

1

The Shuttle Architecture

This chapter describes the overall system architecture of Shuttle and outlines its central functional components. The main design goal is to help **CSP! (CSP!)** customers to recover from intrusions in their applications deployed in **PaaS! (PaaS!)**. We consider three actors: the **CSP! (CSP!)**, which ~~provide~~ provides the platform, the *tenants*, whose ~~often are also known as system administrators, that~~ deploy their applications in the platform, and the *users*, ~~which~~ whose access the applications. Shuttle is a service designed to be offered by a CSP!.

We introduce the main requirements in Section 1.1 and describe a generic **PaaS! (PaaS!)** architecture in Section 1.2. The remaining sections describe the architecture of Shuttle and discuss the main design choices. mais info aqui?

1.1 Requirements

This thesis addresses the problem of providing an intrusion recovery service for applications deployed in **PaaS! (PaaS!)**. Our overall goal is to *make PaaS! applications operational despite intrusions*. também me dá essa ideia mas é porque saltamos o capítulo 2, o paragrafo ficou aqui para relembrar e re-introduzir o leitor ao objectivo More precisely, we aim to create a service, named Shuttle, to help **PaaS!** tenants to recover from the following problems in their applications:

- *Software vulnerabilities*: non-authorized users compromise state by exploiting software vulnerabilities ~~, e.g., that~~ allow invalid requests to be executed.
- *Malicious or accidentally corrupted requests*: users, authorized or not, compromise the application state accidentally or intentionally issuing valid requests.

For instance, two common attacks that can be used to compromise application state consist in: (1) attackers stealing valid users' credentials and using them to access their data; and (2) doing a **SQL! (SQL!)** Injection attack by mixing **SQL!** meta-characters with normal input and doing otherwise invalid queries to the database. Both attacks can be performed using apparently valid requests. Consequently, many prevention mechanisms fail to block them.

In order to achieve the above goals, the service shall meet the following requirements:

- *Remove intrusion effects*: Remove corrupted data at operating file system, database and application levels in the application containers and update affected legitimate actions. exacto, consigo lidar com o sistema de ficheiro, que é o estado persistente do sistema operativo porque re-instacio a aplicação.

- *Remove selected malicious actions:* Help tenant to track the intrusion producing the set of actions affected by an externally provided list of malicious actions.
- *Support software update:* After recovery, the application state has to be compliant with the new version of the software.
- *Recover without stopping the application:* Recover the application without exposing users to application downtime. **na prática este objectivo não foi conseguido a 100% mas, mesmo que o júri me dê nas orelhas, acho que aqui devo ter os objectivos ideais do projecto sem ajustá-los ao que consegui**
- *Determinism:* Despite concurrent re-execution of requests, the result of re-execution is the same as the result of first execution if the application source code and requests remain equal.
- *Low runtime overhead:* The recording of operations or state for recovery purposes should have a negligible impact in the runtime performance.
- **NoSQL! database snapshot:** **NoSQL!** databases will have to be extended to support database snapshots, in order to reduce the recovery time.
- **PaaS! integration:** The source code of the application shall remain unmodified as much as possible. **PaaS!** developers do not need to install or configure Shuttle. Shuttle is built in a generic manner and it is reused in each deployed application.

Shuttle shall *support software updates* to prevent future intrusions and allow operators to try new configurations or software versions without effects in the application behavior perceived by users.

1.2 Platform as a Service

PaaS! (PaaS!) is a cloud computing model for automated configuration and deployment of applications onto the cloud infrastructure [? ? ? ?]. **PaaS!** enables developers to develop and deploy web applications into production fast by abstracting many details of the underlying infrastructure. Developers access the infrastructure resources, such as storage, through a set of services. These services are often pay-per-usage. **PaaS!** provides a deployment environment for a set of languages.

Applications are deployed in one or more application servers, e.g., Java Tomcat or Javascript NodeJS. *Containers* [?] hold these application servers and provide the required isolation level between the various applications. The word container is often used to refer to lightweight in-kernel resource (CPU, memory and device) accounting, allocation and isolation mechanisms like the *Linux control groups* [?]. These mechanisms isolate the process, network and file system used by applications that share the same operating system. They can run either directly on the host operating system or in a virtual machine. In this document, we use the word *container or instance* to describe an isolated deployment unit that can be allocated from a resource pool by [an](#) orchestration engine. The deployment unit is created using an image and has storage attached. Therefore, our concept of container includes not only *Linux control groups* like systems but also bare metal servers and guest operating systems running on top of hyper-visors, e.g., Xen [?], KVM [?]. Containers are managed directly or through an orchestration or IaaS system (e.g., OpenStack [?], **AWS! (AWS!) EC2! (EC2!) [?]**, Eucalyptus [?], Omega [?]). Containers have one or more associated storage. When the container loads up, it loads an image onto its storage. The image contains, at least, the operating system and the **PaaS!** system in order to deploy the application in the container.

In order to let Shuttle as generic as possible, we consider the following components of a minimal **PaaS!** architecture ([Fig-Figure 1.1](#)):

- **Load balancer:** Routes user requests based on application location and container load.
- **Instance controller:** Collects the container metering data and performs the configuration, tear-up and tear-down of containers in the instance.
- **Cloud controller:** Manages the tear-up and tear-down of containers.
- **Metering and billing:** Retrieves the metering data from each container. The load balancer uses this information to perform request routing while the cloud controller automatically decides when to scale.

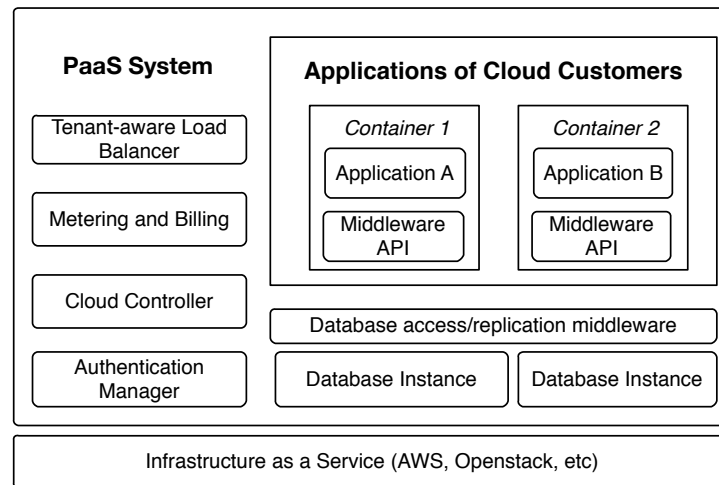


Figure 1.1: Generic PaaS! architecture

- **Containers:** The isolated environment where applications run.
- **Cloud Instances:** The guest operating systems or bare-metal machine where the containers run.
- **Authentication manager:** Provides user and system authentication.
- **Database instance:** A single **DBMS!** (DBMS!) shared, or not, between multiple applications. Most of database middleware are built-on multiple containers to provide scalability and replication.
- **Authentication manager:** Provides user and system authentication.

CSP! (CSP!) also deploy the database management systems in containers. Their data is accessed as a service by the developers. Most of the applications deployed in **PaaS!** are designed to scale horizontally, i.e., to scale adding more containers. Therefore, the database and/or the session cookies often maintain the application state. The **PaaS!** systems are often integrated with code repositories and software development tools reducing the time deploy of applications in cloud environments. Users may access the website connecting to the load balancer via **HTTPS!** (**HTTPS!**), which will decrypt the **SSL!** (**SSL!**) session and forwards-forward the unencrypted requests to application containers. As the traffic increases, the load balancer may become a performance bottleneck if the system does not provide enough resources to handle the user traffic.

We assume applications to store their persistent state only in databases. Shuttle's architecture can be extended to encompass object storage, for instance **AWS!** (**AWS!**) **S3!** (**S3!**). We do not consider a possible state stored in the filesystem because **PaaS!** applications are supposed to be scalable, thus the instances file system is frequently destroyed.

1.3 Shuttle Overview

Shuttle is an intrusion recovery service for **PaaS!**. It recovers from intrusions on software domain due to software flaws, corrupted requests, input mistakes, ~~corrupted data and suspicious intrusions~~ and corrupted data in **PaaS!** containers 1.2(Section 1.2). While previous works (Chapter ??) aimed to recover applications supported by a single database, Shuttle targets **PaaS!** applications deployed in multiple instances and backed by **NoSQL!** databases. Since typical **PaaS!** applications are designed to support high usage loads, our main contribution is a scalable intrusion recovery service that is transparent for application developers.

Shuttle is an automatic recovery mechanism based on the record-and-replay approach. Applications supported by Shuttle can operate in one of two states: *normal execution* and *recovery*. During

normal execution, Shuttle records the data required to recover the application afterward: it does periodic database snapshots, logs user requests and database operations. When an intrusion is identified, tenants use Shuttle to recover their applications starting the recovery phase.

~~Processes~~ The processes described in Section ?? lead us to define *how to remove the intrusion effects* and *how to recover a consistent state*. During the *recovery phase*, Shuttle removes the intrusion effects creating a branch of the system execution in which it loads a snapshot that contains an application state before the intrusion began. It builds a consistent state replaying (re-executing), in the new branch, the legitimate requests logged during the *normal execution*, performing either full or selective replay (Section 1.5.8). In the meantime, the incoming requests are executed in the previous branch. When ready, it sets the new branch as the single execution branch.

Shuttle aims to be integrated by **CSP!** (**CSP!**) into their **PaaS!** architecture as a novel service. Services provided in **PaaS!** are expected to be well-tested and available without setup because they are offered by **CSP!** and shared by multiple tenants. Our approach hides the Shuttle implementation and operation within the database and load-balancing **PaaS!** services. Shuttle components can be shared by multiple clients but the data of each client remains isolated. For sake of simplicity, we ~~consider~~ present Shuttle considering a single tenant implementation.

We consider a minimal **PaaS!** architecture to let Shuttle as generic as possible. We consider a client-server model in which clients access applications using the **HTTP!** (**HTTP!**) protocol ¹. **HTTP!** requests are received by a load balancer that forwards them to web/application servers, which access a shared database. PaaS components are represented with solid line in Figure 1.2, while Shuttle components are represented with dashed line. **PaaS!** platforms with Shuttle have the following components:

- *Proxy*: Logs every **HTTP!** user requests, adds an unique mark to its header and forwards it to the load balancer. The proxy functionality might be part of the load balancer but conceptually it is a different component.
- *Load balancer*: Routes requests to different application servers taking into account their load (part of the **PaaS!** platform).
- *Application servers*: The application (or web) servers are the components of the **PaaS!** platform that run the application logic. This logic uses a library to access the database service. Shuttle uses a *database client interceptor* mechanism in this library to log the data items accessed per request.
- *Database instances*: A set of database servers used to store the application persistent state. Shuttle includes in each instance a *database proxy* that logs the requests that accessed each data item and determines the dependencies between requests.
- *Shuttle storage*: A scalable storage component that stores requests, responses and metadata.
- *Manager*: Retrieves dependencies and coordinates the recovery process.
- *Replay instances*: A set of **HTTP!** clients that read previously executed requests from the Shuttle storage and invoke the application servers to re-execute the requests during the recovery process. The manager coordinates the worker instances.

The *Shuttle storage* keeps the content of the user requests and responses. Although we do not consider this aspect in the architecture, this store can be replicated to a remote site to allow tolerating catastrophic failures in a datacenter.

We consider the Shuttle components to be part of the trusted computing base since their integrity and availability are critical to recover the application state. We assume that intrusions tamper the application data, which is stored in the database, not the snapshots neither the stored requests.

Unlike previous works, our design encompasses distributed databases (**NoSQL!**). These databases are designed to scale horizontally. Therefore, Shuttle can also be scaled by adding more database instances.

PaaS offerings are supported by a computing infrastructure, often provided as a service (IaaS model), able to scale the application allocating new instances on-demand or automatically, to maintain the quality of service despite demand oscillations. This elasticity allows allocating replay instances

¹ Shuttle also supports HTTPS by ending the connections at the proxy.

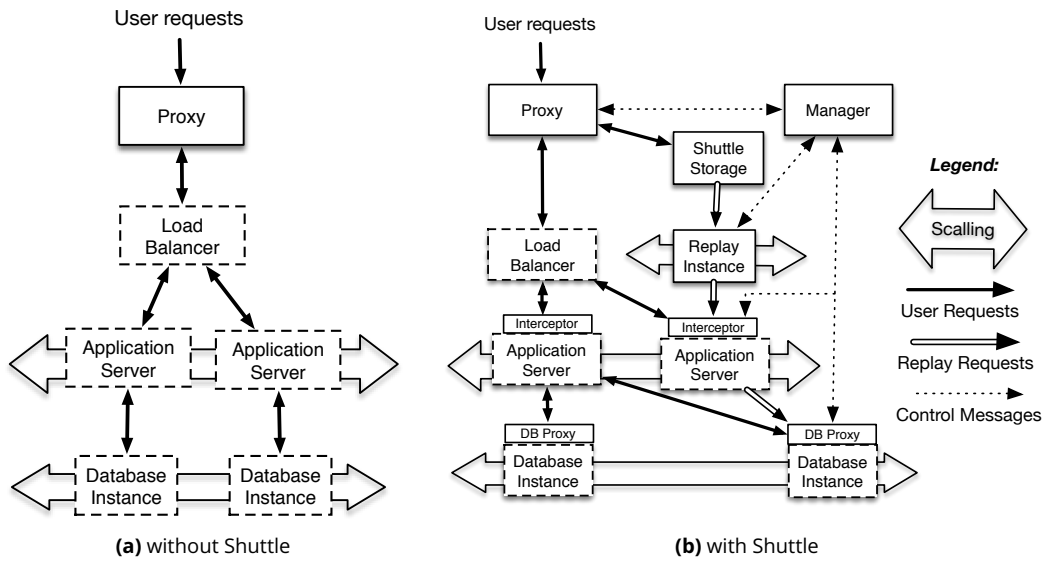


Figure 1.2: Shuttle service architecture: The dashed line components are part of the **PaaS!** architecture. The proxy logs the user requests into the Shuttle storage. The manager coordinates the recovery process where the replay instances replay the user requests.

and to scale the application to attend the requests issued by them during the recovery process. Due to the common pay-per-usage model, these resources are paid only when a recovery process occurs. The remaining cost of the service comes from storing client requests and database snapshots. Our design aims to optimize the available resources to reduce the recovery period and costs.

1.4 Normal Execution

Shuttle logs the data it needs to recover applications during the normal execution phase: user **HTTP!** requests, application **HTTP!** responses, database items accessed by each request and sequence of operations to each database item (Figure 1.3). In this section, we describe the normal execution phase following the path that a request takes to be processed.

Coloco subcapítulos ou não justifica? The proxy intercepts all user **HTTP!** requests, except those to static contents (e.g., images), and adds a new header field named **SRD!** (**SRD!**). Each **SRD!** contains three subfields: **RID!** (**RID!**), which is a unique timestamp; *Branch* and *Snapshot*, which define, respectively, the database branch and snapshot (Section 1.5.2) and a *restraint* flag, which is used to support

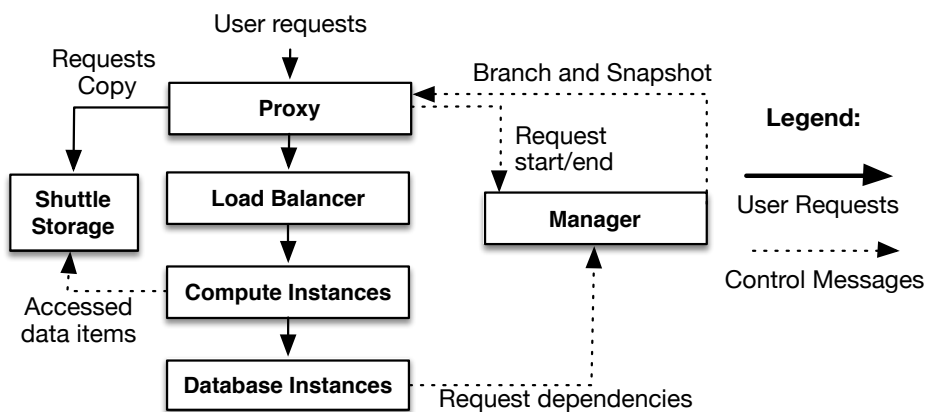


Figure 1.3: Interaction between components during the normal execution

runtime recovery (Section 1.5.6).

The proxy also intercepts every application response, associates the response with the original request and adds a new timestamp to track the ending of the request execution. Requests, responses and their timestamps are stored in the *Shuttle Storage* using asynchronous I/O, which permits the operations to proceed before the transmission has finished.

Requests are sent to the load balancer, which forwards the requests through the application instances according to their usage.

The application instances invoke the database service using the database client library. The database client library intercepts the operations, logs the accessed data items per request and stores this information in the *Shuttle Storage*. The database invocation is tracked at client side because the database may not be available or operations may fail.

On each database instance, the database proxy logs the operations' **RID** and type. The sequence of operations to a data item defines its *operation list*. Periodically, each database instance iterates the operation list of every data item to establish the dependencies between requests. Shuttle also performs snapshots periodically. The snapshot operation stores a new version of data item on the next write operation (Section 1.5.2).

In summary, every request-response pair is timestamped and logged by the proxy, the application instances the accessed data items per request, the database logs the sequence of operations per data item. The sequence of operations of each data item is kept in the database instance in which the data item is stored. The remaining data is stored in the *Shuttle Storage*, which can be located, or replicated, in a remote site to prevent catastrophic disasters rebuilding the application state using the requests and a previous snapshot. The manager retrieves, asynchronously, the requests' start and end timestamps, which are sent by the proxy, and their dependencies, which are collected by the database instances. Shuttle uses the information retrieved to generate the request dependency graph (Section 1.5.3). **vou deixar as respostas**

1.5 Recovery

The intrusion recovery process consists of three steps. The first step concerns the intrusion detection, in which tenants detect intrusions, suspicious behaviors or software flaws. Tenants may use automated tools such as **IDS! (IDS!) [?]** to detect intrusions. The second step is vulnerability management in which vulnerabilities are identified, classified and mitigated. This work assumes that tenants identify the malicious requests (the subsequence of actions $A_{intrusion}$ whereby the attacker compromises the application) correctly and modify or remove them. Alternatively, tenants can provide an updated and vulnerability-free software version (Section 1.5.1). In addition, Shuttle provides several methods to help tenants to identify the malicious requests: determine the set of requests that accessed a set of affected database entries after an estimated intrusion moment; group requests by user-session; compare database versions to check if the vulnerabilities are correctly mitigated (Section ??).

The third step consists in removing the intrusion effects. Intrusions affect the application integrity, confidentiality and/or availability (Section ??). ~~To recover~~ Recovery from confidentiality violations is out of the scope of this document. However, we argue that the design of the applications should encompass cryptography techniques which may reduce data relevance and protect the data secrecy [?]. Shuttle aims to recover applications from integrity violations, which often harm the availability. Shuttle can accomplish some of the goals of intrusion tolerance, keeping the application availability. Applications can keep providing a, possibly degraded but adequate, service during the intrusion recovery. Incoming requests are executed while the recovery process occurs without externalization to users (Section 1.5.6). In addition, Shuttle reduces the system downtime by reducing the time to recover when intrusions happen. Shuttle does not ~~replaces~~ replace the intrusion prevention, detection and tolerance mechanisms, which are the primary lines of defense against attacks. **este paragrafo secalar deveria ir para o overview**

In order to remove the intrusion effects, Shuttle loads a database snapshot, which is selected by the tenant. The selected snapshot shall be previous to the intrusion moment in order to replace the value of every data item by a non-tampered value. Shuttle *manager* orders the **PaaS!** controller to launch a new set of application instances and deploys an updated source code version, which may include code fixes. Then, the manager orders the database instances to load the selected snapshot

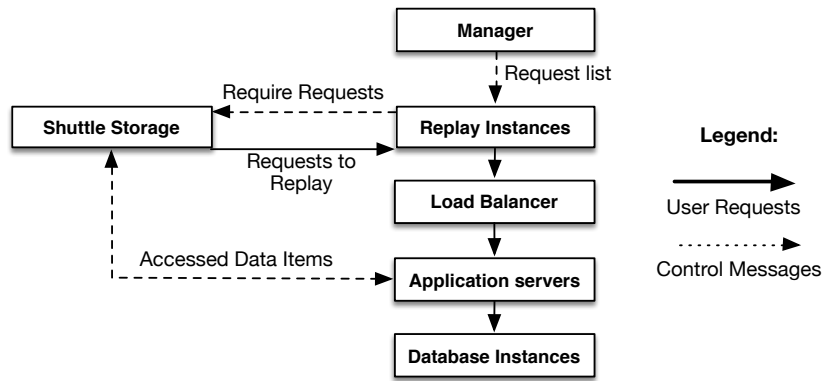


Figure 1.4: Interaction between components during the recovery process

(Section 1.5.5). The application is intrusion-less now that the snapshot is previous to the intrusion and the application is redeployed on new instances.

After, the manager initiates a set of *replay instances* to replay the legitimate requests of the sequence of legitimate actions that happen after the snapshot, $A - A_{snapshot} - A_{intrusion}$. The replay instances retrieve a list of requests to replay and get the requests **HTTP!** package from the *Shuttle Storage* (Figure 1.4). Most **of PaaS!** systems scale automatically and horizontally, i.e., they increment or decrement the number of containers based on the measurements of the containers usage. Therefore, the application-logic and data tiers scale to attend the requests from the replay instances, increasing the recovery speed.

The database separates the versions used by the replayed requests and the new requests, preventing the application from exposing a downtime. After the recovery process, the new requests are also **forward forwarded** to the recovered database version (Section 1.5.6).

The main version of Shuttle loads a previous database snapshot and replays every legitimate user request. An algorithm concerning selective replay **using taint via replay** is introduced in Section 1.5.8.

In the following sections, we discuss each of key process of the recovery phase in further detail.

1.5.1 Intrusion and vulnerability correction

The recovery process starts when intrusions are detected or the application software requires an update. Intrusion detection is out of the scope of this work. We assume that tenants, or system operators, detect one or more intrusions with the following sources:

1. User request (e.g. stolen user session)
2. External action: actions not logged by the proxy (e.g. ssh connection to the instances)

Attacks may:

1. Tamper the database (e.g. adding new entries)
2. Tamper the container (e.g. changing the deployed application in the container)

Shuttle supports the following actions to fix the exploited vulnerabilities:

1. Update the application software
2. Identify a set of tampered database entries
3. Add, modify or remove logged requests
4. Launch cleaned database or application server instances

Shuttle removes these effects of malicious actions redeploying the application in new containers and rolling back the database to a snapshot previous to the intrusion.

Attackers may use external actions to perform the intrusion, for instance gaining control of the instance to modify the database files. These actions are not recorded by Shuttle therefore they are not replayed and the application recovers a consistent state. We analyze several intrusion scenarios in Section 3.3.

If tenants update the application software, they have to ensure that the application's interface remains compatible with the requests that will be replayed. Alternatively, tenants may update the ~~database data items~~ entries in the database and provide a script to modify the requests make them compatible with the new **API! (API!)**.

If the database is tampered using user requests, the tenant has to identify the malicious user requests. In addition, tenants can provide the set of suspicious database entries to Shuttle and it will resolve the set of requests that accessed the suspicious items after the estimated intrusion moment. Knowing the suspicious requests, the tenants shall use Shuttle to add, modify or remove the past requests to remove accidental or malicious behaviors.

In summary, at the beginning of the intrusion recovery process, tenants shall ensure that:

1. The software is correct: previous flaws are fixed, its **API!** is compatible with the requests and its behavior is the expected.
2. The requests, which are selected to replay, are legitimate and their dependencies are correct.
3. The estimated intrusion moment is previous to the intrusion moment (the selected snapshot is intrusion free).

1.5.2 Snapshot

professor, pode enviar pelo menos esta parte do trabalho ao seu colega do brasil para que ele a reveja? quase todos os conceitos deste capitulo estao ligados ao trabalho dele...

Shuttle needs to remove ~~intrusions~~ intrusion effects. In Section ??, we presented two mechanisms to do so: record the data item values or compensate the malicious actions. The first makes a copy of the data item value at a certain instant t , implying more storage resources. The ~~further~~ second applies compensating actions to each action over the data item value after the instant t , which requires more computation resources to invert every action after t . However, the compensation mechanism requires the knowledge of the actions that revert the effects of the malicious actions. Moreover, if a malicious action is not recorded, then compensation mechanisms do not revert its effects. Since the set of operations is unknown and Shuttle aims to remove all intrusion effects, we perform snapshots by recording the value of the database items at a certain instant t (first mechanism).

A snapshot is a complete set of versions of every data item in the system from which data values can be read but to which updates are not made será que a 2ª parte da frase (a partir de "from which" inclusive) faz sentido? : fui buscar a definicao com base na dissertacao de doutoramento mas acaba por ser irrelevante. Like checkpoints, snapshots save the application persistent state at a certain moment tenho medo que algum purista diga que um checkpoint é apenas uma marca num log e que o snapshot é a cópia dos dados... não sei, é um pouco ambiguo e discutivel. Unlike the selective undo approach, which only reverts the tainted data items (Section 1.5.8), the full replay approach loads a snapshot previous to the intrusion instant. This approach reverts every database item and removes the effects of any action that occurred after the snapshot creation.

The duration of the recovery process is mainly defined by the number of requests previous to the intrusion of the set, A_{before} . The snapshot mechanism avoids to replay every request from the beginning of the application, which can take too long. Shuttle requires not only a snapshot but also every action posterior to the ~~snapshotting~~ snapshot instant. Therefore, Shuttle keeps every user request posterior to the oldest stored ~~snapshotting~~ snapshot instant. The snapshot period defines the usage of storage and computation resources. We argue that tenants can balance the costs of storage and computation resources by specifying a policy to perform the snapshot. The policy shall consider the

rate of requests, the data written per request, the expected time to detect the failure and the application capability to provide an possible degraded service during the recovery period.

Shuttle takes checkpoints automatically and according to specified policies. It records the persistent state of the application, i.e., the database values, at a certain instant. The volatile state of the application, e.g. program for instance its stack, is not stored as we consider the web servers to be state-less.

Performing snapshots in distributed databases is not trivial since snapshots have to be consistent with the user requests. We consider each user request may include multiple database operations, each of them to multiple database servers, without using transactions. Consequently, the sets of database operations of each user request cannot be aborted and do not have a global order. If Shuttle replays requests on a snapshot that contains part of the persistent state written by a request during its first execution, the replay will be inconsistent. The database must reflect the effects of a set of completed requests and not the results of partially executed requests. Therefore each snapshot shall be global request-consistent request consistent na dissertação estava global... containing either all or none of the database updates made by every request.

We define request-consistent request consistent global snapshot: a snapshot is global request-consistent request consistent if it records a state of the database which reflects the effect of a set of completed requests and not the results of any partially executed request. This concept extends the definition of derives from the notion of transaction-consistent transaction consistent global checkpoint: a checkpoint is a transaction-consistent global checkpoint if it contains all or none of the updates made by a transaction [?]. Since most of NoSQL! databases do not support transactions, we extend the concept of transaction to request-transaction request transaction. A request-transaction embraces all database operations performed due to the execution of a request. Unlike **ACID!** (**ACID!**) transactions, a request-transaction may be not possible to abort. esta nao serve para dizer que o snapshot não é apenas consistente por transacao, é consistente por request. Não há meios requests. Como não há transactions em NoSQL, a sequencia de operacoes feitas por um pedido, acaba por ser uma transacao sob o ponto de vista do snapshot

Checkpoint algorithms for distributed databases can be classified as log-oriented and dump-oriented [?]. In the dump-oriented approach, the checkpoint is referred to as the process of saving the state of all data items in the database. In the log-oriented approach, periodically a dump of the database is taken and also a marker is saved at appropriate places in the log. When a failure occurs, the latest dump is restored and the operations on the log after the dump was taken is applied to the dump until the marker is reached to restore the database to a consistent state [?]. We take the latest approach.

In addition, the snapshot mechanism shall be non-blocking: the processes shall not stop their execution while taking snapshots.

A straightforward way to take a request-consistent global snapshot is to stop processing new requests, waiting until the currently executing requests finish, then making a copy of each data item. However, this solution incurs on communication overhead to reach a globally inactive state and causes application downtime. Yet, this approach may fit applications that can be unavailable during a certain period, for instance, during a certain period of the night.

Kim and Park [?] propose an approach in which a coordinator broadcasts a checkpoint-request message to every database node. Each database node divides the transactions into two groups: before the checkpoint-request T_p and after T_f . Updates of transactions in T_p are flushed to the current database, while the ones in T_f are flushed in a checkpoint area (a temporary allocated storage area). When all transactions of T_p are done, the checkpoint area is updated with items updated by transactions in T_p but not by T_f . After, the rules of current database and checkpoint area are exchanged. The major drawback of this approach comes from updating the checkpoint area: the database is unavailable during the updating process.

Our solution leverages the existence of a single load balancer and, consequently, single proxy that adds a **SRD!** (**SRD!**) field to every request. Every **SRD!** contains a **RID!**, an unique and incremental identification of each request given by the instant when the request is retrieved. Every database operation is identified by the **RID!** of the source user request.

In order to create a snapshot, tenants define a future instant in time t when the snapshot will

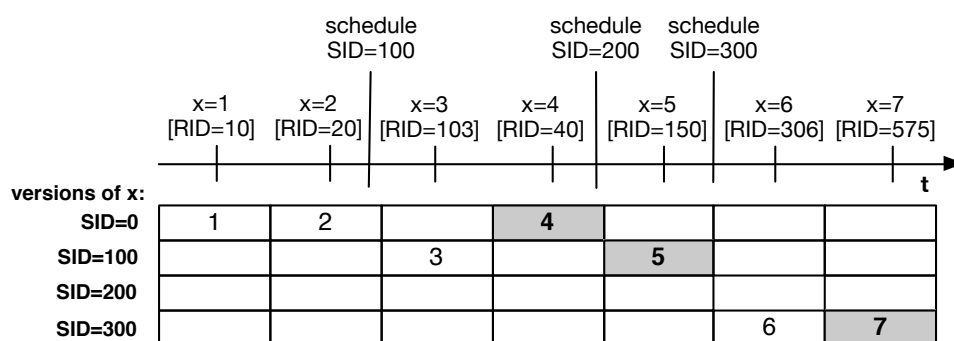


Figure 1.5: Versions stored in the database during a sequence of 7 operations and 3 snapshots. The final values of the stored versions are contained in filled squares. [The tenant schedules 3 snapshots on SID: 100, 200, 300.](#)

RID=10 (SID = 1)	RID=20 (SID=15)	Storage
Write A=10		A1=10
	Read A: A1==10	A1=10
	Write A=20	A1=10, A15=20
Read A: A1==10		A1=10, A15=20
	Read A: A15==20	A1=10, A15=20
Write A=11		A1=11, A15=20
Completed	Completed	A1=11, A15=20

Table 1.1: Concurrent requests: the snapshot instant is 15, the left request accesses data previous to the snapshot (A1) while the right accesses the latest (A15 or A1).

occur. The instant, named **SID!** (**SID!**), identifies the request-consistent global snapshot. The manager passes the **SID!** to every database proxy.

Database proxies use the **SID!** to define the version of the data item used by the operations. Operations with **RID!** (**RID!**) lower than the scheduled snapshot instant (**RID! < SID!**) access the version before the snapshot. Otherwise, the operations access the latest data item version. This mechanism splits requests to accomplish a request-consistent global snapshot, and allows tenants to schedule snapshots without application downtime. Figure 1.5 illustrates the sequence of 7 database operations on the database item *x* and 3 snapshots (excluding the base snapshot).

o schedule é para indicar que, nesses momentos, o tenant diz ao Shuttle que quer um snapshot no SID 100, 200 ou 300 In order to avoid to replay every database operation to obtain the snapshot, the snapshot mechanism shall create a database copy (dump). We avoid blocking the application to copy the versions values using a copy-on-write and incremental method: a new version is created only when the data. When a data item is written for the first time in each snapshot, Shuttle creates a new version of the data item. Since a data item may not be written in every snapshot, for instance e.g., SID = 200 in Figure 1.5, we associate a version list to every data item. A version list tracks on which snapshots the data item has been written. In conclusion, the snapshot is an incremental mechanism because it does not require to duplicate data.

A snapshot might become inconsistent. For instance, Table 1.1 represents the execution of two concurrent requests. Their normal execution is consistent. A snapshot with **SID!**=5 would contain [A1 = 11] and be a global request-consistent. However, if Shuttle loads the snapshot and replays the request 20, its first read operation reads A1 == 11 instead of A1 == 10.

This particular case happens when a request with **RID!** greater than the snapshot instant **SID!** read a version belonging to the snapshot **SID!** and, after, a request with **RID!** lower than **SID!** overwritten that version. Storing a new version and adding a flag on the version list solve solves the problem. Nevertheless, we expect this to happen only in rare occasions. o texto abaixo desapareceu porque ficou resumido

Unlike the approach proposed by Kim and Park, our approach allows to record multiple snapshots keeping various data item versions and does not require to copy the transactions.

The value of **SID!** (**SID!**) must be known by every database instance before the execution of any request with **RID! > SID!**. If the **RID!** was determined by incremental request counter, then Shuttle

would need to analyze the request rate and estimate the **SID!** value. However, the request rate can vary and the snapshot would fail. Notice that our mechanism does not require **any**-clock synchronization because the **RID!** is defined by the proxy timestamp and tenants can schedule a snapshot defining a future time instant. The period between the scheduled moment and the present must be bigger than the communication delay between the manager and the database instances. Consequently, we assume the communication between the manager and database proxies is synchronous; the messages are delivered within a fixed time.

1.5.3 Dependency Graph

An application execution can be modeled as a set of actions and a set of objects. Actions read and write **the shared set of** objects. An action A is dependent from an action B if A reads an object's version written by B .

Requests must be replayed in a consistent manner to obtain a consistent application after the replay phase. The request replay order must ensure that if the requests, application semantics and initial database values are the same, then the final database values are equal. For this propose, the dependencies between actions shall remain consistent: if during the first execution an action A becomes depends **from on** an action B by an object O , then during the replay phase A shall read the object O only after B updates the object O . Otherwise, A may read a version different than the original version.

Previous proposals, for instance in [?], leverage the request serialization provided by snapshot isolation in relational **DBMS!** to order the operations to replay. In contrast, *Undo for Operators* [?] uses the application protocol knowledge to establish the dependency between requests and order **the requests them**. However, accesses to **NoSQL!** databases are not globally serialized. Moreover, the data items accessed during the replay phase may change due to updates to the application **semantic semantics**, request modification or multi-threaded execution. At last, the application semantics is unknown in advance since we want to support any application deployed on **PaaS!**. Taking that into account, we propose a novel approach.

Shuttle tracks the dependencies between actions in a *dependency graph*. A *dependency graph* consists of nodes that represent requests and edges that establish dependencies between them (Figure 1.9). Dependencies between requests are established using the following rules: a request R_A is dependent upon request R_B if there is a data item x such that R_A reads x and R_B performs the latest update on x before the read operation by R_A . Dependencies are transitive except when requests perform blind writes, i.e., requests write items without **read-reading** them first [?]. Therefore, the dependency graph is a mixed graph, if there is a dependency between A to B , then there may be a dependency between B and A .

Previous solutions for relational databases extract the dependencies using a pre-defined, manually-created, per-transaction type template [?], or change the relational database management system code to extract read dependencies [?]. In contrast, Shuttle uses the database proxy to log the database accesses. Periodically, each database proxy traverses, in background, the *operation list* of each data item to collect the new accesses and to generate the dependencies between requests. The Shuttle manager processes the dependencies to update the dependency graph. An alternative approach is to pull the dependencies from each database node only before the recovery process and generate the dependency graph when needed.

The above method may lead to *false positives*, i.e., to flag dependencies that do not exist. For instance, a request may read a data item but not use it to compute the written value, so there is no real dependency. Although tracking variables used by each request during its execution might solve this particular case [?], it would require modifying the code interpreter (e.g., Zend Engine for PHP), which would constrain Shuttle to a set of specific languages. As our approach uses the dependencies to group the requests that can be executed concurrently, false dependencies imply a performance penalty but do not cause data loss or inconsistent state. On selective replay mode, the dependency graph is used to determine the tainted requests and the request that need to be replayed. Again, false dependencies only harm the performance.

When tenants use the dependency graph to determine the set of malicious requests, $A_{malicious}$, they shall take into account that false dependencies may lead to consider legitimate operations as malicious and, consequently, cause data loss.

Complex queries on a relational database may lead to *false negatives*, i.e. a dependency exists but

is not detected. For instance when a read operation would have been executed on a deleted data item if this data item had not been deleted before the request execution [?]. Therefore, legitimate transactions may have different output even when they were not affected by malicious execution during their original execution. Since user mistakes are often ~~deletes~~ delete operations due to wrong query arguments, this is a relevant issue.

In contrast with SQL queries that access the data items that match a query, the **CRUD! (CRUD!)** interface of most key-value stores specifies, in a deterministic and apriori manner, the data item that will be accessed. Shuttle logs every access, even when the data items do not exist, keeping the *operation list* of the deleted data items to track further operations.

Shuttle can not replay requests synchronously, i.e., waiting for the response to the previous request before sending the next. To replay the requests synchronously would not have only performance degradation but also lock the replay phase because requests, which have been originally executed in concurrently during the normal phase, may be depend ~~from-on~~ each other. Therefore, Shuttle replays requests asynchronously and, hence, concurrently. Two requests are executed concurrently if they are dependent from each other. For instance, Figure 1.6a represents the first execution of two requests that increment the variable *A*. The *Request1* depends ~~from-on~~ Request2 and vice versa.

Yet, re-execution of concurrent requests is not deterministic. User requests are processed concurrently using multi-threaded servers and the system messages, including database requests, do not have a delivering order. Therefore, the execution order of two concurrent requests is unknown. ~~Our~~ To deal with this issue, our novel approach uses the *operation list* to turn the re-execution of concurrent requests deterministic. An *operation list* is a sorted list that records the operations to a data item. During the replay phase, the operations to a data item must follow the order established by its operation list. For instance, in Figure 1.6a, the operation list of the data item *A* is: [*Req1* : *Get*, *Req1* : *Put*, *Req2* : *Get*, *Req2* : *Put*, *Req1* : *Get*, *Req1* : *Put*, *Req2* : *Get*, *Req2* : *Put*]. *Req. 1* and *req. 2* are replayed concurrently but the result is consistent because the order is established by the operation list.

During the recovery period, intrusions are removed and the application code is updated. This may cause requests to access different data items than in the first execution. Requests may not access the same sequence of data items or read/write the same content. If an operation contained in the operation list is not performed, the following operations to the data item are blocked and the request fails. To address this problem, at the end of each request execution, the *database client interceptor* fetches the list of data items accessed by the request on its first execution and compares them against the ones accessed during the replay process. The database client library invokes the *database proxy* with the ~~keys~~ data items that have not been accessed to unlock the operations of the remaining requests.

For instance in Figure 1.6b, the *Request1* has a different replay execution performing $B = B \times 5$ instead of incrementing *A*. The second operation of *req. 2* is delayed until the end of the *req. 1* because it succeed the second operation of *req. 1* in the operation list. After the execution of *req. 1*, the database client interceptor unlocks the second operation of *req. 2*.

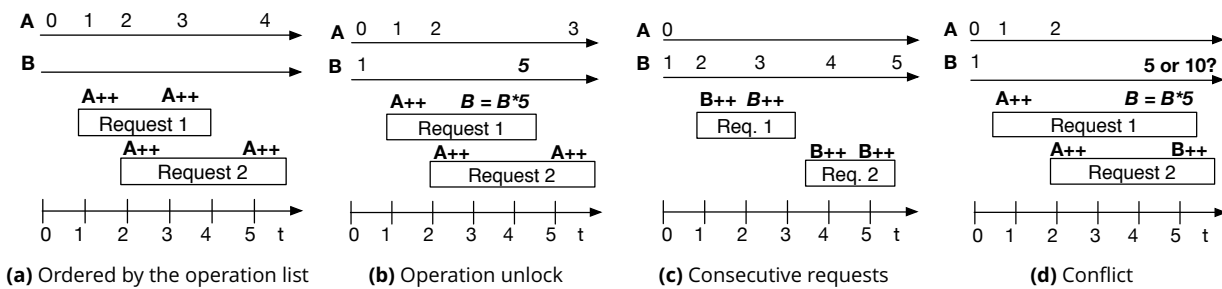


Figure 1.6: Replay two requests with different re-execution

During replay there may be non-deterministic situations, whenever an access is not contained in the operation list. The most complex scenario during the replay is when two requests, originally executed in parallel, access different data items comparing with their first execution, establishing a

new dependency. The result is unpredictable. Consider the possible re-execution in the Figure 1.6d where the req. 2 increments B simultaneously with req. 1 performing $B = B \times 5$. The result is unpredictable because the req. 1 may write before or after req. 2. Since both requests did not access the data item B during their first execution, the operation list does not establish an access order. Therefore, the req. 1 and req. 2 may execute on an arbitrary order. The order of these requests is as deterministic as if during the first execution: the operation of req. 1 can execute before, between or after req. 2.

A naive solution would be to detect the new dependency during the replay process, stop the process and start a new replay process in a new snapshot, including the new dependency.

Brown *et al.* [?] ~~proposes~~ propose the concept of *verb* (Section ??). A verb object encapsulates a single interaction (request/response) of the user and exposes an interface to establish the order between requests and their dependencies. However, tenants shall know the applications' operations to create the verbs defining: a commutativity test, an independence test, a preferred-ordering-test and an application-defined action to handle an inconsistency if the operation fails. Since Shuttle shall support any **PaaS!** application, the applications' semantics are unknown in advance. Therefore, this solution is not adequate.

A sorted-log, in which the accesses are sorted, for instance by **RID!** (**RID!**), would establish that operations must have a strict incremental order. However, operations with smaller **RID!** than the previous are aborted. For instance, the second operation of req. 2 in Figure 1.6a would be aborted.

An alternative solution consists on sorting the requests per *start-end order*, instead of using the dependency graph. A request starts only after all requests with end lower than it ends. Dependencies between requests remain correct, since they are constrained by the *operation list*. Thus the parallel requests are ordered and replayed in a similar manner to the first execution. Two serial requests can have distinct re-executions: if a request starts after the end of the previous. For instance in Figure 1.6c, req. 1 and req. 2 can have distinct re-executions.

If two ~~new~~ operations are re-executed concurrently, then their order is as deterministic as if they happen during the first execution. For instance in Figure 1.6d, the operation of req. 1 $B = B \times 5$ is executed in parallel with the operations $B++$ of req. 2. The order of these requests is as deterministic as if during the first execution: the operation of req. 1 can execute before, between or after req. 2.

In order to turn more operations of the replay process ~~more~~-consistent with the first execution **exato, cada operacao ou é consistente ou não é. A consistencia do processo poderia ser vista como a média da consistencia ou não de todos - uma binomial - mas fica ambigo. Assim fica explicito**, we can leverage ~~the~~-semantic reconciliation, as in Dynamo [?]. The case represented in Figure 1.6d is equivalent to a concurrent update where two parallel writes are performed on distinct database instances. Each request writes a distinct version resulting in conflicting versions of an item. Developers use the application-assisted conflict resolution interface to merge the versions (reconciliation) [?]. In this case, the following read operation would access the values written by the latest operation. For instance, if the latest is req. 1, then it choose between 1 and 2. If the latest is req. 2, then it choose between 1 and 5. This solution can produce a consistent output.

In summary, unlike previous solutions, Shuttle orders the requests using their start and end instants and constraining the operations to the order established on the *operation lists*. This approach allows requests to access new data items during the recovery process and to replay concurrent requests.

1.5.4 Clustering

Despite Shuttle's capability to replay concurrent requests, one of the main challenges of Shuttle is to reduce the recovery period. We assume that critical software flaws and intrusions can be detected in a short period of time, from seconds to one week. If a fault exists during a longer period, then the application may tolerate a longer recovery phase because the recovery process used by Shuttle does not require application downtime (Section 1.5.6). Still, we want recovery to take a fraction of the time elapsed since the snapshot from which recovery starts (e.g., if the snapshot was taken a week before, we want recovery to take much less than that period).

We address this problem grouping the requests into *clusters*. A cluster is a set of requests that have dependencies between them but not from/to requests in other clusters. Clusters are created when the recovery is about to start by inspecting the dependency graph. Since clusters are independent, they are executed concurrently by different *replay instance* without synchronization. Requests within

the same cluster, are performed in start-end order (Section 1.5.3). Given that more requests are executed concurrently, Shuttle launches more application servers and database instances to process the replayed requests. Therefore, the replay phase throughput is bigger than the during first execution and the recovery time is minimized. This mechanism is applicable if the graph dependencies remain unchanged during the recovery phase, i.e., every replayed operation is contained in the operation list but not all operations in the list must be replayed.

Taking the above in account, we define two replay approaches: *serial replay* and *parallel replay*. The first considers every request in the same cluster. The later uses the dependency graph to group the requests in independent clusters. Both approaches replay the requests in a start-end order, supporting concurrent requests 1.5.6. In contrast to *serial replay*, *parallel replay* allows to perform more requests in parallel but it does not support new dependencies during the replay phase. Therefore, *parallel replay* requires that tenants ensure that the dependencies between requests do not change during the replay process. Since the dependencies between requests often remain constant and novel dependencies are easily detected, we consider *parallel replay* represents a significant advantage for **CSP!** (**CSP!**). These approaches are compared in Chapter 3.

1.5.5 Instance Rejuvenation

PaaS! systems launch instances/containers and deploy applications or databases on them. Attackers may exploit vulnerabilities in the instances configuration to affect the service integrity, confidentiality or availability. For instance, an attacker may explore the shellshock vulnerability in the GNU's bash shell of an out of date instances.

An effective technique to remove intrusion effects and restore the application availability is to terminate compromised containers and launch new containers. We name this approach as *instance rejuvenation*. A similar approach is used in proactive recovery systems for ~~byzantine~~ Byzantine fault tolerance. Castro *et al.* [?] proposes propose a mechanism that recovers the replicas of a system periodically even if there is no reason to suspect that they are faulty. This mechanism aims to prevent an attacker from compromising the service by corrupting a quorum of the replicas without being detected. We extend this approach to **PaaS!** to remove possible intrusion effects in containers, even if there is no reason to suspect that they are affected by the intrusion.

Shuttle interacts with the **PaaS!** controller rejuvenate instances when they are compromised and a new recovery process begins. This process launches new instances. The **PaaS!** controller initializes the new instances with updated container images and deploys an updated version of the application code or database, which may include updates to fix discovered flaws or prevent future intrusions. Shuttle copies the snapshot selected by the tenant to new database instances. Requests are replayed on the novel instances while the incoming requests are processed by the old instances, perhaps with a degraded integrity constrains. This mechanism keeps the application available during the recovery process. After the recovery process, the old instances are terminated.

We assume new instances to be intrusion-free since tenants or **CSPs** can update the image and the image is installed on an empty persistent-state. This approach fits the concept of automatic deployment applications in **PaaS!**. Applications for **PaaS!** platforms are designed to scale horizontally so the number of application instances can be dynamic.

In addition, the instances can be instantiated in a remote site to recover from catastrophic disasters [?]. Snapshots, database operation lists, application code and requests can be replicated to a remote site. If they are available, then Shuttle can launch new instances on a remote datacenter, deploy the application code, load the snapshot and operation lists in the database instances and replay the requests. This is a log-based recovery process [?] that allows to recover the application integrity and availability.

This process can also be used in a proactive manner to renew instances to remove unknown intrusions [?] or to test new application versions with user requests to compare its results against the previous version, using the branching mechanism (Section 1.5.6).

Tenants are responsible for ensuring that request dependencies are correct and the API of the updated code version is compatible, or for providing a script to update each request to the new **API!**. Moreover, the selected snapshot must be consistent according to the specification of the updated version or every request executed since the application begin shall be replayed.

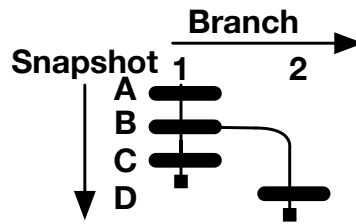


Figure 1.7: Tree model: 3-2 branches and 5-4 snapshots: branch 1 contains the snapshots *A, B, C*; branch 2 contains the snapshot *D*; branch 3 contains the snapshot *E*.

1.5.6 Runtime Recovery

Applications shall remain available during the recovery process, perhaps with a degraded behavior, without exposing downtime to users. To do so, Shuttle considers each recovery process defines a new branch, a model inspired in versioning systems such as git [?].

A *branch* is a sequence of snapshots. Snapshots are analogous to *commits* in git. Each snapshot represents a set of versions of every data item in the database at a certain instant. Each recovery process creates a new branch forking a previous branch on a snapshot chosen by the tenant é fork da branch mas pode ser num snapshot que não o seu ultimo snapshot, bem visto, either explicitly or implicitly (by indicating the initial intrusion instant, selecting implicitly the preceding snapshot). When a new branch is created, a new snapshot is also created on the new branch sim, para fazer de snapshot base, eu tinha um mecanismo mais complicado em que não era preciso criar esse snapshot mas assim: cada snapshot existe numa e só uma branch. Incoming user requests access only the data of the previous branch keeping the application available, while replayed requests access the created branch without compromising the availability of the application. While Aire [?] used a branching mechanism to perform a recovery process in various systems simultaneously, we propose the mechanism to isolate user accesses from the recovery process. This model allows the tenants to select any snapshot as base to a new recovery process and create snapshots in different branches. Applications can contain multiple branches simultaneously. Figure 1.8 represents an application with 3 branches: the branch 1 is the initial application branch where the tenant made three snapshots (*A, B* and *C*). After detecting an intrusion, the tenant considered that snapshot *C* is non-tampered and initiated a recovery process based on it creating the branch 2. Afterwards, the tenant made one snapshot (*D*) on branch 2. However, the snapshot *C* is tampered. So the tenant initialized a novel recovery process based on snapshot *B* creating the If Shuttle launches new database instances, then the new branch 3. In this scenario, the tenant would be unable to recover its application without the branching model. is created in the new instances and write operations occur in the new instances. Read operations occur in the previous instances until the first write operation of the accessed data item in the new instances.

A branch contains sequence of multiple a sequence of snapshots but a snapshot can only belong to a single branch (Figure 1.7). Each snapshot represents a possible version of the data item. A novel data item version is created only when the data item is written for the first time during each snapshot. Consequently, the data item may not have a version for each snapshot. Shuttle keeps a list of the versions in which each database item has been written (Section 1.5.2). In addition, tenants may fork a new branch not only from the most recent snapshot. Therefore, the latest non-tampered version of a data item may not be its latest version.

For instance, in Figure 1.8, a data item *x* may have the following sequence of versions on its version list: [*A, B, C, D, E*]. If *x* has not been written in snapshot *E*, the version *E* would not exist and the latest version of *x* would be *D*. However, the snapshot *D* has been compromised so it is not part of the current branch (branch 3). The latest non-tampered version of *x* is the version *B*. If version *B* does not exist, then the latest version is *A*.

We define the concept of BranchPath: a BranchPath Branch Bath bem visto, foi mesmo copy-past do nome da variavel :P: a Branch Path of a certain branch is the sequence of non-tampered snapshots between the current snapshot and the root snapshot. A BranchPath bem visto, o snapshot pode ser tampered. A Branch Path of a certain branch defines the versions available to operations that belong to that branch. The BranchPath-branch path of branch 3 in Figure 1.8 is {*E, B, A*}. The one of branch 2 is: {*D, C, B, A*}. When a branch is created, its BranchPath-branch path contains the its initial

snapshot and the sub-sequence of snapshots in the ~~BranchPath~~[branch path](#) of the branch of the forked snapshot that are equal or previous to the forked snapshot.

The version accessed by an operation is defined using the ~~BranchPath~~[branch path](#) of the operation's branch and the version list of the accessed data item: operations read the latest version present in the *version list* and in the *branch path* and write the latest version in the branch path. Therefore, a new version, referring the initial snapshot of the new branch, is added to the version list on the first write operation to each data item during the replay.

This mechanism maps the operations to the correct versions and isolates the multiple, perhaps simultaneous, attempts to recovery the application without compromising the exposed application behavior. During the recovery process, users access the, perhaps corrupted, old branch loaded in the current computation and database instances. Therefore, the application remains online, perhaps with a degraded behavior, without exposing downtime to users.

At recovery time, the manager sends the new *branch path* to every database instance. The new incoming users access the, perhaps corrupted, old branch while the requests being replace access the new branch. Therefore, the application remains online, perhaps with a degraded behavior, without exposing downtime to users.

At some point, when the recovery is finishing, the user requests have to start being issued to the new branch. To do so, after replaying the requests, the proxy flag *restraining* is set and every new request is marked with the *restrain* flag. Database accesses marked with *restrain* are delayed. After replaying the requests retrieved during the recovery process, the proxy sets the new branch in the subfield *branch* of **SRD!** of the new requests, the *restrain* flag is disabled and the database nodes are notified to proceed the accesses. This mechanism delays the processing of some requests, but this has typically a duration of seconds, compared with a recovery process that may take many minutes or even hours.

This mechanism ensures the recovery phase to be finite. However, if the rate of requests being replayed is much higher than the rate of new requests, then the restrain phase can be shorter. The requests retrieved during the replay phase are also replayed without restraining new requests. Consequently, the restrain phase is required only to change branch.

podemos fazer replay dos pedidos recebidos durante o recovery process sem o restrain. O restrain apenas tem de acontecer apenas no final ao mudar de branch. É um erro. Nada impede que esses pedidos executem mas simplifica porque senao o processo poderia nunca acabar... - é verdade, mas diversos pontos: 1- o restrain não aparece só aqui, ms logo quando defines o SRD 2- tens sempre uma espécie de restrain mas pode ser muito mais curto do que é agora 3- se retiras o restrain o que fazes com os experimentos que o medem? acho que podes deixar fora

While Aire [?] used a branching mechanism to perform a recovery process in various systems simultaneously, we propose the mechanism to isolate user accesses from the recovery process. Our model allows the tenants to select any snapshot as base to a new recovery process and create snapshots in different branches. Applications can contain multiple branches simultaneously. Figure 1.8 represents an application with 3 branches: the branch 1 is the initial application branch where the tenant made three snapshots (*A*, *B* and *C*). After detecting an intrusion, the tenant considered that snapshot *C* is non-tampered and initiated a recovery process based on it creating the branch 2. Afterwards, the tenant made one snapshot (*D*) on branch 2. However, the snapshot *C* is tampered. So the tenant initialized a novel recovery process based on snapshot *B* creating the branch 3. In this scenario, the tenant would be unable to recover its application without the branching model. Since tenants may fork a new branch not only from the most recent snapshot. Therefore, the latest non-tampered version of a data item may not be its latest version.

Inactive snapshots and branches, for instance snapshot *D* and branch 2 in Figure 1.8, can be deleted to reduce the used storage resources. In addition, tenants can use the branching mechanism to test their intrusion recovery procedures in background, i.e, without exposing users to test issues.

If an intrusion happens during the replay phase, then its effects are stored in the branch of incoming requests. If the intrusion is detected before the restraining flag is set, then the malicious requests are not replayed in the new branch. Otherwise, tenants shall start another recovery process.

~~If Shuttle launches new database instances, then the new branch is created in the new instances and write operations occur in the new instances. Read operations occur in the previous instances~~

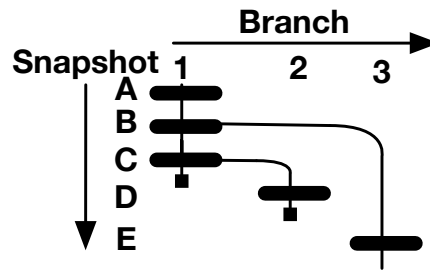


Figure 1.8: Tree model: 3 branches and 5 snapshots: branch 1 contains the snapshots A, B, C; branch 2 contains the snapshot D; branch 3 contains the snapshot E.

until the first write operation of the accessed data item in the new instances.

1.5.7 Non-determinism and consistency

Shuttle provides an **API!** to handle the nondeterminism and inconsistency cases.

An application is called nondeterministic if two subsequent executions with the same user input cannot be guaranteed to have the same final state and outputs. Five of the main sources of non-determinism in **PaaS!** applications are: shared memory, thread concurrency, random number generation, timestamps and message exchanging.

We assume requests to be independent thus they do not share memory and concurrent threads are independent. The **API!** provided by Shuttle provides a deterministic random number generation and timestamps using a timestamp. It uses the **RID!** (**RID!**), which is a timestamp set by the proxy, as timestamp and pseudo-random number seed, so the replay of a request will use the same random numbers and timestamp. We consider a single timestamp per request to be enough for most applications. This mechanism is language independent. User requests and database accesses are ordered in a deterministic way using the operation list. **devo colocar aqui a assinatura da API? como era só getTimeStamp() return long; e getRandom() return randomNumberGenerator(); não coloquei...**

An important aspect of a recovery system like Shuttle is the application consistency seen by users. For instance, if an user does an action based on data written by a malicious action, which result of the user action replay is consistent. Since users have a non-deterministic behavior, they may have to be notified if a recovery took place and their data was modified.

Since Shuttle **unknowns-is not tied to** the application semantics, the actions to compensate the recovery process changes are unknown **before the application is created**. In addition, the application may contain client-side code, e.g., Javascript, that processes the application **responses**. For instance, a recover process reorders a list of items. The client-side code may sort the items so the list is seen ordered by users. A replay process taking into account the client-side consistency is proposed in [?].

Shuttle does not execute requests that returned an error in the first execution. We assume that requests are synchronous so users are immediately notified of the error and do not expect that the request will succeed in future. Similarly to other works in the area [?], we assume that these cases are compensated by the user when they happen. As only requests that did not return an error are replayed, Shuttle considers an inconsistency when a request returns an error or a response is different during replay. Shuttle provides the following **API!** for the application programmer to define how inconsistencies are dealt with (Shuttle calls these functions in case they are launched by the tenant):

1. *preRecover()*: invoked before the beginning of the recovery process.
2. *handleInconstancy(request, previous response, new response, previous **keys**data items, new **keys**data items, action)*: invoked when there is an inconsistency.
3. *postRecover(statistics, old version, new version)*: invoked after the end of the recovery process.

The first function allows tenants to perform a set of actions before the beginning of the recovery process, such as notifying the operations team or taking a new snapshot. The second function takes

as input the operation that caused the inconsistency as well as the response and keys accessed during the normal execution and during the recovery process. It also takes as argument the action to take. Currently we consider three possible actions: 1) ignore the inconsistency; 2) notify the user of the inconsistency; 3) execute another request. This function is invoked, for instance, if a response during the replay is different than the response on the first request execution. Using the *postRecover* function, the tenant has access not only to the statistics of the recovery process but also to an interface to compare the database values before and after the recovery process and the application responses, before exposing the data to the users. Tenants can use this interface to notify their customer to verify their data.

Besides its users, an application may also interact with external services. We simplify the problem by considering that applications only obtain inputs from external services, disregarding the issue of outputs. The problem is treated in [? ?]. Brown *et al.* [?] models each external service as a recoverable application. During the recovery phase, an external service can also be recovered if its input is distinct. Aire [?] proposes to initiate a recovery process in the external service and handles the inconsistencies of this process.

1.5.8 Full and Selective Replay

We propose two approaches for intrusion recovery: full replay and selective replay. Full replay consists in replaying every request done after the **selected** snapshot. Executing many requests **can take takes** considerable time, so this approach is adequate for **issues intrusions** detected reasonably fast after they happen, e.g., a few days **or weeks**.

Selective replay (Section ??) re-executes only part of the requests so it is faster **but requires that tenants or intrusion detection mechanisms than full replay. However, it requires tenants to** provide a set of malicious **requests, actions (i.e., requests)** $A_{intrusion}$. This set is used to deduce the set of tainted requests $A_{tainted}$. A request is said to be tainted if it is one of the attackers requests or if it reads objects written by tainted request [? ? ?].

Tainted requests can also be determined by Shuttle considering the tampered data items and an estimated intrusion moment. Selective replay approach loads only the previous versions of the tainted objects, $O_{tainted}$, and replays only the legitimate operations, which were tainted, $A_{tainted} \notin A_{intrusion}$, to update the application persistent state. Selective replay, as compensating actions, does not remove the effects of unlogged actions because their dependencies are unknown.

In [? ?], the set of tainted operations, $A_{tainted}$, is determined using *taint propagation via replay*. To do so, they load a previous version, from a snapshot, of the objects in $O_{intrusion}$. Then, the actions, which are dependent from the restored objects, are replayed and their output objects are updated. The forward actions, which depend **from-on** the updated objects, are also replayed while their inputs are different from the first execution. The propagation is done thought the output of actions with different execution. Unlike these approaches, Shuttle does not store the input and output of every action, i.e., database operation. Shuttle proposes an approach in which the requests are replayed, at least, until the first snapshot after the selected snapshot. Consequently, the application semantic must remain unchanged, i.e., the same request and same input must perform the same write operations. Otherwise, the dependencies between requests are unpredictable and the tainted requests can not be determined. An approach that allows to update the application semantics is proposed in [?]. We consider storing all versions of a data item has prohibitive storage costs for enterprise applications.

For instance, consider the dependency graph of **Fig.-Figure** 1.9, in which every request reads a data item and writes a new value on the same data item. The request 4 was identified as a malicious request. Therefore, requests 5, 6, 7, 8 are tainted. Since Shuttle does not keep every version of the entries, the value read by request 4 is unknown. In order to get this value, Shuttle must replay the req. 2, which wrote the value read by req. 4. The value read by request 2 is known because Shuttle performed the checkpoint A. Since the application semantics remains the same and its input is known, req. 3 does not need to be replayed. Requests 5, 6, 7 are replayed since they depend **from-on** the malicious req. 4. Values read by request 8 are known due to checkpoint B. Therefore, req. 8 may not be executed if the value of the data items remains the same. Shuttle performs *taint via-replay*: if a request writes in a data item which were not written previously, then the requests which read or write that data item, are also replayed. For instance, the req. 9 may read a data item written by the req. 4 during the replay but not during its first execution.

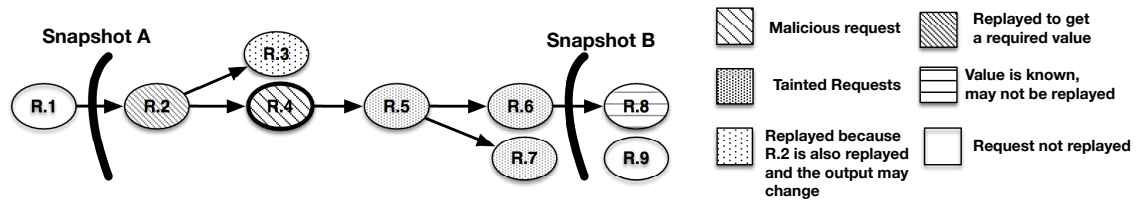


Figure 1.9: Dependency graph: $R.1$ is previous to a snapshot A ; $R.3$ is dependent on $R.2$, which is replayed to get the read values; $R.4$ is a malicious request; $R.5, R.6, R.7$ are tainted; $R.8$ may not be replayed; $R.9$ is independent of the rest

The selective replay process is as follows [\(full replay is simpler so we skip it\)](#):

1. *Determine the malicious requests $A_{intrusion}$.* Based on initial data such as user session compromised or data items accessed or other criteria, the tenant determines the requests $A_{intrusion}$ used by the attacker to compromise the application. For instance, $A_{intrusion} = \{R.4\}$ in Figure 1.9.
2. *Use $A_{intrusion}$ to determine the set of tainted requests $A_{tainted}$.* For each request in $A_{intrusion}$, traverse the dependency graph in causality order and add these nodes to $A_{tainted}$ (in the figure $\div A_{tainted} = \{R.5, R.6, R.7, R.8\}$).
3. *Get the requests needed to obtain the values read by $A_{tainted}$ and their effects.* Instead of storing the input and output of every action or versions of every data item, we propose to replay the actions which $A_{tainted}$ depends on. The data item value is known at the snapshot instant so the algorithm transverses the graph in inverse causality order from each request in $A_{tainted}$ and stores the requests in A_{replay} ($A_{replay} = \{R.2\} \cup A_{tainted}$). A_{replay} is expanded by traversing the graph from each of its elements on causality order to determine the requests which can be affected by the re-execution of A_{replay} ($A_{replay} = A_{replay} \cup R.3$). Requests subsequent to the first snapshot after the latest malicious request may not be repeated as the data item version is known (version read by $R.8$ is stored in snapshot B).
4. *Determine the replay order.* The set A_{replay} is sorted on non-clustered *start-end order*.
5. *Load the previous data item versions* Shuttle loads the version in the selected snapshot of the data items read by the requests in A_{replay} and written by $A_{malicious}$.
6. *Replay the requests* Requests in A_{replay} are replayed. If an access is not contained in operation list, then a new dependency is established and the requests that accessed the data item during the first execution are also replayed as in *taint propagation via replay* [?]. For instance, $R.9$ is replayed if it reads an item written during recovery process but not during normal execution.

Shuttle does not require generating a dependency graph in non-clustered full replay mode. The dependency graph is required in the case of clustered full replay to identify the independent clusters of requests and on selective replay to determine the tainted requests. We use the database operation lists to create the dependency graph and to order the execution of parallel requests without knowledge of the application protocol.

In summary, the selective replay approach reduces the number of requests to be replayed during the recovery process but implies that the application remains unchanged and does not revert the actions performed by unlogged requests.

1.6 Chapter Summary

Shuttle recovers from security intrusions loading a previous snapshot and replaying the legitimate requests. It uses database snapshot and clustering to reduce the recovery time. Shuttle leverages the pay-per-usage model of **PaaS!** to provide a cost-efficient and fast recovery service instantiating the replay instances and more application containers on demand during the recovery process.

	Clustering	Non-Clustering
Selective	✗	✓
Full	✓	✓

Table 1.2: Shuttle Replay modes

Shuttle proposes two approaches to perform replay: selective replay and full replay.

The full replay approach supports parallel re-execution of requests that belong to independent clusters ~~-(clustering)~~. Clustering is not supported on selective replay because the *taint propagation via replay* defines the set of requests to replay at running time. não sei como explicar... o selective replay vai descobrindo quais são os pedidos a repetir enquanto executa. Se os pedidos já tiverem sido executados, em paralelo, teriam de ser undo, etc. não é impossível mas é mais complexo.

The decentralized applications are more vulnerable to failures because of the single proxy architecture. However, we argue that future architectures can consider replication of the proxy, load balancer, Shuttle Storage and database.

2

Implementation

This chapter addresses the main decisions adopted regarding the implementation of the intrusion recovery system Shuttle. Shuttle is supposed to be implemented by the **PaaS!** (**PaaS!**) providers since it requires modifications to the **PaaS!** system and database service.

2.1 Adopted Technologies

In order to implement the proposed work, we evaluated several technologies taking in account the requirements of the problem and the performance, scalability and easiness of implementation in a production mode **PaaS!** (**PaaS!**) provider service. The first Shuttle prototype was implemented in Python. The prototype of a simple application and the proxy allowed to analyze the requirements of Shuttle. The compact code of Python was great for code readability and rapid prototyping. However, despite the stackless variants of Python, the global interpreter lock limits Python to a single active thread per process. Since all modules of Shuttle are multi-threaded and their performance is critical to support medium and large-scale systems, Python was not appropriated. The selected language needs to be likely to run faster than Python and to have better concurrency and network libraries.

C or C++ languages are likely to have a better performance than Python because they are compiled. However, the development time can be considerably larger due to its low level. Java has ~~an~~^a higher abstraction level than C but lower than Python. To argue about the performance differences between Java and Python, we would need to develop a version of Shuttle in Python and another version in Java. However, the proxy prototype using Java has a bigger throughput and lower latency than the one implemented using Python.

Shuttle has a distributed architecture and each module shall process various requests concurrently. Therefore, we also considered languages, such as *Erlang*, *Go Lang* and *Scala*. They facilitate the writing of concurrent programs that share the state by communicating. Moreover, the actors model [?], native in Erlang and available through frameworks as Akka [?] to Scala, could simplify the implementation and improve the performance.

Java is still the most used language in enterprise environments and it is compatible with most of **PaaS!**. The selected database is implemented in Java (Chapter 2.1.2). In addition, Java has ~~a~~^{the} **NIO!** (**NIO!**) library that provides event-driven and asynchronous communication via network. We chose to implement the Shuttle prototype in Java. However, the large number of concurrent tasks and asynchronous message passing communications turned the development of the prototype using Java extremely complex. Moreover, ~~it's~~^{its} concurrency model and network libraries have an abstraction level lower than desired to archive a fast development and good performance.

Therefore, during the development of the prototype, we concluded that *Scala* with *Akka* could be a better option than Java. It would be likely to simplify the development of the distributed system and improve the performance. However, rebuilding the prototype in a different language would also take too long. Therefore, we kept Java. In addition, the proxy performance could be improved using a C or C++ implementation, such as HAProxy [?].

~~Despite the Java, we choose to use a messaging protocol because~~ Shuttle is built of several modules that communicate and coordinate their actions ~~by passing messages.~~ We choose to use a messaging protocol instead of RPC! (RPC!) (such as Java RMI! (RMI!)). The message passing protocol should emphasized simplicity, performance and low overhead. It should also define data structures and service interface easily.

Plain text protocols ~~is a~~ are communication protocol whose content representation is intended to be read by humans. For instance, the SOAP! (SOAP!) [?] specifies rules for using XML to package messages. Despite the heterogeneity and easiness of debug of plain text protocols, the text encoding and message structure has a significant overhead. As opposed, binary protocols are intended to be read by machines. The advantage of ~~terseness~~ compactness translates into smaller messages and faster transmission and interpretation than plain text protocols. The capability of describe data structures using ~~a an~~ IDL! (IDL!) and generate the source code in various programming languages to generate and parse the stream of bytes is also important to reduce the period of implementation and support transparent interaction between multiple programming languages. To cope with these requirements, we selected a binary protocol with **IDL!**.

Apache Thrift [?] and Google's Protocol Buffer [?] are two of the current binary communication protocol that fit these requirements. These protocols are similar, the main difference is the first offers a stack implementation for **RPC! (RPC!)** calls. The exact performance differences between them can only be measured benchmarking the two possible implementations. We choose Protocol Buffers to implement our prototype.

2.1.1 Platform as a Service

Shuttle is implemented as a service in **PaaS!** framework. Since we can not modify the implementation of **CSP!** solutions, such as Google App Engine [?] and Amazon Web Services [?], we chose an open-source **PaaS!** framework. The framework ~~must~~ has to meet the following requirements:

1. *Open-source*: to support updates and modifications
2. *Support to add new containers* ~~(auto-scaling is not mandatory)~~: to add new compute, database and replay instances.
3. *Contain a load-balancer*: to distribute the requests after the proxy
4. Extensible to support a new database management system
5. Deployable on OpenStack and **AWS! (AWS!)**
6. Support for Java Applications

We consider the auto-scaling capability as optional since Shuttle may invoke the **PaaS!** manager to create new ~~web-servers~~ application instances, database instances and replay instances. However, the capability to monitoring the containers usage during the recovery period turns the replay process faster and cost-efficient by adjusting dynamically the required resources (Chapter 3.4).

We require the **PaaS!** system to be deployable on OpenStack [?] and **AWS! (AWS!)**. Due to the costs of a public cloud service, the prototype is tested in a local cloud supported by OpenStack and later on a public cloud **AWS!**.

~~The number of systems that meet these requirements are rising and being actively developed. The main open-source systems are OpenShift by RedHat [?], AppScale [?], Apache Stratos [?], Cloud Foundry [?], Cloudify [?] and Solum project of OpenStack [?]. Solum matches the requirements but it still being actively developed.~~ We chose AppScale [?] to implement our prototype. AppScale meets the requirements except it does not support OpenStack nether supports the required database (Chapter 2.1.2). Therefore, we contributed for AppScale open-source project adding the support for both. AppScale supports auto-scaling monitoring the proxy and nodes: if one of the servers queue

contains more than 5 requests or if the node usage exceeds the 90%, then it creates a new container and deploys the application. [There are many alternatives being actively developed. The main open-source systems are Openshift by RedHat \[? \], AppScale \[? \], Apache Stratos \[? \], Cloud Foundry \[? \], Cloudify \[? \] and Solum project of OpenStack \[? \].](#)

2.1.2 Database

Shuttle could be implemented in **PaaS!** [supported by](#) most of known databases, including relational (**SQL!**) databases ([Database instances in Figure 1.2](#)). However, previous projects encompass relational databases [? ?] and **PaaS!** applications are often supported by **NoSQL!** databases. Shuttle is the first intrusion recovery system using replay considering **NoSQL!** databases.

NoSQL! databases are designed for extremely large data sets, hundreds or thousands of million entries. Most of the architectures of **NoSQL!** databases claim to scale horizontally near linearly, i.e., adding twice the data means to add twice the nodes. To do so, the data of **NoSQL!** storages is partitioned across multiple servers, for instance, by a range of keys or using consistent hashing [?]. Therefore, **NoSQL!** stores can compensate the overhead imposed by Shuttle distributing the data across the nodes, scaling horizontally instead of vertically.

Unlike relational databases, **NoSQL!** databases do not guarantee **ACID!** (**ACID!**) properties. One of the main differences between the **NoSQL!** storages is their approach to preserve consistency or availability during the network partitions. The **CAP!** theorem states any networked shared-data system can have at most two of the three desirable properties: consistency, availability and tolerance to network partitions [?].

NoSQL! databases support new data structures, which are more flexible than the schema of relational databases:

1. *Column-oriented*: The data is organized in a multidimensional persistent sorted map with a large number of key-value pairs within rows.
2. *Document-oriented*: The format of the values is a JSON or JSON like document. Documents are organized in collections. The document attributes can be included in the query.
3. *Key-value*: Similar to a map where the values are opaque and accessed by an unique key.
4. *Graph*: Heavily linked data with multiple relations.

The evaluated alternatives are summarized in Table 2.1.

Type	Name	Language	API	Replication	Consistency
Column	HBase	Java	Avro, REST, Thrift	Master-Slave	Strongly consistent for a single row within a datacenter. Across rows is eventual consistent
	Cassandra	Java	Cassandra Query Language	Peer-to-peer	Depends from on the selected number of read and writes
Document Oriented	MongoDB	C++	CRUD!	Master-slave	Strong Consistency for a single row (default)
	CouchDB	Erlang	CRUD!	Master-slave	Multi-Version Concurrency Control
Graph	Neo4J	Java	Cypher Query Language	Master-Slave	ACID using master. Updates to slaves are eventual consistent by default
Key value	Memcached	C	CRUD!	None	The client selects the correct shard
	Voldemort	Java	CRUD!	Peer-to-peer	Many-writer eventually consistent system: vector clocks and versioning

Table 2.1: Available No **SQL!** databases

HBase and Cassandra follow the Google Big Table column-oriented database [?]. HBase has master-slave architecture and provides get, put, delete operations but also scan, server-side atomic operations and atomicity on row-level writes. Cassandra has a peer-to-peer architecture based on Dynamo [?]. The consistency of the operations is determined per-operation basis by the number of read and written replicas. Strong consistency requires to read/write a quorum while on eventual consistency the storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value. [?]. Cassandra resolves conflicts using the timestamp and a last-write-wins policy.

MongoDB and CouchDB are two of document-oriented stores. Their **API!** allows to **CRUD! (CRUD!)** documents. MongoDB has master-slave architecture: one master per shard (a partition of the key-space) and the shard's slaves are eventual-consistent. Therefore, there are no conflicts. On other hand, CouchDB records any changes as a version of the document and allows conflict resolution via programmatic merge.

Graph stores, such as Neo4J, target a specific kind of applications.

Memcached and VoldemortDB are in-memory distributed key-value stores. Memcached does not provide a replication mechanism by default and it is often used to cache web pages. Like CouchDB, VoldemortDB is an eventual consistent store that accepts asynchronous concurrent writes.

The correct **NoSQL!** database depends on the particular application, which has different size of data sets, complexity, rate between writes and reads, consistency requirements and methods to access the data. Column-oriented is beneficial for applications that must access a subset of values. Key-value structure is beneficial for applications that retrieve entire values.

We selected VoldemortDB [?] to implement Shuttle. Voldemort is an open source implementation of Dynamo [?], which is the base of DynamoDB provided on **AWS!**. Voldemort is a key-value store developed and in used by LinkedIn [?]. Voldemort has a put, get, delete, update (**CRUD! API!**) so we avoid the delay of parsing complex queries. Voldemort does not allow range queries, i.e., the user can retrieve only one value each time, so data items accessed by them are known before the request execution and, consequently, we expect fewer false dependencies (Section 1.5.3).

Voldemort accepts asynchronous concurrent writes and treats the result of each modification as a new and immutable version of the data. Versions of the data can conflict if two or more replicas are updated concurrently during a network partition. A network partition occurs when a failure causes the system to be partitioned in multiple sub-systems and two nodes of the system cannot communicate. When the network partition is healed, Voldemort uses the vector clock of each version to determine the causality between those versions [?]. A conflict exists if two versions do not have a causality relation. Voldemort resolves the conflicts during the following read operation. The conflict is solved is application-assisted: the application reads both versions and based on its semantics, the application chooses the correct version. We leverage the semantic reconciliation to solve distinct parallel replay with different execution (Section 1.5.3).

Voldemort keeps the persistent state on a local transactional database: BerkeleyDB, MySQL or an in-memory key/value data structure. It also allows to choose the serialization protocol, for instance: Java Serialization, Avro, Thrift and Protocol Buffers, which is also used by Shuttle as message passing protocol. Since Voldemort is open-sourced therefore we can modify its implementation to include the Shuttle requirements. We used the release 1.9.0 of Voldemort backed by BerkeleyDB and serialized using Protocol Buffers.

On an earlier stage, we considered and attempted to implement Shuttle on MongoDB. However, MongoDB is implemented using C++, does not support conflict solving and it extends the **CRUD! API!** with further complex operations such as scans. A future work may concern the implementation on a column-oriented database, such as Cassandra.

The *Shuttle Storage* stores for each request: the **HTTP!** request, **HTTP!** response, the accessed keys and the start and end instants. Since an application is expected to retrieve hundreds or thousands of million requests, the data set stored in Shuttle Storage is expected to be extremely large. The size of each data item is expected to be on dozens of kilobytes. The write operations are frequent while the read operations occur during the recovery process. Each data item is written twice: once to add the request, response and start/end timestamps and once to add the accessed keys. The storage must be available during the recovery process and may be replicated to a remote site to prevent catastrophic disasters. Conflicts may occur within a data item: one version containing the request/response, the other containing the accessed keys. Therefore, eventual consistency is tolerated if the database man-

agement system is able to solve conflicts within the data item. At last, the fields of each data item are known and immutable.

Taking into account the above requirements, we selected to implement the prototype using Cassandra [?] as Shuttle storage. Cassandra keeps the versions per-column so it can resolve conflicts within the data item. In addition, it allows to choose the number of read and write servers. Due to the simple content format and the required operations, most of distributed databases, including Amazon's DynamoDB, Google's Cloud Datastore, fulfill the defined requirements.

The external storage should be protected, at least, against attacks to integrity and confidentiality. If the integrity of at least one request is affected, then Shuttle can not replay the requests previous to the most recent affected request and the application is unrecoverable if a snapshot **posteriori to following** that period does not exists. For sake of simplicity, we ignore the security features of the Shuttle storage in this implementation as we consider the Shuttle storage to be part of the trusted computing base.

2.2 Normal Execution

Shuttle can operate in one of two states: *normal execution* and *recovery*. During normal execution, Shuttle collects dependencies between requests, stores the accessed **keysdata items**, sets the correct branch to write the data item values and performs snapshots. Figure 2.1 summarizes the messages exchanged between Shuttle's modules during the normal phase. Shuttle identifies each user **HTTP!** request with an unique **SRD!** (**SRD!**) and identifies all database operations of each request with its **SRD!**. The database operations are logged and sent to the manager, which generates the dependency graph. In this Section, we describe the normal execution phase following the path that a request takes to be processed.

2.2.1 Proxy

The proxy retrieves all **HTTP!** user requests and adds the **SRD!** to their headers. The **SRD!** contains the following fields:

1. **RID!** (**RID!**) (long).
2. Branch: the branch accessed by the request (short).
3. Restraint: to support branch change during the recovery phase (boolean).

vou optar por manter os pontos, adicionei o tipo de dados que cada um tem The proxy could be implemented using an by modifying an existing open-source proxy, e.g., HAProxy or Nginx. They are well tested in production and implemented in C. Moreover, they are likely to perform better than a prototype using a higher-level language, such as Java (Section 3.4.1.2). However, for sake of expected simplicity, we implemented a new **HTTP!** proxy using Java. Nevertheless, implementing most part of the **HTTP!** protocol specification in an efficient manner using Java ended up to be a complex task. aqui discuto mas defendendo o C, nao o Scala ou outras libs, dai ter ficado separado...

The proxy is made of three parts: the client-side module, the backend module and the asynchronous request saver (Figure 2.2).

The client side module retrieves **HTTP!** requests, parses their content and modifies their headers. We used the **NIO!** library of Java to implement a TCP server. The server retrieves user requests and

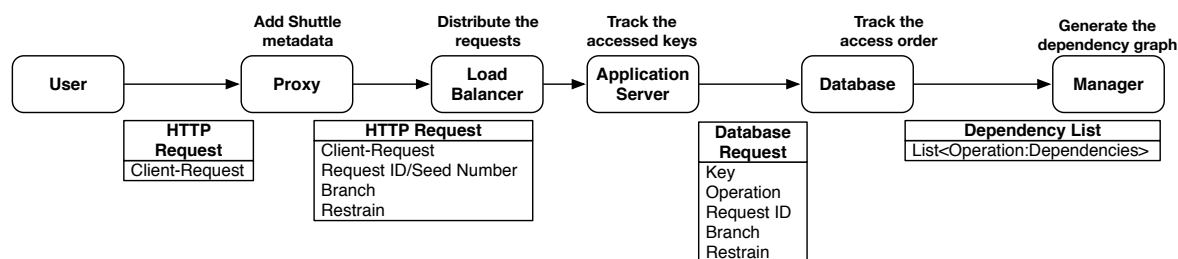


Figure 2.1: Iterations-Interactions between components during normal execution

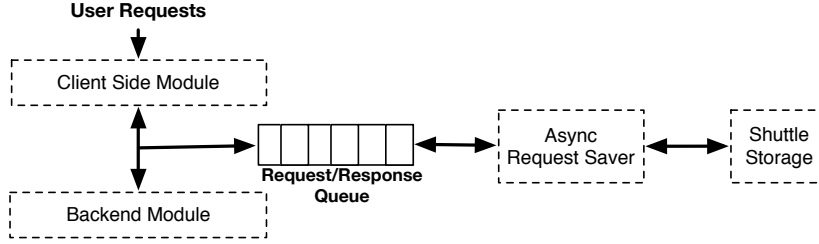


Figure 2.2: Proxy internal modules

modifies their header (Algorithm 1). The values of *branch* and *restrain* are defined by the Shuttle's manager while the **RID!** is a timestamp of the proxy instance. Java provides timestamps with a precision of millisecond. This precision limits the proxy performance to one thousand requests per second. Therefore, we used a counter to identify the same millisecond, up to 1000 requests per millisecond (1 million requests per second). The precision of the timestamp weights on the network and storage overhead. The **RID!** is also used as a timestamp for the application and as a pseudorandom number generator, so the request generates always the same sequence of non-repeating numbers. The fields of **SRD!** (**SRD!**) represent a fixed overhead of 35 ASCII characters, corresponding to 35 bytes, in every request header.

Proxy's TCP client threads send the requests to the load balancer. We use a fixed size thread pool to associate each **HTTP!** response to the correct **HTTP!** request. The thread that retrieves, processes and forwards the request to the load balancer, blocks until its response is retrieved. When the response is retrieved, the Proxy associates a *end-timestamp*. A future work shall implement the proxy as a fully asynchronous proxy. To do so, the application server shall include the **SRD!** in the responses. Since the delay of creating a new TCP connection to the load balancer is considerable, our implementation keeps the TCP session with the load balancer and implements the **HTTP!** keep-alive specification. **HTTP!** keep-alive is a standard that allows a single TCP connection to send and receive multiple **HTTP!** requests/responses, instead of opening a new connection for every request/response pair.

Algorithm 1: Proxy

Input: received user **HTTP!** request
Result: user **HTTP!** request with modified header

- 1 **Initialization:** *establish a HTTP! connection to the load-balancer*
- 2 *package* ← Read TCP package from client
- 3 *request* ← Parse *package* to bound **HTTP!** request
- 4 *SRD* ← Get Shuttle's state: *id, branch, restrain*
- 5 Add *SRD* fields to request's header formatted as:
 ID: *id*
 B: *branch*
 R: *restrain*
- 6 Send *request* to load balancer
- 7 Add *SRD* and *request* to *queue*
 /* Retrieve response */
- 8 *package* ← Read TCP package from load balancer
- 9 *response* ← Parse *package* to bound **HTTP!** response
- 10 *endTimestamp* ← get current system time
- 11 Send *response* to client
- 12 Add *endTimestamp* and *response* to *queue*

Shuttle proxy stores *requests, responses, end – timestamp* and *SRD* in the Shuttle storage. If the data would be sent synchronously to the storage, then the response delay would increase. Therefore, we implemented a message queue (Figure 2.2). The client-side and backend module add the data as a message in the queue. Worker threads dequeue the messages and store the data in Cassandra. This allows asynchronous behavior: requests can proceed before the transmission has finished. The worker threads are wake when the queue reaches a defined threshold or a timeout period expires.

The message queue is implemented as a *synchronized list* in Java.

2.2.2 Application Server

The Shuttle implementation in application servers shall meet two architecture requirements: log the database accesses per request and include the requests' **SRD!** on every database invocation. We want to limit the number of hooks in the application and avoid modifying the database **API!**. The database service **API!** shall remain equal so as the tenants' applications code. Shuttle shall be transparent to application developers.

Shuttle architecture establishes that every database operation must include the **SRD!** (**SRD!**). To do so, we could implement a proxy that modifies every database request or modify the database client **API!** to include the **SRD!** as an argument. For sake of performance, we modified the Voldemort client to include the **SRD!** field on every database operation. We modified the Voldemort messages, which are described using **protobuf!** (**protobuf!**) **IDL!** (Appendix ??). The implemented *interceptor* gets the **SRD!**, which is written in request header, tracks the access and invokes the modified Voldemort client (Algorithm 2) **exactamente isso, devo rescrever?**.

Algorithm 2: Voldemort **API!** interceptor (example of put operation)

```
1 Function put (key, value, store)
2   srd ← trackKeyAccess (key, store, "put")           // get SRD and track
   access (Algo.4)
3   voldemort.put (key, value, srd)
```

In order to extract the **SRD!** from the requests' header, we took in account that **PaaS!** systems deploy applications in application engines. **In-concrete** Specifically, AppScale deploys the Java applications on the servlet engine WildFly [?] (formerly known as JBoss). In addition, most of Web Service frameworks, e.g., **JAX-WS!** (**JAX-WS!**), and **MVC!** (**MVC!**) frameworks, such as Spring, encompass the concept of interceptor chain, also known as filters or handlers. An interceptor chain is a sequence of handlers that contain methods that are invoked before and after the request processing by the application controller.

We implemented **an-a** *request interceptor* in Spring. Application developers only need to add a single line to their applications *web.xml* file to add the developed interceptor to the interceptor chain. A future work may concern the implementation interceptor for other application engines.

We also took into account that each request in Spring is binded to a single thread. Therefore, Shuttle keeps an hash table that associates the **TID!** (**TID!**) with the **SRD!** (**SRD!**) and list of accessed keys of the request that it is processing. The *request interceptor* parses the request header and creates an entry in the hash table (Algorithm 3).

Algorithm 3: Shuttle interceptor: Pre handler

Data: *map* is a static hash table
Input: HTTP user request

```
1 Function preHandle (request)
2   keySet ← new empty list
3   srd ← parse request header
4   tid ← get current thread id
5   entry ← create entry pair: {keySet, srd}
6   map.put (tid, entry)
```

Before every database operation, the database client library invokes the client interceptor to log the access. The interceptor resolves the **TID!** (**TID!**) into the request's **SRD!**, adds the new key to the set of accessed keys of the **SRD!** and returns the **SRD!** (Algorithm 4). The client library appends the **SRD!** in the database operation request.

The post-process interceptor (Algorithm 5) accesses the map extracting the accessed keys and the **SRD!** of the current thread. The accessed keys are stored among the **HTTP!** request/response in the Shuttle storage. In addition, this interceptor also adds the **SRD!** to the response header.

Algorithm 4: Shuttle interceptor

```
1 Function trackKeyAccess (key, store name, operation type)
2   tid ← get current thread id
3   entry ← map.get (tid)
4   entry.addKeyAccess (key, store name, operation type)
5   return entry.srd
```

Algorithm 5: Shuttle interceptor: After completion handler

Data: *map* is a static hash table

Input: HTTP user request and **HTTP!** response

```
1 Function afterCompletion (request, response)
2   entry ← map.remove (tid)
   // add accessed keys of entry to Shuttle Storage
3   shuttleStorage.add (srd, accessedKeys)
4   add srd to response header
```

2.2.3 Database

[The](#) Shuttle database proxy records database operations, selects the correct data item version to access according to the Shuttle state and performs snapshots. The proxy can be implemented in the database management system or as an external **TCP! (TCP!)** proxy that accesses the requests. In this work, we implemented [the](#) proxy as a new library in the database management system. The database management system invokes the library before the execution of every database request. The proxy tracks the operations order to each data item recording an ordered list of the **RID!** that accessed the data item. The operation sequence of each data item is sent to the *manager* to generate the dependency graph (Chapter 2.2.4).

We assume without loss of generality that applications store their state in distributed key-value stores, such as Dynamo [?], where the values are often accessed using a **CRUD! (CRUD!)** API. The simple **API!** reduces the performance overhead to track accesses while the independence between keys turns Shuttle into a scalable service. Shuttle can be extended to support other **NoSQL!** schemes, for instance column-oriented storage like Cassandra [?].

Voldemort is a distributed database store implemented in Java. We choose in-memory and **BDB! (BDB!)** as storage engines and Protocol Buffers as serialization and message passing protocol. Voldemort provides a **CRUD! (CRUD!)** API with 3 methods: get, put and delete. Voldemort accepts asynchronous concurrent writes and treats the result of each modification as a new and immutable version of the data. It detects [versions-version](#) conflicts at read-time and handles them using vector clocks or semantic reconciliation. In this implementation, we assume the replication mechanisms to be disabled.

In [the](#) previous section (Section 2.2.2), we introduced the modified version [of](#) the Voldemort client library, which encompasses the Shuttle interceptor. The interceptor adds the requests' **SRD!** to every database operation. We modified [the](#) format of the Voldemort messages, which are described using **protobuf! (protobuf!) IDL!**, to encompass the **SRD!** (Appendix ??). This technique makes Shuttle transparent for application developers.

The Voldemort implementation has been modified to invoke the Shuttle database proxy on every **API!** call. The proxy architecture is based on the strategy design pattern: two operation schedulers decide how the operations shall be processed. The first scheduler, named *newScheduler*, is used by operations of new incoming requests. The second, named *replayScheduler* and described in Section 2.3.3, is used by operations of requests being replayed (Figure 2.3). Schedulers control the access to data items ordering the operations execution and selecting the correct version of data item to access, i.e., the snapshot and branch of the data item to be read/written by the operation. Voldemort is a key-value store, i.e., each data item is identified by a unique key. We implement the versioned storage concatenating the key with the version's snapshot (a version/snapshot can only be written in one

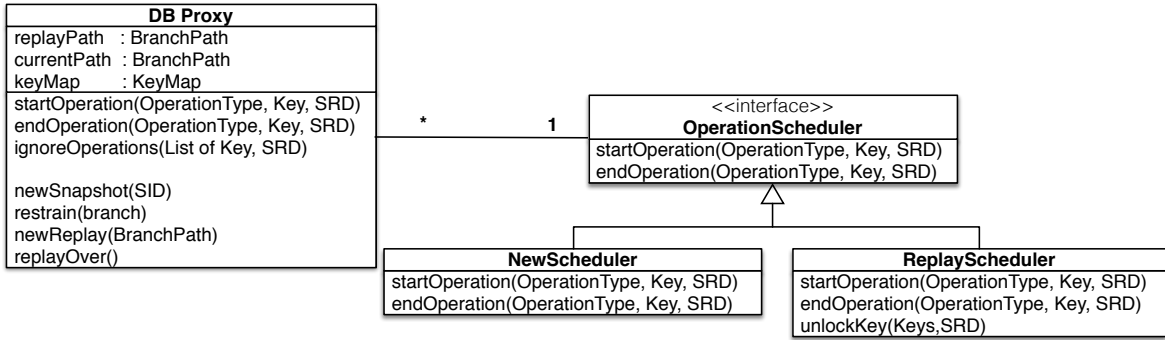


Figure 2.3: UML of Database proxy and schedulers

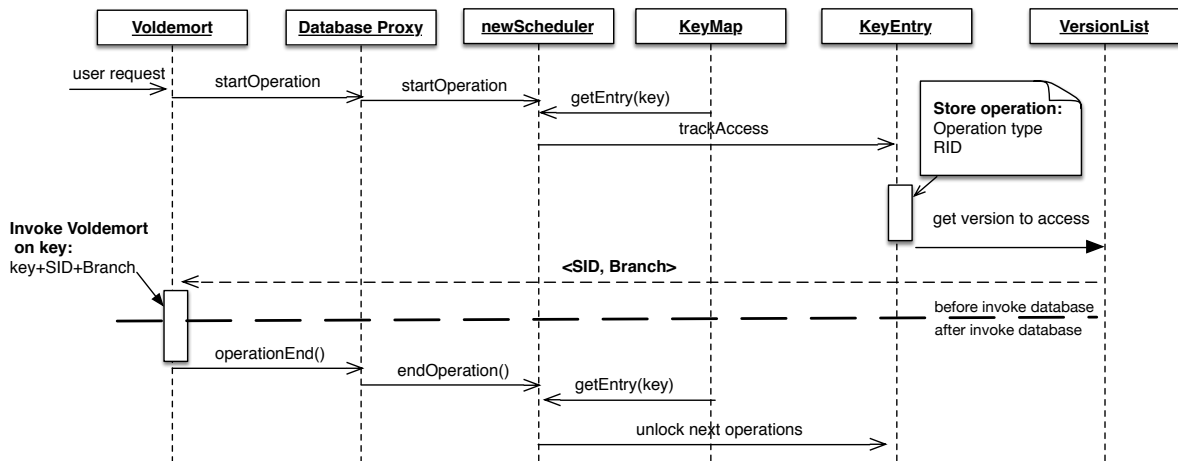


Figure 2.4: Sequence of invocations between the main data structures of database proxy

branch).

Each database proxy maintains two **BranchPath** *branch path*. A **BranchPath** *branch path* of a certain branch is the sequence of non-tampered snapshots between the current snapshot and the root snapshot (Chapter 1.5.6). The first **BranchPath** *branch path*, named *current branch*, refers to the branch used by new incoming requests, while the second, named *replay branch*, wraps the branch used by the requests being replayed.

Before each replay process, the Shuttle manager sends the **BranchPath** *branch path* of the new replay branch, in which the requests will be replayed. At the end of replay process, the database nodes are also notified and the *replay branch* becomes the *current branch*. If the branch of the request is the *replay branch*, then it is being replayed. Otherwise, *its-it is* a new request.

First, we consider operations of new incoming requests. Shuttle tracks operations of new requests and allows a single write or multiple read *operations* per key. To do so, the Shuttle interceptor invokes the *newScheduler*, which invokes the *keyMap* (Figure 2.4). The *keyMap*, which is the main data structure of database proxy, is a concurrent hash table that binds each key to its Shuttle metadata, named *keyMapEntry*. To improve Shuttle's performance, the *keyMap* is internally partitioned to permit concurrent reads and updates.

Each *keyMapEntry* object wraps the Shuttle metadata for one key/data-item: *version list*, *operation list* and *read-write lock* (Figure 2.5). The *version list* contains the list of versions: snapshot in which the entry *as-has* been written. A data item may not be written in every snapshot (Section 1.5.6).

The *operation list* contains the sequence of operations of a certain key (data item). Each operation has an operationType (put, get, delete, get version) and the **RID** of the retrieved requests. The *read-write lock* lock serializes the operation that access the key value allowing concurrent reads or a single write at each time. This pessimistic concurrency control method may decrease the system performance comparing, for instance, with a multi-version concurrency control. An alternative imple-

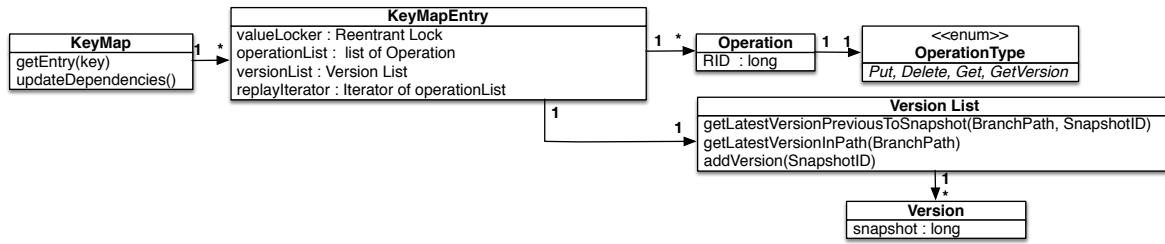


Figure 2.5: KeyMap UML: the entities used to track operations and versions of each key.

mentation can write the **RID!** among the value in the store and track only the read accesses, parsing the accessed **RID!**. The **client-library-client library** may parse the value to get the **RID!** but the write order is relevant. For sake of simplicity, we used a pessimistic concurrency control with a reentrant mutual exclusion lock.

When *newScheduler* is invoked by an operation from a new incoming request, it gets the *keyMapEntry* of the key and attempts to gain the access to its *read-write lock* (Algorithm 6). Then, it adds the operation to the *operation list*. At last, Shuttle gets the correct version to access based on the *BranchPathbranch path* and the *list of versions* of the key (Section 1.5.6).

If the request **RID!** is smaller than the current **SID!** (**SID!**) (line 7), then the request does not belong to the current snapshot and shall access the latest, but previous to the snapshot instant, version of the value that is in the *current BranchPathbranch path*. Otherwise, the request belongs to the current snapshot. Read operations access the latest version (line 12) in the *current BranchPathbranch path*. A write request may create a new version if the version has yet not been written in the current snapshot (**RID!** is bigger than the **SID!** of the latest version) (line 13). Versions are created with the current **SID!** (**SID!**). The key accessed by the request is the concatenation of the original key with the version returned by the Algorithm 6. After the access, the *read-write lock* is unlocked and the next operations proceed. isto é talvez o mais complexo e fundamental de toda a tese, tem de ficar perceptível - discutir se mudamos para o capítulo anterior.

Algorithm 6: Method to track a new request, get the version to access and perform snapshot

```

1 Function trackAccess (operation type, srd, branch path)
2   if type is put or delete then
3     lock.writeLock()
4   else
5     lock.readLock()
6   operationList.add (new Operation(type, srd.rid))
7   if srd.rid < branch path.sid then
8     // Access only the versions of previous snapshots
9     return latest version in (versionlist ∩ branchpath : version < srd.rid)
10  else
11    // Access the latest version
12    latest ← get latest version in (versionlist ∩ branchpath)
13    if type is get then
14      return latest
15    // New version is created if the request belongs to a newer
16    snapshot
17    if branch path.sid > latest.sid then
18      version ← new Version(branch path.sid)
19      versionList.add (version)
20    return version

```

Since the *version list* keeps a pointer to the latest version and the pointer is updated if wrong, the average complexity of the Algorithm 6 is $\mathcal{O}(1)$: get the latest version and check if it belongs to the *BranchPathbranch path*, which is a HashSet. If the *BranchPathbranch path* size becomes a consid-

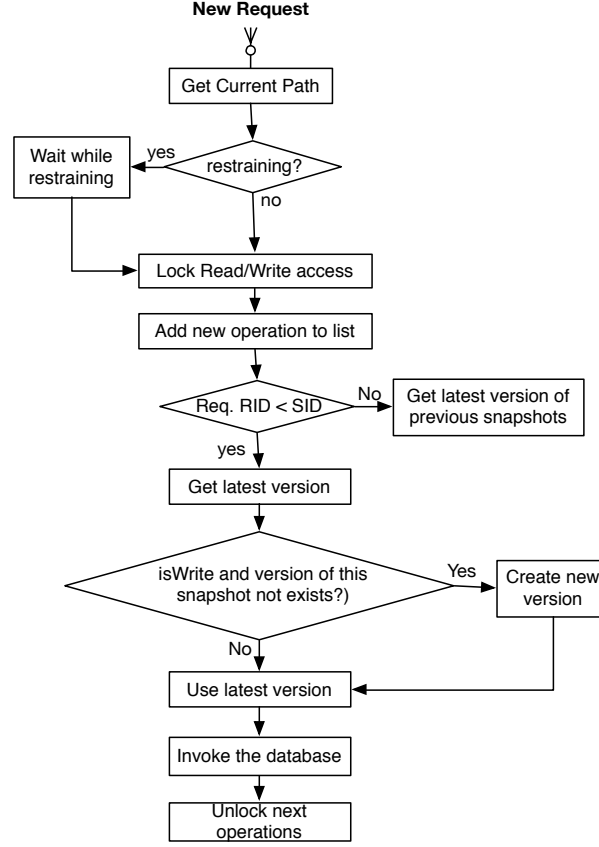


Figure 2.6: Diagram of processing a new database request

erable storage overhead, the **BranchPaths** branch paths and version lists can be implemented as a bitmaps.

Shuttle collects the dependencies between requests transversing the operation list of every *KeyMapEntry* in the *KeyMap*. Each *KeyMapEntry* keeps a pointer to the latest collected dependency in the operation list, so only new operations are transversed.

In order to determine the dependencies between requests, Shuttle firstly lookup for the write operation previous to the latest collected operation (Algorithm 7). Then, it iterates the operations following to the latest collected operation. If an operation of request *A* reads a data item written by an operation of request *B*, then *A* depends **from on** *B*. Dependencies are logged on a hash table that associates the operation **RID!** with the **RID!**s which it depends **from on**. Every database node sends its *dependenciesTables* to the manager periodically.

Algorithm 7: Dependency collection

```

1 Initialization: dependencies  $\leftarrow$  new hash table<Long,Long>
2 foreach entry in keyMap do
3   lastWrite  $\leftarrow$  RID! of the latest write operation previous to the earlier operation to collect
4   foreach operation in entry.operationList do
5     if operation.type == READ then
6       dependencies.add(operation.rid, lastWrite)
7     else
8       lastWrite  $\leftarrow$  operation.rid
9   entry.lastCollected  $\leftarrow$  latest operation in entry.operationList
10 send dependenciesMap to manager
  
```

NoSQL! databases are designed for distributed environments. The proposed dependency tracking and snapshot mechanisms are horizontally scalable: each database node remains independent and each data item is independent from the others. The only locks shared by data items are the

KeyMap partition lock and the KeyMapEntry lock. Adding more partitions, can improve the overall performance. Each node is responsible for logging local dependencies and communicates to a central entity (Manager) to update the dependency graph.

2.2.4 Manager

The ~~manager retrieves dependencies, orders~~ main duties of manager are: retrieve dependencies between database operations and generate the dependency graph; coordinate database nodes to create new database snapshots ~~and coordinates~~; coordinate the recovery process.

~~The manager aims to retrieve database operations and~~ In order to generate the dependency graph, ~~Dependencies~~, dependencies are pushed to the manager by the database nodes. The dependency graph is updated during the normal execution. An alternative approach could pull the dependencies from each database node during the recovery phase and generate the graph during the recovery period.

The dependency graph is a set of vertexes, each one representing a request, and a collection of edges that each connect a pair of vertexes. Each edge specifies an one-way dependency between two requests. Therefore, the dependency graph is a directed graph. The dependency graph is also *not connected*, i.e., it consists of a set of connected subgraphs. Each vertex is uniquely identified by its key, the **RID!** of the request that it represents.

The dependency graph has ~~4~~ four main operations: insert new dependencies, get requests sorted by start instant, get requests dependent from a certain request, get requests from which a certain request depends ~~from on~~.

The first operation inserts new vertexes (requests) and adds edges between the new vertexes and all vertexes from which new vertexes depends ~~from on~~. The second visits every vertex by the order of their key (start instant). The two latest operations visit all vertexes reachable from a certain vertex using the *dependencies to* and the *dependencies from* directions, respectively. Even the dependency graph is, conceptually, a directed graph, the two latest operations are easier to implement on *undirected graphs*.

Two of the main representations of graphs are ou seja: para representar ou guardar um grafo em memória, pode ser matrix ou com ponteiros; são as estruturas de dados para o grafo: adjacent matrix and adjacent list. On a dependency graph, each insert operation requires to create several edges, one per dependency. An edge can be inserted on an adjacency matrix with complexity $\mathcal{O}(1)$. However, we consider the dependency graph to be a *sparse graph* because each request is expected to be dependent from a small part of all requests. Therefore, the adjacent list representation is expected to be more space efficient than the matrix.

The three main implementations of a adjacent list graph are: an indexed array, ~~object-oriented nodes with pointers~~ object oriented, é a estrutura mais fácil: cada nó é uma estrutura com valor e ponteiro para outro nó and a hash table. The vertex keys are expected to be a sparse numeric sequence with irregular number of edges. Therefore an indexed array implementation would not be space efficient. Insert operations in object oriented graphs requires searching for objects to create new edges. However, a graph search algorithm, for instance breadth-first search (BFS), has a time complexity of $\mathcal{O}(|Vertex| + |Edges|)$ in the worst case. Hash table implementations use a hash table to associate each vertex in a graph with an array of adjacent vertexes. To clarify, each hash table object represents a vertex. The object has list of keys that represents the edges.

We implemented the dependency graph as a hash table. Doing so, insert operations have $\mathcal{O}(1)$ complexity. Operations to obtain the first level of dependencies of a certain requests has also have $\mathcal{O}(1)$ complexity. Even so, we also take in consideration that the performance of these operations is influenced by constant factors and by the hashing function and data distribution.

The hash table keys are the **RID! (RID!)**, which is also the start timestamp of the request. The object associated with each key wraps: the end timestamp; the list of requests from which the request depends ~~from on~~ (executed before); the list of requests dependent from the request (to execute after) (Figure 2.7). For sake of simplicity, the graph is kept in the memory of the manager. A production mode implementation, in which the memory limit can become a bottleneck, shall implement the dependency graph in a **DHT! (DHT!)** or a key-value **NoSQL!** store.

The manager updates the dependency graph when a new table of dependencies is retrieved from a database node (Algorithm 8). The retrieved hash table associates each **RID! (RID!)** with the **RID!** of the requests from which the request depends ~~from on~~. The algorithm creates an undirected graph,

Key	Value
RID: long	end : long before : list of RID after : list of RID

Figure 2.7: Entry of the dependency graph HashMap

which can be considered as two directed graphs: the first representing the dependencies from; the seconds the dependencies two. In other words, the graphs are symmetric: one graph can be generated inverting the directions of the edges of the other graph.

Algorithm 8: Add dependency to dependency graph

Input: *dependencies* \leftarrow HashTable<RID, list of RID>

```

1
2 foreach {rid, dependencyList} in dependencies do
3   entry  $\leftarrow$  get or create a GraphEntry in dependency graph for rid
4   entry.addBefore(dependencyList)
5   foreach dependentRID in dependencyList do
6     entry  $\leftarrow$  get or create a GraphEntry in dependency graph for dependentRID
7     entry.addAfter(rid)

```

The manager maintains the *BranchTree*. BranchTree is the implementation of the branching model introduced in Chapter 1.5.6. The BranchTree is implemented as a hash table that associates each branch and its snapshots. The tree model allows tenants to create new branches or snapshots.

In order to create a snapshot, tenants define a future instant in time, t , when the snapshot will occur. The manager forwards the value of t , named **SID!** (**SID!**), to every database instance. Each database node adds a new snapshot with the retrieved **SID!** in the *current* *branchPath**branch path*. The manager insert the **SID!** in the entry of the current branch in the *BranchTree*.

In order to create a new branch and start a new replay process, tenants shall specify a base snapshot. The branching algorithm finds the branch of the selected snapshot and copies the snapshots in the list, which are equal or previous to the selected snapshot, to create a new entry in the BranchTree for the new branch. Then, the algorithm creates a snapshot in the new branch. In other words, the new *BranchPath**branch path* is the concatenation of the new snapshot, in which the replay will be written, and the sequence of snapshots of the branch of the selected snapshots that are equal or previous to it. The new *BranchPath**branch path* is send to the database proxy of all database nodes.

Shuttle's manager prototype retrieves *tenants**tenant's* commands via command line. Each module of Shuttle, including the manager, has a TCP server that retrieves requests from other modules. Requests are formatted using Google's Protocol Buffer [?]. Messages exchanged between the manager and the remaining modules are in Appendix ??.

2.3 Recovery

In this section, we introduce the implementation of the recovery process in Shuttle.

The recovery process begins when the tenant selects a non-tampered snapshot. The manager creates a new branch and sends its *BranchPath**branch path* to all database proxies. The proxies set the new *BranchPath**branch path* as *replay branch* (Chapter 2.2.3). After, the manager generates the sequence of requests to replay, named *execution list*. The replay instances are launched and retrieve this list (Figure 2.8). Then, the replay instances are notified to start replaying the requests. After replaying all the requests, each replay instance notifies the manager. The manager sets the proxy state to *restraining mode* and commands the replay instances to replay the incoming requests retrieved during the recovery period. After replaying the remaining requests, the database nodes are notified to disable the restrain and *replay branch* becomes the *current branch*. At end, the proxy *restraining mode* is disabled and the following new requests are done in the new branch.

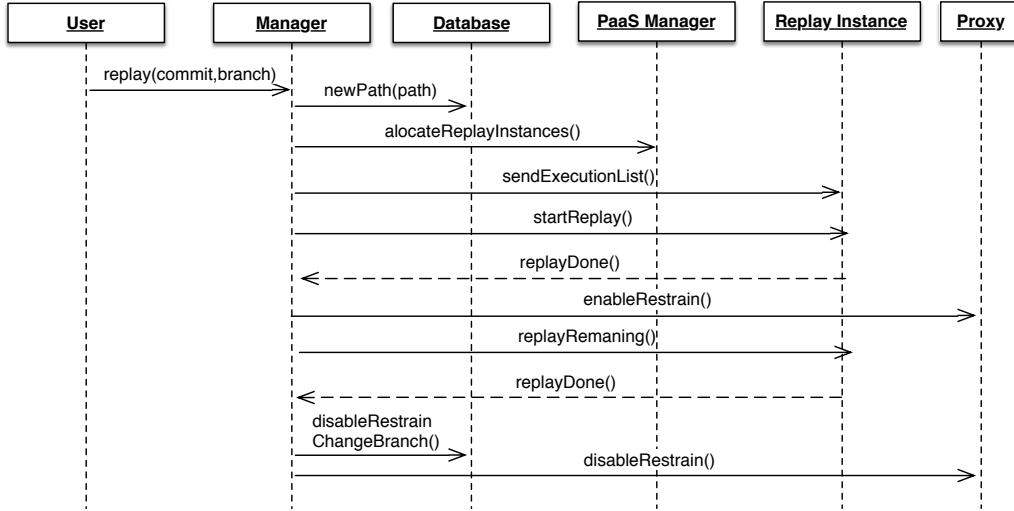


Figure 2.8: Message sequence during replay phase

2.3.1 Manager

Tenants start the recovery process by selecting a snapshot that does not contain the result of any malicious action, ~~start the recovery process~~. Shuttle creates a new branch and sends its BranchPath branch path every database proxy. After, the manager generates the request execution list, i.e., the sequence by the requests shall be replayed.

Shuttle can perform selective or full replay. It also can use clustering to re-execute different sequences of requests in parallel or not (Table 1.2).

Shuttle uses the Algorithm 9 to generate a request execution list for full-replay. The algorithm implements the start-end request ordering approach proposed presented in Section 1.5.3. Since the graph is implemented as a hash table (Section 2.2.4), the keys are not ordered. At recovery time, the keys are ordered and the graph is iterated in order. The complexity of this operation is $\mathcal{O}(n \log n)$. A request must be executed concurrently with a previous request if it began before a previous request ends. Otherwise, the request was not executed concurrently with any of the previous requests.

Algorithm 9: Start-end ordering algorithm

Input: *graph*: dependency graph (Hash Table)

Input: *snapshot*: non-tampered **SID!** (**SID!**) selected by the tenant

Output: *executionList*: sorted list of **RID!** to replay

```

1 Sort graph keys
2 latestExecuting ← -1           // the biggest end timestamp being executed
3 foreach request in graph do
4   if request.start => latestExecuting then
5     add separator to executionList /* none of following requests were
        executed concurrently with previous                               */
6   else
7     add request to executionList
8   latestExecuting ← max(latestExecuting, request.end)
  
```

The clustering mechanism identifies the connected subgraphs of the dependency graph. The algorithm traverses the vertex marking all nodes adjacent adjacent nodes that are reachable from a node. The set of nodes reachable from a node defines a cluster. Then, we choose a non-marked node and repeat the procedure.

When the tenants provide a set of malicious actions, $A_{intrusion}$, Shuttle performs selective replay instead of replaying every request. The selective Selective replay uses the dependencies between requests to expand a provided set of malicious requests, $A_{intrusion}$, to get which requests are needed to be replayed. Set The set of malicious requests is expanded adding all requests dependent from

the initial malicious request set. The traversed nodes are the tainted requests, $A_{tainted}$. Then, the set $A_{tainted}$ is expanded transversing the nodes of the set on the opposite direction: visit requests from which the traversed node depends ~~from-on~~ and that are posterior to the selected snapshot (Figure 1.9).

The list of data items read by the requests in A_{replay} is obtained querying the Shuttle storage. The storage contains the keys accessed per request, which was logged by the application servers (Section 2.2.2). The snapshot versions of these items is loaded and the requests in A_{replay} are replayed. The modified data items are merged with the current database values without copying data values. To do so, the ~~branchPath~~branch path of a selective replay branch contains the current ~~branchPath~~branch path. This mechanism avoids tainting the modified data items and copying them into the current database state. This mechanism does not support runtime recovery but the recovery period is considerably smaller than using full-replay.

2.3.2 Replay Instances

The Shuttle manager requires the **PaaS!** controller to launch a set of *replay instances*. If the **PaaS!** system does not support auto-scaling, the manager requests the **PaaS!** controller to add new database and ~~web-server~~application instances to attend the flow of replayed requests. Otherwise, the auto-scaling mechanism will detect the overload of the ~~web-servers and database~~application and database instances and trigger the creation of new instances. In the first case, the number of ~~web-servers is changeless~~application instances is constant during the replay phase, so the instances send the requests directly to the ~~web-server~~application instance instances, avoiding the overhead of the load-balancer. In the ~~later~~second case, the requests are sent to the load-balancer that distributes the requests through the instances. In the current implementation, requests are sent to the load-balancer because AppScale supports auto-scaling and the number of instances during the replay phase is unknown.

Each replay instance retrieves a set of lists of **RID!** (**RID!**) to execute and the branch in which they shall be executed. Replay instances have a main thread to schedule the execution and a thread pool to execute requests asynchronously. Each thread fetches the **HTTP!** requests from the *Shuttle storage*, modifies their header setting the **SRD!** field *branch* and sends them to the load-balancer. Each response is compared against the response during first execution. An alternative approach can compute the hash of responses in background and compare their values.

If a request has been removed, for instance if it is a malicious request, the replay instance fetches the keys accessed by the malicious request during its first execution and invokes the database proxy to unlock the execution.

The prior evaluation, on serial-replay schema, showed fetching each request before sending is a considerable overhead. We duplicated the throughput and reduced the Shuttle storage usage by using a thread in background to fetch batches of requests from the Shuttle storage. The fetching thread and the execution threads communicate through a in-memory message queue.

The start-end algorithm (Algorithm 9) groups the requests that shall be executed concurrently. The separator marks the end of a group of concurrent request. When a separator is reached in the request list, the execution thread waits until all responses are retrieved. Even so, the number of concurrent requests within a group can overload the servers and database instances. In addition, since application server and replay instances are supported by thread pools, the system reaches a *deadlock* if all threads of one of the thread pools are blocked waiting for further requests. Taking that in consideration, all components in Shuttle shall be asynchronous and message-driven.

As requests are replayed asynchronous, the replay instances must use an end-to-end flow control protocol to avoid the sender send requests too fast for the application servers to receive and process. We implemented a simple flow control protocol, which does not have to lead with message ordering and retransmission because the underlying **TCP!** protocol handles transmission failures. We established two thresholds for the number of pendent responses that define 3 deltas on the request rates. Consequently, the request throughput of each replay instance is dynamic to avoid the servers overload. One of main challenges to evaluate the Shuttle's prototype is to tune the threshold parameters. Future implementations shall access the **PaaS!** controller in order to watch the instances' metrics and do flow control based on it.

2.3.3 Database

In Section 2.2.3, we introduced the ~~implementation of newScheduler~~ of the Shuttle's database proxy. ~~Schedulers intercept requests and define their execution order.~~ The ~~newScheduler~~, which defines the order and logs the new operations. In this ~~Section~~section, we introduce the *replayScheduler*. This scheduler constrains the replayed operations, per data item, to an order consistent with the original execution of the application. **ficou bem?**

Database operations are processed by the *replayScheduler* if the *branch* field of the operation **SRD!** is the same than *replayBranch*. The *replaySchedule* gets the *KeyMapEntry* of the accessed key, which contains the list of operations logged during the execution phase. When the first operation, which is being replayed in current replay branch, accesses the data item, Shuttle creates an operation list iterator and associates it with the key.

~~Operation list iterators control~~ The iterator of each operation list controls the replay process: ~~they allow or block it allows or blocks~~ operations (Algorithm 10). ~~Each~~The iterator keeps a list of operations allowed, executing, waiting or to be ignored. The first allowed operation is the first operation after the snapshot selected by the tenant, i.e., the smallest **RID! (RID!)** that matches the condition **RID! (RID!) \geq snapshot** (line 30).

When a new operation is retrieved, the iterator checks if there are operations executing or allowed to execute. If not, then the iterator transverses more operations of the list and adds them to the *allowed list* (line 9). The algorithm fetches the following consecutive read operations or one write operation. Operations previous to the selected snapshot are not considered.

If an operation is not the following operation, i.e., operation \notin allowed, then the operation is delayed and added to the *waiting* list (line 16). Otherwise, the operation is allowed to proceed and is added to the *executing* list.

After its execution, the operation is removed from the *executing* list and blocked operations (in *waiting list*) are unlocked to check if they can access. For sake of simplicity, the concurrency control is not included in the pseudo-code.

Algorithm 10: Access iterator algorithm

```
1 Initialization:
2   all: list of all operations
3   allowed: list of operations allowed to execute
4   executing: list of operations executing
5   waiting: list of operations waiting to execute
6   ignoring: set of operations to ignore
7   nextOperation  $\leftarrow$  first operation after the snapshot

8 Function startReplayOperation (operation)
9   if allowed is empty and executing is empty then
10    fetchMoreAllowedOperations ()
11   if operation in allowed then
12     // move from allowed list to executing
13     allowed.remove (operation)
14     executing.add (operation)
15   else
16     waiting.add (operation)
17     thread sleep // block operation
18     waiting.remove (operation)
19     // Attempt to execute
20     startReplayOperation (operation)

19 Function fetchMoreAllowedOperations ()
20   if nextOperation is write then
21     allowed.add (nextOperation)
22     nextOperation  $\leftarrow$  iteratorNext (nextOperation)
23   else
24     /* add all read operations until next write */
25     while nextOperation is not write do
26       allowed.add (nextOperation)
27       nextOperation  $\leftarrow$  iteratorNext (nextOperation)

27 Function iteratorNext (operation)
28   /* get next operation bigger than the base snapshot and not ignored */
29   do
30     operation  $\leftarrow$  operation.next
31   while operation.rid < SID or operation.rid  $\in$  ignoring
32   return operation

32 Function endReplayOperation (op)
33   executing.remove (op)
34   foreach operation in waiting do
35     wake operation thread
```

The iterator ignores operations that belong to previous snapshot (line 30), i.e., *operation.rid* < *snapshot*, or to the *ignoring* list. The *ignoring* list contains the operations that will not be executed. If a request is malicious, the replay instances invoke the database **API!** method: *ignoreOperation(RID!, keys)* to ignore operations in all keys accessed by the request. In addition, when application servers retrieve requests and their replay flag is set, at the end of the request re-execution, the post-process interceptor of the database client library fetches the keys accessed by the requests during their first execution (Chapter 2.2.2). **frase mã?** The process compares the keys accessed during the replay phase against the ones accessed during the first execution. The database client library invokes the method *ignoreOperation(RID!, keys)* to ignore the database operations that the request did not perform during the replay phase. This mechanism allows the blocked operations to proceed.

An alternative approach could be: if a request locks, then it invokes a lock, which is shared between all database servers, informing which request should execute before it and waiting. When the request

ends, the lock notifies the blocked processes. This approach does not require to store the accessed keys but requires to share a lock, which can be a bottleneck, and to store which requests have been replayed.

Shuttle supports runtime recovery, i.e., the application remains online during the recovery process (Section 1.5.6). The field *branch* contained in every request allows the database proxy to separate the operations of new incoming requests from the operations of requests being replayed. The first use the *current branch* while the later use the *replay branch*. As introduced above, the *branchPaths* and the scheduler algorithms define the versions which these operations read/write. This allows the new incoming requests to proceed during the recovery process.

At the end of the recovery process, the incoming requests must be switched to the application with fixed state, i.e., the requests should access the data items versions of the *replay branch*. When all requests are replayed, the manager interact with the proxy to set the **SRD!** subfield *restrain* in every following incoming request. The *newScheduler* blocks the execution of new operations that have the *restrain* flat set. Requests performed after the beginning of the replay process are replayed. After, the manager broadcasts a message to every database node to set the *replay branch* as *current branch* and ignore the *restrain* flag. The proxy is notified to stop setting the *restrain* flag. Consequently, the blocked requests and further new incoming requests are processed. This mechanism causes a delay on the request process in exchange for a simplicity and low communication overhead.

2.4 Application Example: Ask

We developed a **QA!** (**QA!**) web application, named *Ask*, to evaluate Shuttle prototype. *Ask* is based on Stack Exchange [?] or *and* Yahoo! Answers [?]. The application data structure is as follows (Figure 2.9). A question has a title, a known number of views by clients, a set of tags, which represent the themes of the question, and a set of answers. Each answer has a text and a number of user votes. Each user can only do a single vote per answer, incrementing or decrementing one unit the number of votes. Each answer has a set of comments.

The data is stored in four Voldemort stores (database schemas): questions, answers, comments and tags. For example, insert a new answer requires to add an entry to the *answers store* and modify the entry of the corresponding question in *questions store* to update the answer list.

Independent user sessions would be trivial to replay in parallel. Therefore, the application semantics implies the following dependencies: a) questions are independent; b) a new answer depends *from-on* previous answers and votes to the same question; c) a new comment depends *from-on* the commented answer; d) a new vote depends *from-on* the voted answer (Table 2.2). In Chapter 3.3, we compare the number of clusters considering or not dependencies between questions with the same tag.

Figure 2.10 represents *an example of* the application dependencies graph generated by Shuttle when two questions, two answers and one comment are created. Questions are independent, since they *don't do not* share a tag. Dashed entries represent read-only requests without consequent writes. A future work may consider to ignore these requests.

The application is implemented using Java Spring [?], which is one of the most used Java enterprise web systems and it is compatible with most of the current **PaaS!** systems (Section 2.1). The implementation is independent of Shuttle. Shuttle does not require the application to be modified, except adding the Spring interceptor (Section 2.2.2).

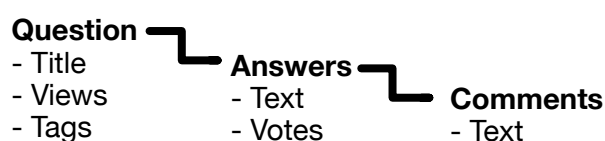


Figure 2.9: Data structure

	Read <u>Read</u>	Write <u>Write</u>	Depend from <u>Depend on</u>
New Question	(Tags)	(Tags) Question	(Tags)
New Answer	Question	Question, Answer	Previous answer to the same question
New Comment	Answer	Answer, Comment	Commented answer
New Vote	Answer	Answer, Vote	Voted answer

Table 2.2: Dependency list

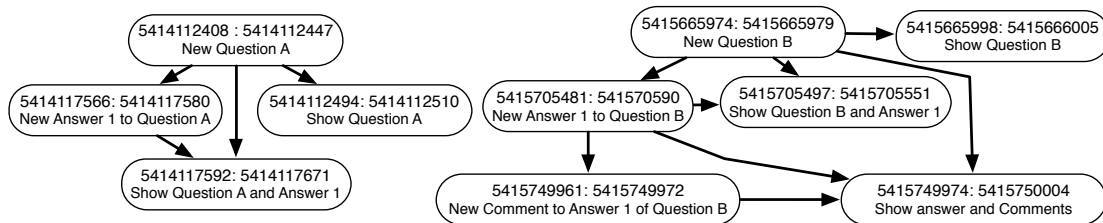


Figure 2.10: Example of a dependency graph generated by Shuttle

2.5 Chapter Summary

We presented the main implementation details of Shuttle prototype in Java. The total number of lines of code, except the unit tests, is summarized in Table 2.3.

The main development challenge is to implement 8 separate modules: TryOut master/slave, Proxy, Manager, Replay master/slave, database client interceptor, database proxy. In addition, the performance of each module is critical to demonstrate that this approach is valid. Therefore, each module is multi-thread and requires implementation of concurrency control. At least, the large data set implies issues with memory allocation.

Components	Lines of code
Proxy	1400
Voldemort	1800
Manager	100
Replay	900
Database Client Interceptor	300
TryOut	3000
Ask	1700
Total	11 000

Table 2.3: Components of Shuttle prototype and an estimate of their complexity, in terms of lines of code

3

Evaluation

In order to evaluate the developed solution, we performed several tests to measure and evaluate the accuracy and performance of Shuttle. Due to public cloud costs, we performed a accuracy test prototype on a single machine and evaluate the prototype performance on a public **CSP!** (**CSP!**).

The following sections detail the steps and decisions towards evaluating the proposed solution, starting by the definition of the developed prototype application and performed tests. The recovery accuracy and performance are evaluated on a set of intrusion scenarios. Finally, we present a cost estimation for Shuttle in **AWS!** [?]. The success of Shuttle is determined by its capability to recover from the intrusion scenarios in a correct, timely and scalable way.

3.1 Prototype Application

pois, concordo com o que diz... deixei aqui porque era o levantamento de requisitos da aplicacao para fazer o teste We considered several application types to evaluate Shuttle. The selected application shall represent a generic web application, retrieve **HTTP!** requests and be deployable on AppScale. It should also fit the **NoSQL!** key-value store model and store its state in Shuttle version of Voldemort [?]. The selected application should imply dependencies between requests but unlike some web applications shall not imply external consistence issues. Accordingly, the application shall read only static data from other services, except the database.

We consider real-world applications such as office productivity, gaming, media content, data aggregation, wikis, diagramming and e-commerce applications (with shopping cart). For sake of simplicity, we developed *Ask* instead of using real-world applications (Section 2.4). Even *Ask* implementation is based on the real-world application StackExchange, its implementation is simpler than it. This allows to analyze the dependencies and reason about the results. We expect to evaluate the dependencies created by different types of application in future.

3.2 Tests Description

We developed a **HTTP!** load testing and benchmarking tool, named *TryOut*, to evaluate Shuttle and measure its overhead simulating a real application load from multiple clients. *TryOut* consists on multiple **HTTP!** clients, which are deployed in various nodes, coordinated by a master node. It measures the average, minimum and maximum response time, the request rate and the throughput of each **HTTP!** client. *TryOut* also compares the database entries and the user responses with the expected

values to measure the precision and recall of our solution. TryOut allows to measure the latency for a given throughput and number of concurrent clients. Requests can be issued asynchronously or synchronously. It also allows to measure the maximum throughput supported by the service. Its main feature, comparing with tools such as Jmeter [?], ab [?], [?] and wehttp [?] is the capability to perform **HTTP!** requests based on data contained in files. This feature allows us to easily test Shuttle with a data set extracted from an application in production.

In order to use real-world data, TryOut performs requests to the developed **QA!** (**QA!**) application (Chapter 2.4) with data extracted from the *Stack Exchange Data Dump* [?]. StackExchange, which is one of the largest **QA!** applications and includes *StackOverflow.com* [?], is ranked 57th for traffic in the world [?]. A 24 hours window has: 36 million page loads, 148 million **HTTP!** requests (1712 requests per second), 267GB retrieved, 1TB sent, 334 million **SQL!** queries [?]. A question takes, in average, 28ms to be rendered. The architecture encompasses 1 load balancer, 2 **SQL!** servers, a Redis [?] (a key-value cache and store) server and 3 web servers. Their architecture scales vertically, for instance the **SQL!** servers have 384 GB of memory with 1.8TB of SSD storage. Since 98,82% of the requests are read requests, the StackExchange infrastructure relies on caching.

StackExchange provides a **SQL!** database dump of each portal of StackExchange. We processed all dump files, a total of 60GB of text files, using MapReduce [?] to extracted the questions, answers, comments and votes. We used four MapReduce jobs to process the data: group comments per answer; group answers per question; output grouped by question; output grouped sorted by time. Doing so, we obtained two files: one grouped by question, the other sorted by they. Tables 3.1 and 3.2 describe the collected data. The [original](#) data was created between 1 August 2008 and 4 May 2014 (181565797230 milliseconds or 2102 days).

	Total
Questions	8 860 649
Answers	15 475 157
Comments	36 486 605
Votes	70 898 355
Tags	74 205
Views	11 097 152 219
Text of questions and answers (UTF-8)	26.76 GB
Text of comments (UTF-8)	5.61 GB

Table 3.1: Data description

	Average	Std. Deviation
Views per question	1252	6216
Tags per question	1	1.41
Answers per question	1	1.73
Comments per answer	2	3.16
Votes per answer	4	17.11
Text per answer (UTF-8)	883 B	1133.9 B
Text per comment (UTF-8)	153 B	122.5 B
Text per question (UTF-8)	1477 B	1951.1 B

Table 3.2: Data description per question

bem visto... eu tinha deixado como borrao para dar para ver a variacao ao longo dos dias. Mudei para umar curva benziana The figure ?? shows the number of new questions, answers and comments per day in the collected data. Considering the collected data, we can set a supremum limit of 182 000 *write requests* per day (2 per second if we consider a constant request rate).

In order to set the throughput expectation for Shuttle, we assume the following request rate ranges for each category of application (Table 3.3). We consider small-scale applications to retrieve less than 5 million requests per day, enterprise scale retrieve between 5 and 100 million and web scale process from 100 million to billions of requests per day. For instance, the maximum request arrival rate of one of the Portuguese Ministry of Finances web applications in 2013 was 836 requests per second [?]. *On the same day, the portal retrieved and* 16 million requests in total [?]. In contrast, the total number of requests per day of StackExchange is 148 million (1712 requests per second). We aim that Shuttle fits on current requirements for small and enterprise services.

Category	Requests per day (Million)
Small	[0, 5[
Enterprise	[5,100[
Portuguese Ministry of Finances	16
Web scale	[100,+∞[
StackExchange	148

Table 3.3: Throughput per web application category

3.3 Accuracy

We In this Section, we evaluate Shuttle's ability to correctly recover applications in different three intrusion scenarios. diz aqui como é que fazemos essa avaliação, ou seja, resume aquilo que aparece a seguir ao longo da secção These scenarios encompass five of the **OWASP! (OWASP!)** Top 10 Application Security Risks: injection flaws, broken authentication, security misconfiguration, missing function level access control and using components with vulnerabilities [?]. We consider three classes of intrusion scenarios:

- Malicious requests that accidentally or maliciously compromise the application
- Software vulnerabilities
- External channels (e.g. **SSH! (SSH!)** connections)

We deployed all Shuttle modules, web-server application instance and database on a single node, a 2.9GHz dual-core Intel i7 with 8GB of memory running the Java 1.7 HotSpot **VM! (VM!)**. This configuration handled, on average, 500 requests per second from a single-thread client. We selected a subset of the *StackExchange Data Dump* [?] containing 100 000 requests originally performed from 31 of July until 12 of September of 2008: 6992 questions, 28 993 questions, 2200 comments, 61 795 votes . Requests were sorted per date, establishing 92 939 dependencies. For the sake of simplicity, we ignored view-only requests because the responses to every write requests imply to read the modified question.

We considered that intrusions happen at 2nd of September, when the database contains 4338 questions, 18286 answers, 422 comments and 38334 votes (61380 requests). The attack is detected in Sep. 12th, assuming a pessimistic delay of 10 days. During this period, the application retrieved 38 620 requests.

Table ?? presents a summary of the accuracy tests. It contains the number of data items tampered by the intrusion (*#intrusion*) and the number of user requests that read data items written by tainted requests or malicious requests (without considering the intrusion requests). Recovery using *full replay* requires to replay every request from the latest snapshot before the intrusion instant until the detection instant: in this example at least 38 620 requests (*#replayed (fr)*). Selective replay only re-executes tainted requests, unless some data item versions need to be recreated. On the worst case, the system does not contain any snapshot and every data read by the tainted requests shall be recreated (*#replayed (sr)*).

deveria pegar no recall e precision? O Recall é 1 porque todos os maliciosos e tainted foram replayed. A precision é que é diferente. Coloquei a precision entre parenteses.

3.3.1 Result consistency

Shuttle aims to support various **PaaS!** applications. Unlike previous works that know the application semantics [?], the semantic of **PaaS!** applications is unknown to Shuttle. In order to evaluate the consistency of the results, we must need to define the results expected by tenants.

For instance, if an attacker created a new question and a legitimate user attempts to create a question with the same title. During the replay, the attacker action will be removed. Should the legitimate request be replayed and create the question? On one hand, the user may have created a question with another title but same content. On the other hand, the request contains the title that the user pretended and he may have not created another question.

Therefore, we argue that developers should consider the application recovery during the design of the application. For instance, if questions are identified by the hash of their text instead of their title, then the legitimate users create a new question only if the text is not equal to the one of other questions. Shuttle does not replay actions which fail during their first execution. Alem de termos a API de consistencia, queria argumentar que a consistencia do recovery process é definida pelos developers. São os developers que devem ter em conta que os pedidos poderao ser repetidos e como é que a aplicacao ira reagir a isso. No exemplo o resultado depende de como é que eles identificam as questoes: hash do conteudo ou titulo. A 1a nao permite conteudos iguais, a segunda nao permite

titulos iguais

1o discuti o que é a consistencia dos resultados, aqui vou fazer o papel de developer/tenant e dizer o que espero do sistema quando ele recupera. Para haver resultados errados, tenho de definir o que é o correcto/esperado. Considering the semantics of our **QA!** prototype, we identified the following most likely attack actions:

1. **Create new question:** The attacker created a new question with a specific title. The following legitimate user requests, which try to create a question with the same title, fail. When the attacker request is removed, the legitimate user requests will perform correctly and create the question. Answers to the removed question will be included in the new question.
2. **New answer:** The attacker answer is removed correctly. Since comments to a non-existing answer fail, the comments are also removed.
3. **New answer to every question:** Same behavior as to a new answer for a specific question.
4. **Modify the question:** The attacker can change the title or the text. The first case is similar to create a new answer. On the second case, Shuttle loads the previous text value.
5. **Modify the answer:** If the attacker changes the text, then Shuttle restores the previous text. However, the comments content may become inconsistent with the answer.
6. **Delete all questions or answers:** Shuttle restores the deleted data. However, the duplicated questions and answers will fail to create.

3.3.2 Malicious Requests

In the first class of scenarios, we consider three cases in which an attacker has stolen an user credential. The attacker uses the stolen credential to ~~personify~~ impersonate a legitimate user and to perform malicious actions. The method used to obtain the credential is out of the scope of this dissertation. This scenario is similar to a legitimated user who makes usage mistakes. We consider attackers:

- a Deleted every question created by the user
- b Deleted every user answer
- c Modified every user answer

o paper tem o mesmo conteudo mas melhor compactado e 1a) The attacker deletes the user's 4 questions, performing 4 delete requests that remove 106 associated comments and answers. The tenant identifies the malicious requests through the user session and selects a snapshot previous to the intrusion instant. Users cannot access deleted questions, so no request is tainted. If Shuttle has a snapshot containing the deleted questions, then *selective replay* does not need to replay any request and merges the deleted questions on the current system state. If the latest snapshot is previous to the creation of the 4 questions, then *selective replay* replays 605 requests to recreate the deleted questions, their answers and votes. The result is merged with the current branch, rebuilding the deleted questions.

To clarify consider the Figure 3.1 in which the *Req. 4* deletes a question with one answer and one comment. *Req. 4* is a malicious request and the values written by it ($Q1, A1, C1$) need to be removed. If tenant selects the *snapshot B*, then the items $Q1, A1, C1$ are restored to the version before the delete operation and no operation is replayed. If the tenant selects the *snapshot A*, then *Req.1, Req.2, Req.3* are replayed to obtain the value of the $Q1, A1, C1$. ficou perceptivel? devo pegar no algoritmo da Section arquitetura selective replay e explicar melhor?

The following scenarios are similar.

1b) Deleting the user's 48 answers implies that 58 data items are deleted and 14 answers and comments are tainted as they execute after the intrusion instant answering and voting without knowing some answers. If a snapshot containing the user answers exists, then the *selective replay* approach replays only 14 tainted answers and comments. Otherwise, it replays 379 requests: the total number of requests to recreate the tainted questions and then merge the result.

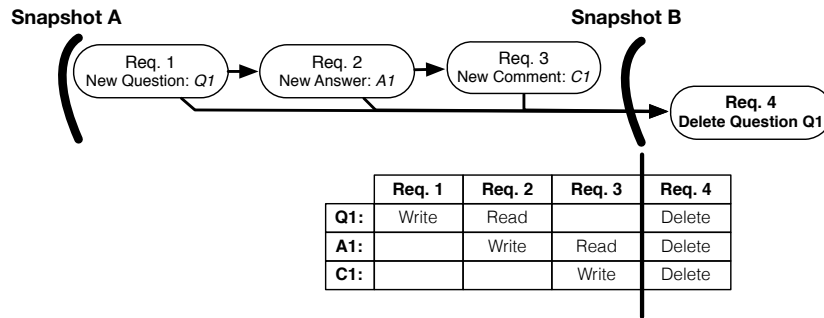


Figure 3.1: Example of request dependencies: a malicious request deletes a question with one answer and one comment

1c) 48 data items are modified while 52 requests are tainted because the users replays, votes and comments the modified questions after the intrusion instant. For recovery, the 52 tainted requests shall be replayed. If Shuttle does not have a snapshot containing the questions, then 253 requests have to be replayed to recreate them.

3.3.3 Software Vulnerability

On the second class, we evaluate intrusion scenarios where software flaws allow attackers to modify the database without authorization. For instance, a code version added a flaw that allows *SQL-Injection*. We consider two independent scenarios where the attacker:

- a Deleted every question
- b Deleted every answer

In 2a), the deleting of every question removes 4 338 data items. In 2b), the questions are preserved but 1 278 answers, votes and comments are tainted as the user did not see the deleted answers.

Instead of identifying the requests that explored the vulnerability, the tenant patches the code to remove the application vulnerability. The instance rejuvenation mechanism terminates the current application instances and deploys the newest application version (Section 1.5.5). Then, they use the *full replay* to repeat all requests since the beginning of usage of the software version with the flaw. Requests that explored the vulnerability fail to execute and a consistent application state is recovered.

Shuttle can also be used to perform preventive updates. Consider the following case: the security team considered that: a) every answer shall be ciphered using the username; b) users without weak-passwords shall not be allowed to answer questions. The development team can rapidly develop a new software version fixing these vulnerabilities. Without Shuttle the system administrator would need to create a script to make the database consistent with the new software version. Shuttle can remove these steps, which require extensive human intervention, by replaying every user request.

3.3.4 External Channels

On the third class of intrusion scenarios, we consider a case where the proxy does not log the attacker actions. In this section, we propose how to recover from a shellshock attack using Shuttle [coloco ref?](#).

Shellsock vulnerability allows an attacker to execute commands that are interpreted by the bash, i.e., perform code injection. When an attacker sends a **HTTP!** request containing the header *User-Agent: () { ;; }; <malicious command>* and the web server passes this header field as a bash variable, the bash interprets the malicious command. For sake of clarity, we omit further details. In summary, this vulnerability allows a **HTTP!** request to run bash commands.

We consider a case in which an attacker used an **HTTP!** request to create a new **SSH!** user account in one application server. Then, the attacker established a **SSH!** connection to an application server using the new user. The attacker stolen a database credentials stored in the application code. The attacker used the credentials to modify at least 2000 data items bypassing Shuttle's database proxy.

As consequence, the database operations are not logged and the number of tainted requests is unknown.

Tenants can use Shuttle as a request logger to detect the string that exploits the shellshock vulnerability: `() {;>};`. If an intrusion is detected, the tenant selects a snapshot previous to the request creation, which does not contain data tampered by the intrusion. Since the application instances can be tampered, Shuttle uses the instance rejuvenation mechanism to terminate these instances and redeploy the application in new instances, in which the shellshock vulnerability is fixed.

Shuttle removes the attack effects and recovers the application consistency performing *full replay*. It loads a full database snapshot instead of undoing operations. As consequence, even non-logged operations are undo. Then it replays every HTTP request posterior to the snapshot instant. Since malicious actions were not logged, they are not replayed. **fica demasiado seco, sem conclusao?**

The described scenario assumes that Shuttle's trusted computing base is not compromised.

3.3.5 Discussion

The number of requests to replay is defined by the snapshot instant: on *full replay* Shuttle replays all requests performed after the intrusion instant, while on *selective replay* Shuttle replays the requests necessary to read the values of the entries before the intrusion and the tainted requests. While *selective replay* seems to have a big advantage comparing with *full replay*, which performs, in these scenarios, at least 38 620 requests, most real applications have more dependencies thus the number of tainted requests is bigger. For instance, if the order between questions with the same tag is considered as a dependency, the number of dependencies rises from 92 939 to 109 118 and the number of independent clusters decreases from 6992 to 56. We plan to further analyze the dependencies established by different applications. **há o risco do arguente dizer: então mas porque é que tiveste de simplificar os testes? não funciona em aplicações complexas?**

3.4 Overhead

The performance evaluation aims to prove Shuttle scalability and capability to support small and enterprise applications (Section 3.2). Its results should match the evaluation metrics bounds that allow to use Shuttle in real environments. We evaluate Shuttle's performance considering the throughput of the application, the size of the logs and the recovery time. We also estimate the cost of deployment of Shuttle on a public cloud provider (AWS [?]).

We run Shuttle in **AWS! (AWS!)**. All instances are connected by gigabit ethernet (780Mbps measured with *iperf*, 0.176ms round-trip time measured with *ping*) and have the *Java 1.7 HotSpot* 64-Bit Server **VM!** version installed. Even when the **Java-JVM! (JVM!)** performance increases as the bytecode is **compile-compiled** by the **JIT! (JIT!)** compiler, we run every test on new **JVM!** instances. They have a 32GB local general purpose **SSD! (SSD!)**. We used the local **SSD!** since its performance on read is better than a 500 Provisioned **IOPS! (IOPS!) SSD!** of 25GB 3.4. We allocated 5GB of heap to the **JVM!** and we disabled database replication **—(each data item is stored in one data node).** **o sistema de replicacao de base de dados foi desactivado**

	putc()	Block Write	Create,Change,Rewrite	getc()	Block Read
EBS SSD	631	83264	44448	1562	105723
Local SSD	619	106546	186204	1577	614790

Table 3.4: Comparison between a EBS Provisioned IOPS (SSD), which supports up to 500 IOPS, and a local SSD instance using bonnie++ (KB/s)

3.4.1 Performance Overhead

In this section, we quantify the overhead imposed by each component of Shuttle.

3.4.1.1 Database

We measured the overhead of Shuttle on database instances. We expected it to be the major impact of Shuttle because Shuttle logs every operation. We used **YCSB!** (YCSB!) [?]]. **YCSB!** was developed to measure the performance of the new generation of cloud data serving systems, such as Voldemort used in Shuttle. We extended **YCSB!** to support the latest version of Voldemort and to use **protobuf!** (protobuf!) as request format.

We used 6 *m3.large* instances (7 **ECU!** (ECU!)s, 2 vCPUs, 2.8 GHz, Intel Xeon E5-2680v2, 3.75 GiB memory, 2 x 16 GiB Storage Capacity) to run each database node. The database service was accessed by two *c3.xlarge* instances (14 **ECU!**s, 4 vCPUs, 2.8 GHz, Intel Xeon E5-2680v2, 7.5 GB of memory, 2 x 40 GB Storage Capacity).

We ran 4 out of 5 **proposed** workloads (Table 3.5) proposed in [?]]. Workload E (Short ranges) is not supported by Voldemort. **deixei apenas na tabela, iria estar a ler a tabela...** Our database consisted of 60 million 1KB record (total database size: 60GB). Each instance thus had an average of 10GB of data, more than it could cache entirely in memory. Read operations retrieve entire records, while update operations modify one of the ten record fields [?]].

Workload	Operations	Key Selection	Application Example
A - Update heavy	Read: 50% Update: 50%	Zipfian	Session store recording recent actions
B - Read mostly	Read: 95% Update: 5%	Zipfian	Photo tagging; add a tag is an update, but most operations are to read tags
C - Read only	Read: 100%	Zipfian	User profile cache, where profiles are constructed elsewhere (e.g., Hadoop)
D - Read latest	Read: 95% Insert: 5%	Latest	User status updates; people want to read the latest

Table 3.5: Workload description

We expected the major Shuttle impact on database to be the read-write lock. Shuttle serializes the requests per-key using a read-write lock. In addition, the underlying **BDB!** (BDB!), which is used by Voldemort as persistent-storage, also uses a read-write lock. However, the results in Figure ?? prove that Shuttle imposes a negligible latency overhead for the various types of workload and request rate.

Shuttle also iterates over the *operation lists* to determine the dependencies and sends them to the *manager*. The instance's CPU usage was 10% in average and the throughput did not reveal any variation when the dependencies are being collected.

Shuttle snapshot mechanism requires creating a new data item in the Voldemort database and creating a new data version. A new version takes 264.077 μs (standard deviation 864.293 μs) to be created, while overwrite a version takes 266.635 μs (std. deviation 577.124 μs).

We conclude that Shuttle database proxy does not represent a significant performance overhead.

3.4.1.2 Proxy

We setup an environment with 3 **AWS!** *c3.xlarge* instances to measure the proxy overhead. The first runs a **HTTP!** benchmark tool *weighttp* [?]]. The second runs a HAProxy (v1.5.3) as load balancer and the Shuttle's proxy. The third runs a WildFly serving a 1KB static file. The benchmark tool performed 2 million requests using 8 threads and 200 clients.

Proxy Overhead: From left to right: direct to the server; only with the load balancer (HAProxy); Shuttle logging but without load balancer; overhead of Shuttle with load balancer when not logging and logging.

Figure ?? represents the throughput limit imposed by Shuttle's proxy. Shuttle's proxy imposes a considerable performance limitation (c) comparing with the load balancer (b). However, the logging mechanism has a negligible influence on the maximum request throughput (column d comparing with column e). In addition, **Table ?? the table in Figure ??** shows that Shuttle imposes a negligible overhead on requests latency considering a constant request income rate of 2500 requests per second. **este**

parágrafo está bem escrito?

	Workload A	Workload B
Shuttle	6325 ops/sec [5.78 ms]	15346 ops/sec[3.62 ms]
No Shuttle	7148 ops/sec [5.07 ms]	17821 ops/sec[3.01 ms]

Table 3.6: Shuttle overhead in terms of application throughput (ops/sec) and response latency (ms)

We conclude that the performance overhead due to logging is negligible but the proxy is a considerable throughput limitation. The main cause is its implementation: while HAProxy is implemented using C and it is heavily optimized, Shuttle's proxy is a prototype implemented in Java. Therefore, we expect a C implementation of proxy can overcome the performance overhead. However, we claim that our prototype implementation is enough for most of small and enterprise services deployed in **PaaS!** because the throughput limit imposed by the proxy is lower than the request rates presented in Table 3.3. Moreover, the recovery time is dependent from the application performance but independent from the proxy's performance.

The overhead of Shuttle on each application server is database client interceptor that logs the accessed keys per request. Our experiments demonstrated that logging and storing the accessed keys implies a negligible overhead on the response latency. **este parágrafo fica um pouco deslocado certo? é só para não ter ficado nada por avaliar**

3.4.1.3 Total

pois, total é uma marca de produtos petroquímicos. Eu não consegui arranjar um nome. Até aqui temos estado a ver parte a parte qual o overhead, aqui é do sistema completo, com tudo a funcionar
We evaluate the overhead of Shuttle by measuring the throughput of the *Ask* application with and without Shuttle (Table 3.6). We do not consider a particular scenario or replay scheme (full/selective) but define instead the number of requests recovered per experiment. We run 6 **AWS! c3.xlarge** instances. We use one client, one instance with Shuttle proxy and a load balancer (HAProxy), three WildFly application servers and one Voldemort database.

We considered two workloads: (A) has 50% reads, 50% inserts and (B) has 95% reads, 5% inserts. Insert operations adds questions, answers, comments and votes of the data sample, while read operations access the latest inserted questions. The insert operations insert the questions, answers, comments and votes of the data sample, while the read operations access the latest inserted questions. We consider a large data sample from *StackExchange Data Dump* [?] with 3 million requests (250812 questions, 335312 answers, 717937 comments, 1695939 votes).

Table 3.6 shows that Shuttle imposes an overhead of 13-20%, which seems reasonable considering the benefits of having it. We believe the main cause of overhead is the current proxy, which is not very optimized. The current version written in Java performs considerably better than a previous version in Python, but we expect to be able to do much better by rewriting it in C. Still, while the proxy and database instances consume a low level of resources, the application instances consumed the maximum of CPU resources available.

Considering the request rates presented in Table 3.3, we claim that a small environment with 6 machines is enough to support a small to medium application running with Shuttle.

In conclusion, we prove that a single proxy architecture, which simplifies the Shuttle design by globally order the requests, is adequate.

3.4.2 Recovery

We measured the performance of the recovery process. We do not consider a particular scenario or replay scheme (full/selective) but define instead the number of requests recovered per experiment. The recovery process can be summarized in the following points:

- Generate the list of requests to replay;
- Launch replay instances; Launch new application servers and database instances, if instance rejuvenation is used;
- Replay the requests.

The main factors of influence on recovery performance are: 1) the number of requests to replay, which depends ~~from~~ on the request arrival rate; 2) the detection delay; 3) the number and type of database operations of each request.

We setup a test environment with 6 *c3.xlarge* instances in an **AWS! VPC! (VPC!)**, sharing the same placement group (Figure 3.2). The first instance runs the *TryOut* testing application. The second runs Shuttle's manager, replay and Shuttle Storage (Cassandra). The third contains the application server WildFly with the application *Ask*. The ~~latest~~ last instance runs the database.

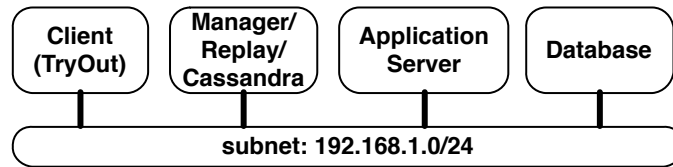


Figure 3.2: Instances deployed on **AWS!**

We considered a workload of 95% reads, 5% inserts (StackExchange network has a read/write ratio of 98.82%). We consider a subset of *StackExchange Data Dump* [?] with 50 000 insert request (1432 questions, 3399 answers, 8335 comments, 36834 votes). Therefore, we consider a total of 1 million requests. The selected questions had 899 407 views in StackExchange. We consider 950 000 view requests.

TryOut performed the 1 million requests in 672 seconds on average (1486 requests/second, standard deviation of 20 seconds) using 25 concurrent threads (Figure ??). The average response latency was 15.82 *ms* (90th percentile of 22.826 *ms* and 95th of 27.721 *ms*). The resultant dependency graph has 1 million entries that establish 9 425 579 dependencies.

3.4.2.1 Graph

The list of requests to replay is generated using the dependency graph. A *c3.xlarge* instance can insert 1 million requests, each depending from other 10 requests, in the graph in 6 seconds (166 000 requests per second). If each request depends ~~from~~ on 2 other requests, then the same process takes 2 seconds. Each new request is represented as a novel entry in a Hash Table and each dependency requires modifying other entry (as the graph implementation is undirected). For web-scale applications, the dependency graph shall be implemented on a distributed hash table because of the graph storage requirements.

The period to generate the execution list is defined by the time to sort the keys of the dependency graph hash table and copy the list. We sort the keys using Java's sorting algorithm, dual-Pivot Quick-sort, which has a asymptotic complexity of $\mathcal{O}(n \log n)$ for most of the data sets. On serial replay mode, the algorithm takes 150 *ms* (standard deviation of 31ms) to sort a dependency graph with 1 million entries and 279 *ms* (standard deviation of 24ms) to copy the sorted list. On clustered replay mode, the clustering algorithm takes on average 2218 *ms* (standard deviation of 1869 *ms*) to determine the 1432 independent clusters of the graph. Then, it takes 533 *ms* (std. dev. of 165 *ms*) to sort the clusters and copy the list.

3.4.2.2 Instance Rejuvenation

The time elapsed to launch a novel *c3.xlarge* instance in **AWS!** depends on the current load of EC2. We measured 25 launches. The average time elapsed is 25 seconds (95th percentile of 40 sec) **não :(não encontro onde é que deixei os valores das samples :(**. The time to deploy and launch the application depends on the application itself. The *Ask* application is launched in less than 1 minute. The process is done by the **PaaS!** controller. We consider these delays negligible comparing with the total recovery period.

3.4.2.3 Request Replay

We performed full-replay considering a single-cluster of 1 million requests in 30 minutes (1717s or 584 requests per second) (Figure ??). The average request rate is 726 requests per second (std. dev. 72). The performance is improved when considering 1432 independent clusters: 9 minutes (544s or 1795 requests per second) (Figure ??). Despite the fact that independent clusters can be replayed concurrently, we consider the maximum of 30 clusters being replayed concurrently. The average throughput is 1966 requests per second (std. dev 224).

The serial replay mode is not capable of fully exploring the application servers. While the first execution is performed using 25 client-threads and clustered replay using 30 client-threads, the serial replays uses one client-thread. We expect to solve this performance issue on future implementations.

3.4.2.4 Restrain

We measured the duration of the restrain period considering two clients with a constant throughput of 400 requests/sec (Figure ??).

Considering serial replay, the average response time of new incoming requests is 3.94 *ms* (95th percentile of 10.34 *ms*, 90th of 7.53 *ms*). Since the serial replay mode is not capable of fully exploring the application servers, the new incoming requests are not affected by the replay process. The replay process takes on average 30 minutes (1850 *ms*). Since the Shuttle takes 30 minutes to replay 1 million requests, the new flow generates 298 thousand new requests. Shuttle takes 18 minutes to replay these new requests. The user requests are suspended until the new requests are replayed and the branch is changed. This delay is considerable because the throughput of new requests is close to the throughput of the requests being replayed. The delay can be reduced by replaying the new requests and then block to replay the newer requests, i.e., creating several phases of replay. fazer varias fases, cada vez mais curtas porque vai compensando a diferenca. In addition, if the replay rate is bigger than the rate of new incoming requests, then the restrain phase is shorter.

If Shuttle performs clustered replay, the application servers are overloaded. Consequently, the average response time of new incoming requests is 5.51 *ms* (95th percentile of 13.56 *ms*, 90th of 10.60 *ms*) and the throughput of the new incoming requests drops from 200 to 95 requests per second. On other hand, the throughput of requests being replayed drops from 1795 to 1686 req./sec. The recovery process takes a total 10 minutes (635 seconds). Therefore, the customers perform 56 000 requests. These requests are replayed in 46 seconds (Figure ??).

When a new application server is added, the throughput of the requests being replayed rises from 1686 to 2517 requests per second (397 seconds to recover), while the response latency to new requests is 4.61 *ms*. Therefore, we prove that adding more servers to perform replay allows to keep the quality of service for the customers.

3.4.2.5 Scalability

We measured ~~the Shuttle capability to scale horizontally, i.e. how Shuttle leverages the horizontal scalability (adding more instances) of PaaS! to reduce the recovery period.~~ We added *c3.large* instances *c3.large* (7 **ECU**s, 2 vCPUs, 2.8 GHz, Intel Xeon E5-2680v2, 3.75 GiB memory, 2 x 16 GiB Storage Capacity). The application server *c3.xlarge* instance was downgraded to *c3.large*.

The serial replay with 2 application servers and 1 database takes 1347 seconds, while with 2 application servers an 2 database instances takes 1303 seconds (Figure ??). At larger scale, considering 2 databases, 6 application servers and 2 Shuttle storage instances, the process takes 1745 *ms*. Therefore, due to its implementation, the serial replay does not scale well. All instances remain with low resource usage. We expect to do much better by rewriting the implementation of the serial replay in the replay instances.

We conclude the application servers to be the main performance harm. The performance of the application servers can be improved. ~~We did not optimize their implementation.~~ If application developers for **PaaS!** optimize their applications to execute more requests per unit of time, then the recovery period is reduced. Most of **PaaS!** controller watch the load of the instances. One area of future development is to dynamically adapt the throughput of the replay instances to the load of the application and database instances.

3.4.3 Space Overhead

The ~~memory and~~ storage overhead is relevant because ~~every request is stored~~ it is a payed cloud resource and Shuttle stores every user request. The space overhead is defined by ~~the variables in~~ Table 3.7.

Module	Contains	Depends from <u>on</u>
Shuttle Storage	Request Response (optional) Accessed keys Start and end instants	request content response content key size and number of database operations per request constant overhead
Database instance	Version list Operation list	number of snapshots per data item number of operations per data item
Dependency graph	Dependencies	number of dependencies per request

Table 3.7: Variables on Shuttle's storage

Requests to static contents, e.g., images, are ignored. For instance, a question page of StackOverflow [?] implies 30 static content requests. Shuttle is not implemented for web-services with large requests, such as file hosting services, because the data would be duplicated in the requests and storage. Shuttle architecture would remain similar on those services but would require an algorithm to reduce the storage overhead by detecting duplicated data.

Most of applications' requests, including headers, vary in size from 200 bytes to over 2KB, depending on the number of application cookies [?]. As applications use more cookies and user agents expand features, typical header sizes of 700-800 bytes ~~is~~ are common [?]. ~~Shuttle adds the SRDI,~~ which has 35 bytes, to every request. Requests of Ask based on the StackExchange Network have an average size of 216 bytes (std. dev of 124 bytes, 95th percentile of 274 bytes, 99th of 494 bytes in a sample of 200 thousand requests, in which 95% are read requests).

Requests and keys are stored in the *Shuttle storage* (Cassandra) while the dependency graph and database operations are kept in manager and database instances. Values at Table 3.8 represent the size of each component in memory ¹ to store the workload, defined above (1 million requests, from which 95% are requests to read a question). No snapshot has been taken and the data is not compressed.

	# objects	size (total) (MB)
Shuttle Storage:		
Request	1 million	212
Response	1 million	8 967
Start/End	2 million	16
Keys	137 million	488
Total		9 684
Database node:	14 593 entries	
Version List	14 593	1.4
Operation List	9 million	277
Total		282
Manager:		
Graph	1 million	718

Table 3.8: Storage used by Shuttle

The start/end instants have a constant size of 16 bytes because each timestamp is a long. The size of the list of keys accessed by the request depends on the key length and the number of accesses. The request of the workload defined above represent 212 MBytes. The main overhead are the requests, as we are storing them complete (the full **HTTP!** pages). Notice that Shuttle has to store the responses

¹<https://code.google.com/p/memory-measurer>

only if the tenants use the **API!** to solve inconsistencies (Section 1.5.7). In addition, the size of the responses can be reduced if applications fetch only data using a **REST! (REST!) API!** instead of the entire page.

Since **HTTP! packages-messages** are similar, we evaluated how a compression technique can reduce the storage usage. While Cassandra stores 9GB of data, the compression algorithm of Cassandra, *lz4* [?], reduces the size in disk to 4.9 GB (including Cassandra's metadata). For instance, considering an arrival rate of 1000 requests per second (86 million per day), a half-year (15.638 billion requests) requires 3.3 TB.

The snapshot mechanism requires to track a new version when a data item is written by the first time after a snapshot. Each version has 10 bytes. The overhead can be reduced implementing the version list as a bitmap. In addition, each database operation implies to store 13 bytes to record its **RID! (RID!)** and type in the *operation list*. The total database storage overhead encompasses synchronization mechanisms and object references. Notice that the storage and performance overhead can be distributed by several instances because the data items are independent.

The dependency graph is a double-linked graph implemented as a Hash Table. Each of its entries represents a request. It contains the start and end instants of the requests and two lists of **RID!**: requests to execute before, requests to execute after. Each entry in the Hash Table has, on average, 718 bytes bytes, and depends *from-on* 1 requests and 5 requests depend *from-on* it (Table 3.9). The start and end instants has 32 bytes. Each dependency has 16 bytes.

For the sake of performance in selective replay, the dependency graph is doubly-linked, a single-linked graph requires, on average, 458 bytes per entry. In order to reduce the storage overhead, a doubly-linked graph can be generated from a single-linked graph.

	# objects	size (bytes)
Start	1	16
End	1	16
Before	1	61
After	5	162
Total	-	718 MB

Table 3.9: Memory used by each entry of dependency graph

Tenants can reduce the storage overhead removing old snapshots, operations and requests. However, they shall take into account that Shuttle needs a snapshot previous to the intrusion instant to recover the application.

In conclusion, the main storage overhead are the **HTTP!** responses. We propose several ways to reduce the storage overhead. Since the data stored by Shuttle is likely to be similar, compression techniques can reduce the storage overhead.

3.5 Monetary Cost

We measured the monetary cost of the intrusion recovery process using a public cloud *providersprovider*. The intrusion recovery costs are come from two sources: storage and computation resources. The following prices represent the current cost of **AWS! (AWS!)** in North Virginia.

We consider an execution of *Ask* application with a constant arrival rate of 250 requests per second (20 million per day) and the storage usage defined in Section 3.4.3. This is a common rate for an enterprise application, for instance the Portuguese Ministry of Finances (Table 3.3). We also define that Shuttle shall allow to recover from attacks that have occurred during the last 3 months. Therefore, Shuttle needs to store 2 billion requests. We assume that the recovery process re-executes 50 million requests, independent of the schema (full/serial replay).

Table 3.10 represents the estimated storage overhead based on the measurements introduced in Section 3.4.3. We do not consider Shuttle to store the responses.

The user requests can be stored in **AWS! (AWS!) S3!**, DynamoDB, Glacier and **EBS! (EBS!)** (Table 3.11). *S3* is a scalable object storage, *EBS* block level storage, *Glacier* low-cost storage service for data archiving and online backup, *DynamoDB* low-latency **NoSQL!** database. Their usage costs are also distinct (Table 3.12).

	1 million	20 million (day)	2 billion (quarter)
Shuttle Storage	716 MB	14.32 GB	1.432 TB
Graph	718 MB	14.36 GB	1.436 TB
Database Nodes	282 MB	5.64 GB	564 GB

Table 3.10: Storage overhead considering different number of stored requests

Data per month	Cost per GB-month			
	S3	Glacier	EBS	DynamoDB
First 1 TB	\$0.0300	0.0100	0.05	0.25
Next 49 TB	\$0.0295	0.0100	0.05	0.25
Next 450 TB	\$0.0290	0.0100	0.05	0.25
Next 500 TB	\$0.0285	0.0100	0.05	0.25
Next 4000 TB	\$0.0280	0.0100	0.05	0.25
Next 5000 TB	\$0.0275	0.0100	0.05	0.25

Table 3.11: Pricing of Amazon Web Services S3, Glacier, EBS and DynamoDB

Glacier has the lowest cost to store data for long period but [the](#) highest per insert operation. In contrast, DynamoDB provides the lowest cost per insert operation, lower latency but the highest per stored gigabyte. The best storage model depends on the application usage pattern. We propose to store the most recent data in *DynamoDB* and archive the data in periodically in *Glacier*. We assume that the data in DynamoDB is archived every day, i.e., each batch stores a day.

Shuttle generates an average of 35 GB per day, which costs \$8.75 per month to store in DynamoDB. Considering a provisioned capacity of 36,000 writes per hour and 180,000 strongly consistent reads per hour, the DynamoDB usage costs \$4.83 per-month.

The Glacier stores 3.433 TB, which represents the archives of the latest quarter, so it costs \$34.33 per month. Since the daily backup can be compressed in a single file, the number of put requests per month is lower than 1 thousand so it costs less than \$0.05. Tenants can retrieve up to 5% of average monthly storage for free. Since we consider an attack that taint 1 million requests, the required download of 2 GB is free. This data is loaded in DynamoDB to perform the replay process.

In total, the storage overhead of Shuttle costs \$47 per month [if the application retrieves 20 million requests per day](#).

Since Shuttle is designed to be integrated with the cloud provider infrastructure, namely the load balancer and the database, the costs of the database and proxy overhead are hard to predict. The Shuttle manager is also included in the underlying cloud provider infrastructure and can be shared by multiple clients.

The Table 3.13 represents the costs of the computation instances in **AWS!**. The data transfer between computing instances and the database is free in the same region using private **IP!** (**IP!**) addresses.

In Section 3.4.1, we demonstrate that Shuttle requires two extra *c3.xlarge* instances: the manager and the proxy. To recover the application, Shuttle can replay 1 million requests in 400 seconds using 5 application servers (*c3.large* instances) and 1 *c3.xlarge* replay instance.

Considering a full-hour, these instances have an associated cost of \$1 per instance-hour, which means that Shuttle can replay 1 million requests by the cost of \$1. Since Shuttle allocating more instances reduces the recovery period, Shuttle leverages the elasticity and pay-per-usage model of cloud computing to provide a cost-efficient intrusion recovery solution.

In conclusion, the cost of Shuttle is dominated by the storage because the replay instances are allocated on demand and paid-per usage.

Notice that the costs are estimations for the application prototype and vary with the type and usage of the tenants applications. In addition, Shuttle aims to be integrated as a service in the public **CSP!** (**CSP!**) infrastructure: the proxy is integrated into the load-balancer and the database proxy is integrated in the database instances. Each manager and Shuttle storage can be shared by several tenants. Therefore, we expect providers to define a pay-per-usage model of Shuttle service. In addition, providers can reduce the costs because the data stored by several tenants is similar thus can be compressed. We do not address the creation of a pay-per-usage model in this document but we

Operation	Usage cost per month			
	S3	Glacier	EBS	DynamoDB
Put	0.005 (1k ops)	0.050 (1k ops)	0.05 (1M ops)	0.0065 (Hour)
Get	0.0004 (1k ops)	0.01 (1 GB)	0.05 (1M ops)	0.0065 (Hour)

Table 3.12: Usage cost of Amazon Web Services S3, Glacier, **EBS** and DynamoDB

	vCPU	ECU	Memory (GiB)	Instance Storage (GiB)	Cost per Hour
t2.small	1	Variable	2	EBS Only	\$0.028
t2.medium	2	Variable	4	EBS Only	\$0.056
m3.medium	1	3	3.75	1x4 SSD	\$0.077
m3.large	2	6.5	7.5	1x32 SSD	\$0.154
m3.xlarge	4	13	15	2x40 SSD	\$0.308
m3.2xlarge	8	26	30	2x80 SSD	\$0.616
c3.large	2	7	3.75	2x16 SSD	\$0.120
c3.xlarge	4	14	7.5	2x40 SSD	\$0.239
c3.2xlarge	8	28	15	2x80 SSD	\$0.478
c3.4xlarge	16	55	30	2x160 SSD	\$0.956
c3.8xlarge	32	108	60	2x320 SSD	\$1.912

Table 3.13: Pricing of Amazon Web Services **EC2** Instances

would like to address it in our future research.

3.5.1 Discussion

In this section we measured the performance overhead, the duration and scalability of the recovery process, the storage requirements and the cost of the used resources. We evaluated Shuttle usage for the prototype application Ask. Shuttle is designed to support various **PaaS** applications with distinct semantics.

The evaluated metrics depend on several aspects of the application. Taking into account all that was mention, we identified the following factors to be the most relevant:

1. Request rate
2. Request/response size
3. Detection delay
4. Dependency between requests
5. Number of operations per request

Clearly, much future research and development will be needed to create a version of Shuttle for web-scale applications. The performance of Shuttle modules can be optimized and tunned to archive better performance and lower storage footprint, reducing the costs and recovery period.

Nevertheless, we validated our single proxy architecture. The proxy imposes a throughput limitation but this limitation is acceptable for most of applications. Moreover, this limitation does not affect the recovery period.

We presented a cost estimation for Shuttle usage. In future, we expect to provide a generic pricing model that **CSP** (**CSP**) can use to provide Shuttle on a pay-per-usage manner.

3.6 Chapter Summary

The main conclusions of the Shuttle evaluation presented in this chapter are:

1. The full replay has lower precision than selective replay but the recovery time is acceptable and allows to remove malicious actions not logged by the proxy.

2. The prototype accuracy and performance is acceptable for small and enterprise applications.
3. It is possible to duplicate the number of requests replayed per second by increasing the number of application servers from 1 to 3.
4. The main storage overhead is the response storage but the data can be compressed.
5. The usage cost of Shuttle is low considering its advantages.

4

Conclusion

This dissertation described Shuttle, the first intrusion recovery system for **PaaS!** (**PaaS!**) that uses a record-and-replay approach. We aim to define a generic architecture that allows **CSP!** (**CSP!**) to offer an intrusion recovery system as a service to their tenants. This service is available without setup and can be provided in a pay-per-usage manner. Our research focused on developing a scalable service to meet the intrusion recovery ~~needs of small and enterprise applications~~ of Cloud tenants. The success of Shuttle will be measured mostly by the impact of two of this dissertation's main contributions: a service integrated in **PaaS!** that leverages the resource elasticity and pay-per-use model and a new process to establish the requests' order during the replay process.

This chapter reflects on these contributions, discusses future work and concludes.

4.1 Conclusions

Our approach to develop an intrusion recovery system for cloud computing focus on ~~restore~~ restoring the applications integrity when intrusions happen, instead of trying to prevent them from happening. Previous works address this problem ~~on at~~ on operating system level [? ?] or distributed systems [?]. These systems ~~can might~~ can be adapted to recover from intrusions ~~on in the~~ on **IaaS!** (**IaaS!**) model. Other works ~~target~~ target aim to recover databases [? ?] and web services [? ? ?], so they can be adapted to recover services delivered ~~on in the~~ on **SaaS!** (**SaaS!**) model. The closest research to ours is Undo for Operators (UO) [?]. Both works address the problem of providing a generic intrusion recovery system. However, UO requires tenants to configure the dependencies and order for each possible operation of the application's protocol. Our approach does not require configuration. Nevertheless, none of the previous works does recovery in cloud environments. ~~Since are responsible for managing the applications and ensure their security, we~~ We consider an intrusion recovery system to be a significant asset for CSPs because they are responsible for managing the **PaaS!** applications and ensure their security. The **PaaS!** model imposes novel challenges because its applications scale and run in various instances backed by distributed databases.

Having the above in mind, we proposed Shuttle, an intrusion recovery service for **PaaS!**, that aims to make **PaaS!** applications operational despite intrusions. Shuttle recovers from software flaws and corrupted requests. As consequence of its architecture, our solution also supports preventive maintenance and to test the application with real user requests. Shuttle loads database snapshots to remove the intrusion effects and replays the legitimate user requests to recover the application integrity.

In order to remove the intrusion effects, we proposed a novel method to perform snapshots of **NoSQL!** databases. This method, which is based on copy-on-write, performs globally transaction-

consistent and incremental snapshots without system downtime. We also proposed a novel process that redeploys tenants' application in new application instances to remove all intrusions in the previous instances and update their software versions to fix previous flaws or prevent future vulnerability exploitations.

In order to restore the application integrity, we proposed a new method to ~~re-executes~~ re-execute requests. This method supports that the requests have been executed concurrently. It replays the requests on the same order than on their first execution and constrains the execution order using the list of operations performed on each database item. Since database items are independent, our replay algorithm is scalable. In addition, we proposed a semantic-reconciliation mechanism to solve conflicts during the recovery process.

Previous works use the dependency graph to establish the requests order. Instead, we use it to create independent clusters of requests that can be re-executed concurrently. We evaluated that this technique reduces the recovery period considerably.

We accomplish intrusion recovery without service downtime using a branching mechanism.

In summary, the proposed architecture is capable of leveraging the resource elasticity and pay-per-use model in **PaaS!** environments to record and launch multiple clients to replay previous non-malicious user requests as concurrently as possible to reduce the recovery time and costs.

Thus, we have achieved the goal we set out at the beginning of this dissertation: help **CSP!** (**CSP!**) customers to recover from intrusions in their applications deployed in **PaaS!** (**PaaS!**).

4.2 Future Work

The following directions are proposed as a development of the present research:

- Consider the dependencies and result on client's browser: a considerable trend in web-application development is moving the application code to the client's browser. How this will affect the dependencies between requests? How will be the consistency of the replay process?
- Consider more database schemas and operations operations: several operations such as *scan* and *append*. What applications can be build using idempotent operations? How the replay algorithms can encompass them?
- Consider the user sessions in the dependency algorithms.
- Fault-tolerance: how to handle when the recovery process fails when instances fail **quando as instancias, por exemplo de replay, falham**? How to handle database replication?
- Research about how the instance rejuvenation can be used in **PaaS!** to prevent attackers from compromise a quorum of replicas.
- Extend the evaluation scenarios tests to more complex intrusions.
- Evaluate the dependencies created by several types of applications.
- Propose mechanisms to prevent intrusions from spreading.
- Propose a pricing model to deliver a intrusion recovery system on a pay-per-usage manner.

Exacto, há aqui ideias que são apenas prolongamentos ou requisitos para colocar em producao. Queria falar consigo para seleccionarmos apenas 80% delas no máximo. In addition, we propose to develop an user interface for tenants and evaluate their experience. A distinct research direction can evaluate how intrusion recover system for cloud, such as Shuttle, can be integrated and affect the recovery procedures of companies. In particular, how these systems can be integrated and used by the practices, for instance, defined in the Service Design and Service Operation aspects of ~~ITIL~~ **ITIL** (**ITIL**). By doing so, we expect to analyze the advantages and disadvantages of using these services.

The major challenges to implement the current prototype were: concurrent and consistent database snapshot, establish accurate requests dependencies, perform parallel actions replay, repair the system state in time, avoid application downtime and keep the application source code unmodified as much as possible.

As future work, we would like to improve the implementation by:

- Optimize the concurrency mechanisms serial replay in the replay instances
- Migrate the dependency graph to a distributed database
- Improve the clustering algorithm
- Integrate the proxy in a load-balancer implementation
- Analyze compression mechanisms to reduce the memory and storage footprint

Shuttle removes intrusions' effects in **PaaS!** applications and restores their state to an intrusion-free state. We propose, to the best of our knowledge, the first intrusion recovery service for **PaaS!** and the first to support **NoSQL!** databases.

