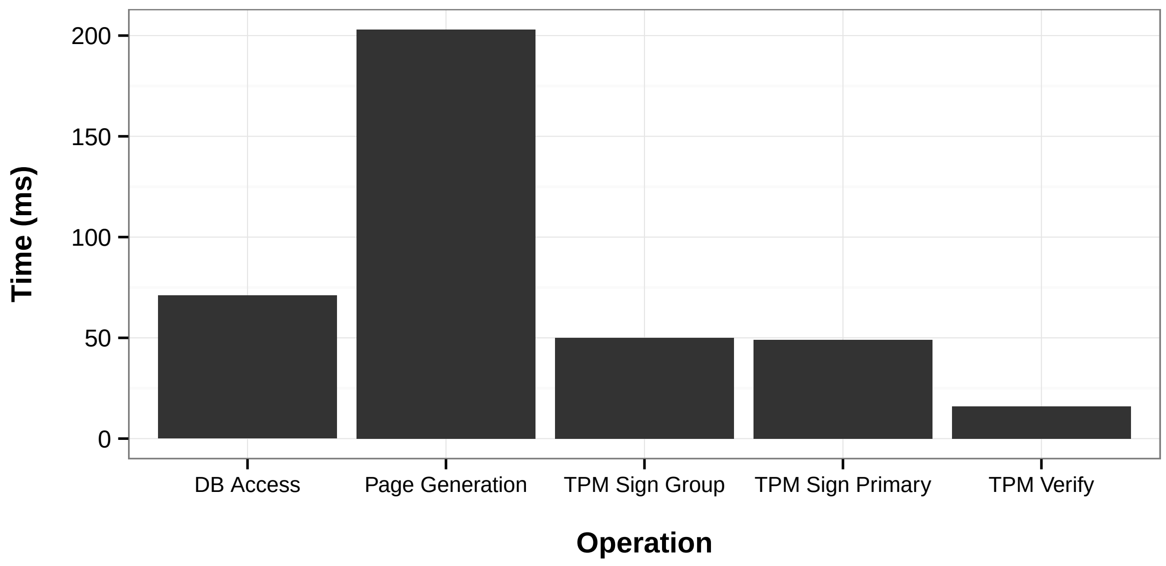


Figure 5.4: Time spent at each stage of the BFT pipeline.

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 5.4 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.

5.3 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.

Starting by the parallelizable requests, only the TPM signature done by the primary must run serially, 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 5.4 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.

Figure 5.5 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 partitions of data, the implementation can be said to scale within the limits imposed by the benchmark itself, that is, TPC-C. Multiple reasons concur to prevent real solutions from reaching a linear speedup. For example, depending on the load and on the number of groups, replicas can spend between 2 and 5% of their time waiting on the global lock that protects 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

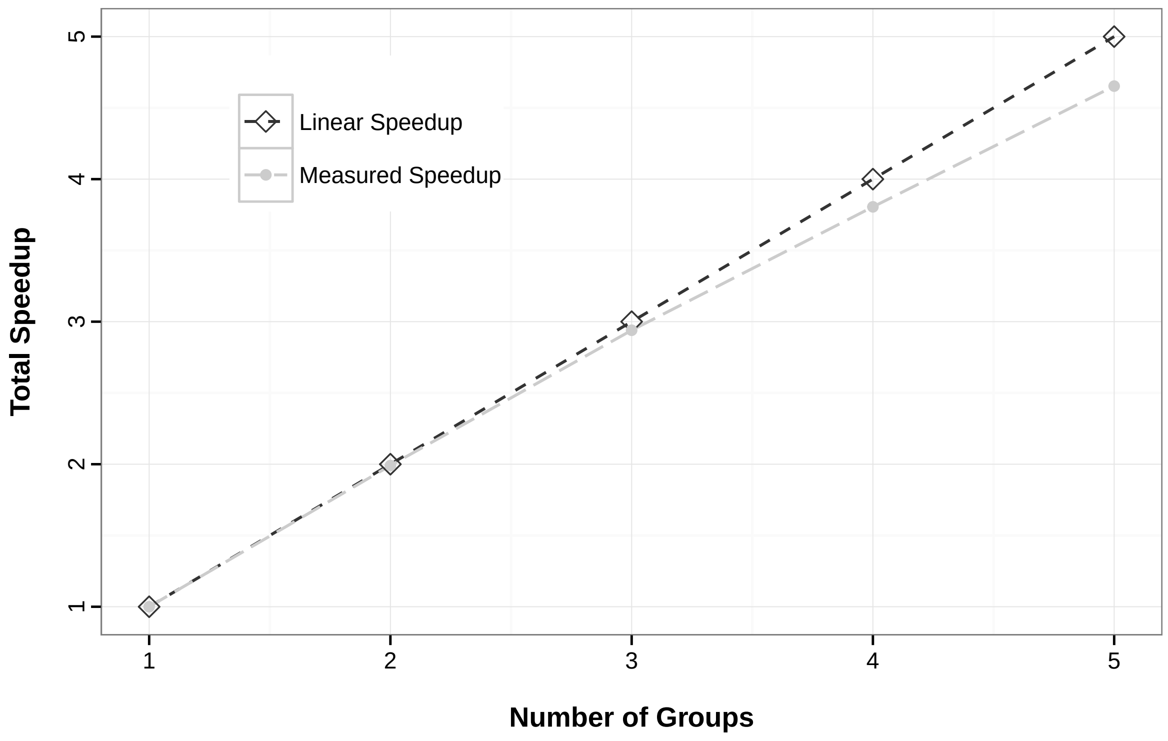


Figure 5.5: Total speedup (measured) *vs.* maximum theoretical speedup.

$710 + 1278 = 1988$ 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 5.1. 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.

$$\lim_{n \to \infty} S(n) < \frac{38900}{1988} \approx 19.57 \tag{5.1}$$

5.4 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 5.6 we increased the number of clients from 1 to 20 and measured the resulting throughput for systems with up to 5 groups. As we mentioned before, 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.

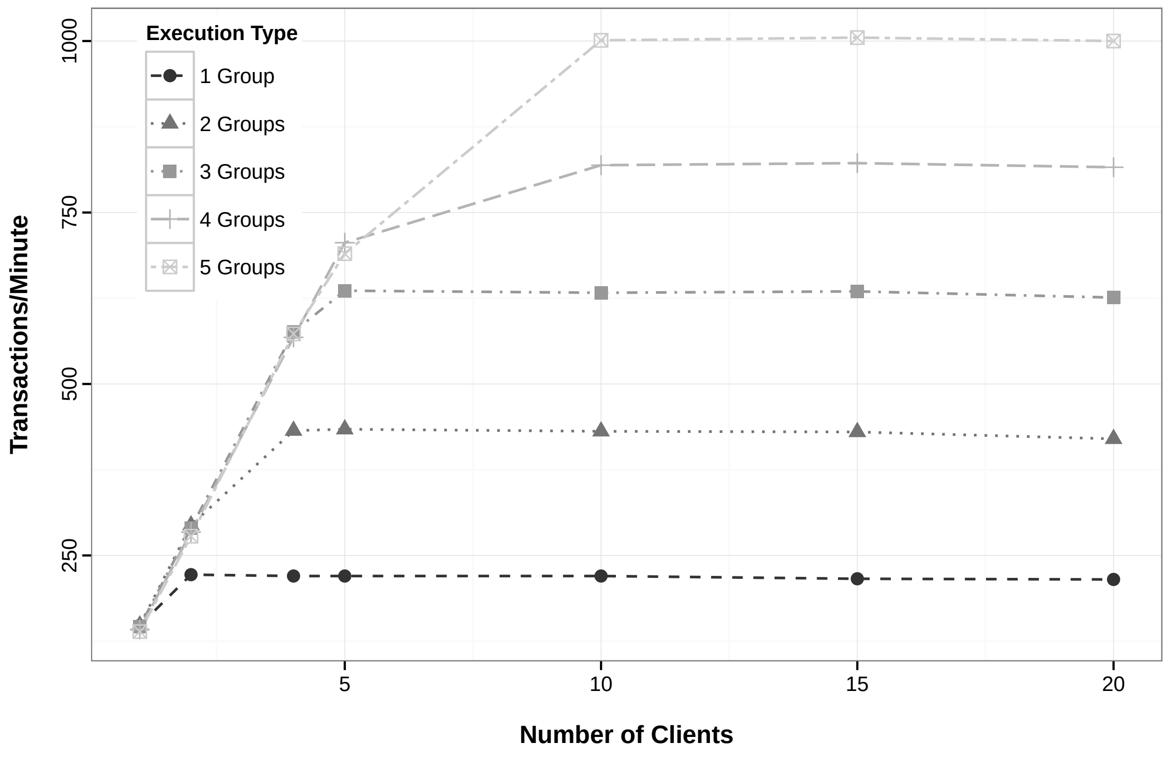


Figure 5.6: Measured throughput for an increasing number of clients, with up to five groups.

The results shown in Figure 5.6 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 start additional 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 if Figure 5.7.

In Figure 5.7 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 5.7 shows that it is possible to respond to a growing load

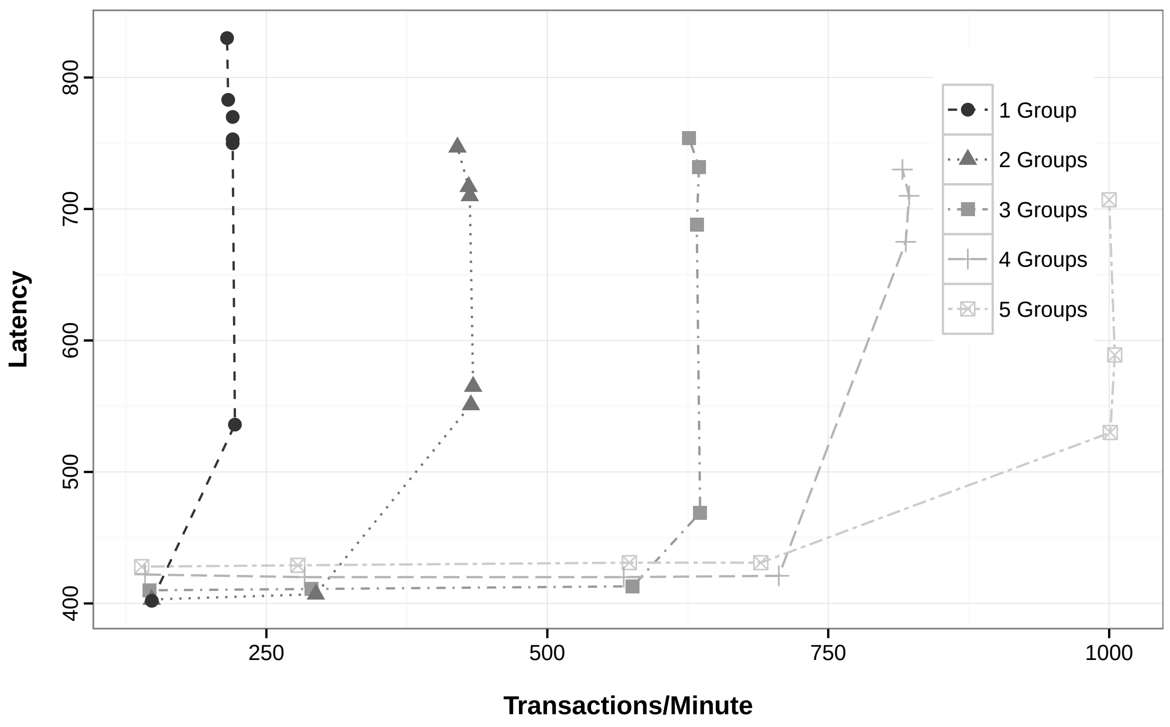


Figure 5.7: Latency of requests *vs.* throughput for systems with up to five groups.

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 [47]. To better understand the response time observed by clients, Figure 5.8 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 1000 transactions per minute.

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 5.9.

The cumulative distribution function of latencies, in Figure 5.9, 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.

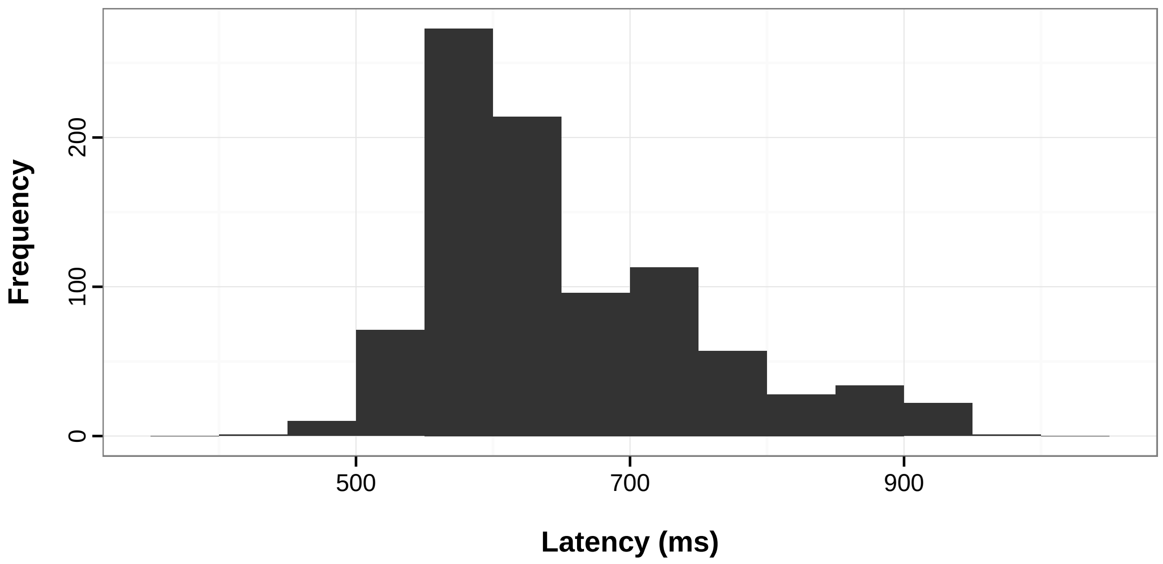


Figure 5.8: Distribution of latencies.

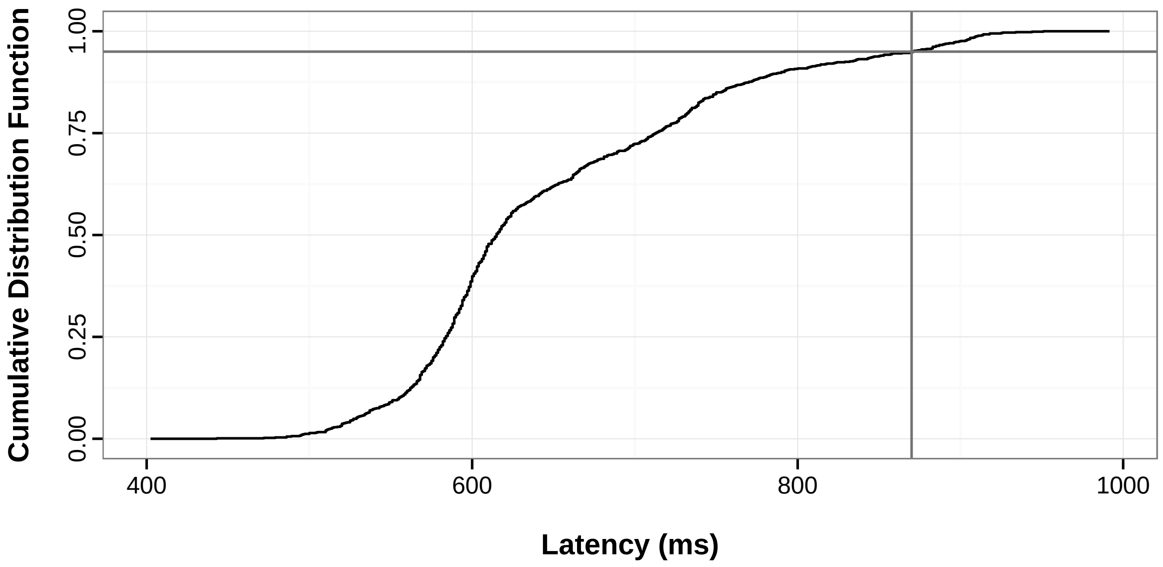


Figure 5.9: Cumulative distribution function of latencies.

CHAPTER 6

Conclusion

This work described a Byzantine fault-tolerant architecture, designed for cloud applications with critical services that may scale according to the computational power required for processing the clients' requests. Therefore, we demonstrated that CloudBFT is capable of taking advantage of cloud's elasticity, without compromising the dependability and consistency. The elasticity and scalability were achieved by creating groups of $2f + 1$ or $3 + 1$ (it varies according to CloudBFT's implementation) virtual machines running on distinct physical machines.

The distribution of virtual machines (groups) across different physical machines is required to avoid that hardware faults, faults caused by intrusions, and cross-VM attacks do not compromise the system. Thus, the system remains consistent (following the specified behavior) even if f virtual machines belonging to the same group were affected by any type of faults.

This work shows that CloudBFT can work with any Byzantine fault-tolerant algorithm with minimal changes. However, these changes must be made carefully, because otherwise the system's consistency and the Byzantine fault tolerance can be compromised. Furthermore, we demonstrate that is possible to have a three-tier system in cloud environments, where the client can trust on the response (webpage) received, since this response was surely decided by a reliable group of nodes.

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, the results show that, within reasonably large bounds, elasticity is possible 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.

6.1 Future Work

To further expand the work presented throughout this thesis, we intend to run more experiments using a larger cluster, to better understand the impact on speedup of the number of database partitions. In addition, these further experiments would allow us to understand better which are the system's components that quickly become the bottleneck when the processing power needs to increase and, therefore, we would be able to circumvent these bottlenecks with more concrete basis.

Although the experimental setup has been built as close as possible to cloud environments, it would be interesting to run these experiments on a real cloud environment, such as Amazon or Google Cloud Platform. This would allow us to perceive how the system would behave on the cloud, and consequently, improving it even more.

Our system keeps a small machine footprint and the processing required from each virtual machine is low. Nevertheless, since the cloud providers charge their clients according to the virtual machines configurations (CPU model, space and type of storage, etc.), as well as by the processing and by the data transferred over the network. It would be interesting to better understand the minimum required configurations of each component (primary, VMs of the group, database nodes) for deploying and executing our system, as well as the CPU and network usage of each component throughout the clients' requests processing. This would allows us to understand better where the system could be improved in terms of minimum requirements of the nodes (CPU, amount of memory, etc.) and resource utilization during the execution (CPU, network, etc.), thereby reducing the cost with the cluster and, therefore making the architecture presented in this work even more feasible.

The $3f+1$ implementation of our system was not fully tested and, therefore, one of our further goals is to test this version in a real cloud environment. This would allows us to better understand the real impact of the TPM on the system's performance.

6.2 Final Remarks

Taking into account the objectives that were defined at the beginning of this work, we can conclude that all goals were fulfilled successfully. CloudBFT demonstrated successfully that is possible to have a system capable to tolerate Byzantine faults in cloud environments, without compromising the scalability and elasticity even under a relational data model. We believe that this may help a wide spectrum of critical services intended to be deployed in the cloud.

Bibliography

[1] K. V. Vishwanath and N. Nagappan, "Characterizing cloud computing hardware reliability," in *SoCC* (J. M. Hellerstein, S. Chaudhuri, and M. Rosenblum, eds.), pp. 193–204, ACM, 2010.

[2] K. Birman, G. Chockler, and R. van Renesse, "Toward a cloud computing research agenda," *SIGACT News*, vol. 40, pp. 68–80, June 2009.

[3] X. Meng, C. Isci, J. Kephart, L. Zhang, E. Bouillet, and D. Pendarakis, "Efficient resource provisioning in compute clouds via vm multiplexing," in *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, (New York, NY, USA), pp. 11–20, ACM, 2010.

[4] 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*, pp. 228–237, IEEE Computer Society, 2010.

[5] 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*, VEE '09, (New York, NY, USA), pp. 31–40, ACM, 2009.

[6] 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.

[7] Amazon Web Services, Inc, "Amazon web services, cloud computing: Compute, storage, database." `http://aws.amazon.com`. Accessed: January, 2014.

[8] Rackspace, Inc, "Rackspace solutions and documentation." `http://www.rackspace.com/cloud/`. Accessed: January, 2014.

[9] Google, Inc, "Google cloud platform." `https://cloud.google.com`. Accessed: January, 2014.

[10] Microsoft, "Windows azure solutions and documentation." `http://www.windowsazure.com/en-us/`. Accessed: January, 2014.

[11] SYS-CON Media Inc., "Twenty experts define cloud computing." `http://cloudcomputing.sys-con.com/node/612375/print`, 2008. Accessed: January, 2014.

[12] I. T. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud computing and grid computing 360-degree compared," *CoRR*, vol. abs/0901.0131, 2009.

[13] N. I. of Standards and Technology, "The nist definition of cloud computing," september 2011.

[14] VMware, Inc, "A performance comparison of hypervisors." `http://www.vmware.com/pdf/hypervisor_performance.pdf`. Accessed: January, 2014.

[15] XenSource, Inc, "Performance comparison of commercial hypervisors." `http://www.cc.iitd.ernet.in/misc/cloud/XenExpress.pdf`. Accessed: January, 2014.

[16] J. Li, Q. Wang, D. Jayasinghe, J. Park, T. Zhu, and C. Pu, "Performance overhead among three hypervisors: An experimental study using hadoop benchmarks," in *BigData Congress*, pp. 9–16, IEEE, 2013.

[17] 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*, pp. 314–327, IEEE Computer Society, 2006.

[18] IBM Corporation, "Ibm x-force 2010mid-year trend and risk report." `http://public.dhe.ibm.com/common/ssi/ecm/en/wgl03003usen/WGL03003USEN.PDF`. Accessed: January, 2014.

[19] NIST, "Guide to security for full virtualization technologies," Tech. Rep. 800125, NIST, Jan. 1994.

[20] salesforce.com, inc, "Salesforce1 platform: Trusted application development platform - salesforce.com." `http://www.salesforce.com/platform/overview/`. Accessed: January, 2014.

[21] Heroku, "Heroku | cloud application platform." `http://www.heroku.com`. Accessed: January, 2014.

[22] Google, Inc, "Google app engine: Platform as a service." `http://developers.google.com/appengine/`. Accessed: January, 2014.

[23] salesforce.com, inc, "Crm and cloud computing to grow your business - salesforce.com." `http://www.salesforce.com`. Accessed: January, 2014.

[24] Google, Inc, "Google apps for business." `http://www.google.com/enterprise/apps/business/`. Accessed: January, 2014.

[25] Workday, Inc., "Workday - enterprise cloud for hr and finance." `http://www.workday.com`. Accessed: January, 2014.

[26] J. Gray and D. P. Siewiorek, "High-availability computer systems," *Computer*, vol. 24, pp. 39–48, Sept. 1991.

[27] Jean-Claude Laprie, "Dependable computing: From concepts to design diversity," in *Proceedings of the IEEE*, pp. 629–638, 1986.

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

[29] M. Biely, P. Robinson, and U. Schmid, "Weak synchrony models and failure detectors for message passing (k -)set agreement," in *OPODIS* (T. F. Abdelzaher, M. Raynal, and N. Santoro, eds.), vol. 5923 of *Lecture Notes in Computer Science*, pp. 285–299, Springer, 2009.

[30] 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.

[31] T. Wood, R. Singh, A. Venkataramani, P. J. Shenoy, and E. Cecchet, "ZZ and the art of practical BFT execution," in *EuroSys* (C. M. Kirsch and G. Heiser, eds.), pp. 123–138, ACM, 2011.

[32] 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.

[33] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz, "Attested append-only memory: Making adversaries stick to their word," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles (21st SOSP'07)*, (Stevenson, Washington, USA), pp. 189–204, ACM SIGOPS, Oct. 2007.

[34] 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 International Symposium on Reliable Distributed Systems (23rd SRDS'04)*, (Florianopolis, Brazil), pp. 174–183, IEEE Computer Society, Oct. 2004.

[35] 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.

[36] Trusted Computing Group, "TPM main specification," Main Specification Version 1.2 rev. 85, Trusted Computing Group, Feb. 2005.

[37] Brickell, Camenisch, and Chen, "Direct anonymous attestation," in *SIGSAC: 11th ACM Conference on Computer and Communications Security*, ACM SIGSAC, 2004.

[38] H. Ge, "A method to implement direct anonymous attestation," *IACR Cryptology ePrint Archive*, vol. 2006, p. 23, 2006.

[39] X. Chen and D. Feng, "Direct anonymous attestation for next generation TPM," *JCP*, vol. 3, no. 12, pp. 43–50, 2008.

[40] BuiltWith, "Framework technologies web usage statistics." `http://trends.builtwith.com/framework`. Accessed: January, 2014.

[41] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, pp. 120–126, Feb. 1978.

[42] E. B. Barker, D. Johnson, and M. E. Smid, "Sp 800-56a. recommendation for pair-wise key establishment schemes using discrete logarithm cryptography (revised)," tech. rep., Gaithersburg, MD, United States, 2007.

[43] H. Krawczyk, M. Bellare, and R. Canetti, "Hmac: Keyed-hashing for message authentication," 1997.

[44] 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.

[45] Amazon, Inc., "Amazon store." `http://amazon.com`. Accessed: January, 2014.

[46] "Bft-smart: High-performance byzantine fault-tolerant state machine replication.." `https://code.google.com/p/bft-smart/`.

[47] 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.