FITCH: Supporting Adaptive Replicated Services in the Cloud

Vinicius Cogo[1], André Nogueira[1], João Sousa[1],
Marcelo Pasin[2], Hans P. Reiser[3], and Alysson Bessani[1]

[1] University of Lisbon, Faculty of Sciences, Portugal
[2] University of Neuchatel, Faculty of Science, Switzerland
[3] University of Passau, Institute of IT-Security and Security Law, Germany

Abstract. Despite the fact that cloud computing offers a high degree of dynamism on resource provisioning, there is a general lack of support for managing dynamic adaptations of replicated services in the cloud, and, even when such support exists, it is focused mainly on elasticity by means of horizontal scalability. We analyse the benefits a replicated service may obtain from dynamic adaptations in the cloud and the requirements on the replication system. For example, adaptation can be done to increase and decrease the capacity of a service, move service replicas closer to their clients, obtain diversity in the replication (for resilience), recover compromised replicas, or rejuvenate ageing replicas. We introduce FITCH, a novel infrastructure to support dynamic adaptation of replicated services in cloud environments. Two prototype services validate this architecture: a crash fault-tolerant Web service and a Byzantine fault-tolerant key-value store based on state machine replication.

1 Introduction

Dynamic resource provisioning is one of the most significant advantages of cloud computing. Elasticity is the property of adapting the amount and capacity of resources, which makes it possible to optimize performance and minimize costs under highly variable demands. While there is widespread support for dynamic resource provisioning in cloud management infrastructures, adequate support is generally missing for service replication, in particular for systems based on state machine replication [8, 23].

Replication infrastructures typically use a static configuration of replicas. While this is adequate for replicas hosted on dedicated servers, the deployment of replicas on cloud providers creates novel opportunities. Replicated services can benefit from dynamic adaptation as replica instances can be added or removed dynamically, the size and capacity of the resources available to replicas can be changed, and replica instances may be migrated or replaced by different instances. These operations can lead to benefits such as increased performance, increased security, reduced costs, and improved legal conformity. Managing cloud resources for a replication group creates additional requirements for resource management and demands a coherent coordination of adaptations with the replication mechanism.

In this paper we present FITCH (*Fault- and Intrusion-Tolerant Cloud computing Hardpan*), a novel infrastructure to support dynamic adaptation of replicated services

in cloud environments. FITCH aggregates several components found in cloud infrastructures and some new ones in a *hybrid architecture* [27] that supports *fundamental operations* required for adapting replicated services considering dependability, performance and cost requirements. A key characteristic of this architecture is that it can be easily deployed in current data centres [3] and cloud platforms.

We validate FITCH with two representative replicated services: a crash-tolerant web service providing static content and a Byzantine fault-tolerant (BFT) key-value store based on state machine replication. When deployed in our FITCH infrastructure, we were able to easily extend both services for improved dependability through proactive recovery [8, 24] and rejuvenation [18] with minimal overhead. Moreover, we also show that both services can reconfigure and adapt to varying workloads, through horizontal (adding/removing replicas) and vertical (upgrading/downgrading replicas) scalability.

FITCH fills a gap between works that propose either (a) protocols for reconfiguring replicated services [20, 21] or (b) techniques for deciding when and how much to adapt a service based on its observed workload and predefined SLA [6, 14, 17]. Our work defines a system architecture that can receive adaptation commands provided by (b) and leverages cloud computing flexibility in providing resources by orchestrating the required reconfiguration actions. FITCH provides a basic interface for adding, removing and replacing replicas, coordinating all low level actions mentioned providing end-to-end service adaptability. The main novel contributions of this paper are:

– A systematic analysis of the motivations and technical solutions for dynamic adaptation of replicated services deployed the cloud (§2);
– The FITCH architecture, which provides generic support for dynamic adaptation of replicated services running in cloud environments (§3 and §4);
– An experimental demonstration that efficient dynamic adaptation of replicated services can be easily achieved with FITCH for two representative services and the implementation of proactive recovery, and horizontal and vertical scalability (§5).

2 Adapting Cloud Services

We review several technical solutions regarding dynamic service adaptation and correlate them with motivations to adapt found in production systems, which we want to satisfy with FITCH.

Horizontal scalability is the ability of **increasing or reducing the number of computing instances** responsible for providing a service. An increase – scale-out – is an action to deal with peaks of client requests and to increase the number of faults the system can tolerate. A decrease – scale-in – can save resources and money. Vertical scalability is achieved through upgrade and downgrade procedures that respectively **increase and reduce the size or capacity of resources allocated** to service instances (e.g., Amazon EC2 offers predefined categories for VMs – small, medium, large, and extra large – that differ in CPU and memory resources [2]). Upgrades – scale-up – can improve service capacity while maintaining the number of replicas. Downgrades – scale-down – can release over-provisioned allocated resources and save money.

Moving replicas to different cloud providers can result in performance improvements due to different resource configurations, or financial gains due to different prices

and policies on billing services, and is beneficial to prevent vendor lock-in [4]. **Moving service instances close to clients** can bring relevant performance benefits. More specifically, logical proximity to the clients can reduce service access latency. **Moving replicas logically away from attackers** can increase the network latency experienced by the attacker, reducing the impact of its attacks on the number of requests processed (this can be especially efficient for denial-of-service attacks).

Replacing faulty replicas (crashed, buggy or compromised) is a reactive process following fault detections, which replaces faulty instances by new, correct ones [25]. It decreases the costs for the service owner, since he still has to pay for faulty replicas, removing also a potential performance degradation caused by them, and restores the service fault tolerance. **Software replacement** is an operation where software such as operating systems and web servers are replaced in all service instances at run-time. Different implementations might differ on performance aspects, licensing costs or security mechanisms. **Software update** is the process of replacing software in all replicas by up-to-date versions. Vulnerable software, for instance, must be replaced as soon as patches are available. New software versions may also increase performance by introducing optimized algorithms. In systems running for long periods, long running effects can cause performance degradation. Software rejuvenation can be employed to **avoid such ageing problems** [18].

3 The FITCH Architecture

In this section, we present the architecture of the FITCH infrastructure for replicated services adaptation.

3.1 System and Threat Models

Our system model considers a usual cloud-based Infrastructure-as-a-Service (IaaS) with a large pool of physical machines hosting user-created virtual machines (VMs) and some trusted infrastructure for controlling the cloud resources [3]. We assume a *hybrid distributed system* model [27], in which different components of the system follow different fault and synchrony models. Each machine that hosts VMs may fail arbitrarily, but it is equipped with a trusted subsystem that can fail only by crashing (i.e., cannot be intruded or corrupted). Some machines used to control the infrastructure are trusted and can only fail by crashing. In addition, the network interconnecting the system is split into two isolated networks, here referred to as *data plane* and *control plane*.

User VMs employ the *data plane* to communicate internally and externally with the internet. All components connected to this network are untrusted, i.e., can be subject to Byzantine faults [8] (except the service gateway, see §3.3). We assume this network and the user VMs follow a *partially synchronous* system model [16]. The *control plane* connects all trusted components in the system. Moreover, we assume that this network and the trusted components follow a *synchronous* system model with bounded computations and communications. Notice that although clouds do not employ real-time software and hardware, in practice the over provision of the control network associated with the use of dedicated machines, together with over provisioned VMs running with

high priorities, makes the control plane a "de facto" synchronous system (assuming sufficiently large time bounds) [22, 25].

3.2 Service Model

FITCH supports replicated services running on the untrusted domain (as user VMs) that may follow different replication strategies. In this paper, we focus on two extremes of a large spectrum of replication solutions.

The first extreme is represented by *stateless services* in which the replicas rely on a shared storage component (e.g., a database) to store their state. Notice that these services do have a state, but they do not need to maintain it coherently. Service replicas can process requests without contacting other replicas. A server can fetch the state from the shared storage after recovering from a failure, and resume processing. Classical examples of stateless replicated services are web server clusters in which requests are distributed following a load balancing policy, and the content served is fetched from a shared distributed file system.

The other extreme is represented by consistent *stateful services* in which the replicas coordinate request execution following the *state machine replication* model [8, 23]. In this model, an arbitrary number of client processes issue commands to a set of replica processes. These replicas implement a stateful service that changes its state after processing client commands, and sends replies to the issuing clients. All replicas have to execute the same sequence of commands, which requires the use of an agreement protocol to establish a total order before request execution. The Paxos-based coordination and storage systems used by Google [9] are examples of services that follow this model.

One can fit most popular replication models between these two extreme choices, such as the ones providing eventual consistency, used, for example, in Amazon's Dynamo [13]. Moreover, the strategies we consider can be used together to create a dependable multi-tier architecture. Its clients connect to a stateless tier (e.g., web servers) that executes operations and, when persistent state access is required, they access a stateful tier (e.g., database or file system) to read or modify the state.

3.3 Architecture

The FITCH architecture, as shown in Fig. 1, comprises two subsystems, one for controlling the infrastructure and one for client service provisioning. All components are connected either with the control plane or the data plane. In the following we describe the main architectural components deployed on these domains.

The **trusted domain** contains the components used to control the infrastructure. The core of our architecture is the *adaptation manager*, a component responsible to perform the requested dynamic adaptations in the cloud-hosted replicated services. This component is capable of inserting, removing and replacing replicas of a service running over FITCH. It provides public interfaces to be used by the *adaptation heuristic*. Such heuristic defines when and plans how adaptations must be performed, and is determined by human administrators, reactive decision engines, security information event managers and other systems that may demand some dynamic adaptation on services.

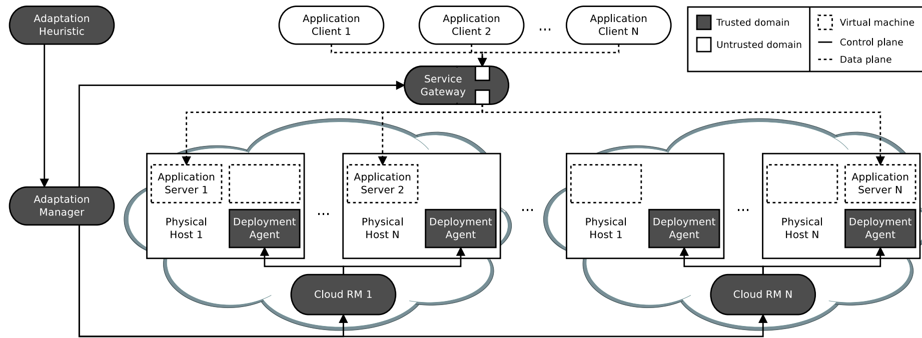


Fig. 1. The FITCH architecture.

The *service gateway* maintains the group membership of each service and supports pluggable functionalities such as proxy and load balancing. It works like a lookup service, where new clients and instances request the most up-to-date group of replicas of a given service through the data plane. The adaptation manager informs the service gateway about each modification in the service membership through the control plane. The service gateway thus is a special component connected to both networks. This is a reasonable assumption as all state updates happen via the trusted control plane, whereas clients have read-only access. Moreover, having a trusted point of contact for fetching membership data is a common assumption in dynamic distributed systems [20].

Cloud resource managers (RM) provide resource allocation capabilities based on requests from the adaptation manager in order to deploy or destroy service replicas. The adaptation manager provides information about the amount of resources to be allocated for a specific service instance and the VM image that contains all required software for the new replica. The cloud RM chooses the best combination of resources for that instance based on requested properties. This component belongs to the trusted domain, and the control plane carries out all communication involving the cloud RM.

The *deployment agent* is a small trusted component, located inside cloud physical hosts, which is responsible to guarantee the deployment of service instances. It belongs to the trusted domain and receives deployment requests from cloud RM through the control plane. The existence of this component follows the paradigm of hybrid nodes [27], previously used in several other works (e.g., [15, 22, 24, 25]).

The **untrusted domain** contains the components that provide the adaptive replicated service. *Application servers* are virtual machines used to provide the replicated services deployed on the cloud. Each of these user-created VMs contains all software needed by its service replica. Servers receive, process and answer client requests directly or through the service gateway depending on the configuration employed.

Cloud physical hosts are physical machines that support a virtualization environment for server consolidation. These components host VMs containing the replicas of services running over FITCH. They contain a hypervisor that controls the local resources and provides strong isolation between hosted VMs. A cloud RM orchestrates such environment through the control plane.

Application clients are processes that perform requests to service instances in order to interact with the replicated service. They connect to the service gateway component to discover the list of available replicas for a given service (and later access them directly) or send requests directly to the service gateway, which will forward them to the respective service instances. The application clients can be located anywhere in the internet. They belong to the untrusted domain and communicate with the application servers and gateway through the data plane.

3.4 Service Adaptation

The FITCH architecture described in the previous section supports adaptation on deployed replicated services as long as the adaptation manager, the gateway and some cloud resource managers are available. This means that during unavailability of these components (due to an unexpected crash, for instance), the replicated service can still be available, but adaptation operations are not. Since these services are deployed on a trusted and synchronous subsystem, it is relatively simple to implement fault tolerance for them, even transparently using VM-based technology [12].

The FITCH infrastructure supports three basic adaptation operations for a replicated service: *add*, *remove* and *replace* a replica. All adaptation solutions defined in §2 can be implemented by using these three operations. When the request to adapt some service arrives at the adaptation manager, it triggers the following sequence of operations (we assume a single replica is changed, for simplicity):

1. If adding or replacing:
   - 1.1. The adaptation manager contacts the cloud RM responsible for the physical host that matches the requested criteria for the new replica. The cloud RM informs the deployment agent on the chosen physical host asking it to create a new VM with a given image (containing the new replica software). When the VM is launched, and the replica process is started, the deployment agent informs the cloud RM, which informs the adaptation manager that a new replica is ready to be added to the service.
   - 1.2. The adaptation manager informs the gateway that it needs to *reconfigure* the replicated service to add the newly created replica for the service. The gateway invokes a reconfiguration command on the service to add the new replica.[1] When the reconfiguration completes, the gateway updates the current group membership information of the service.
2. If removing or replacing:
   - 2.1. The adaptation manager informs the gateway that it needs to *reconfigure* the replicated service to remove a service replica. The gateway invokes a reconfiguration command on the service to remove the old replica.[1] When the reconfiguration completes, the gateway updates the group membership of the service.
   - 2.2. The adaptation manager contacts the cloud RM responsible for the replica being removed or replaced. The cloud RM asks the deployment agent of the physical host in which the replica is running to destroy the corresponding VM. At the end of this operation, the cloud RM is informed and then it passes this information to the adaptation manager.

[1] The specific command depends on the replication technique and middleware being used by the service. We assume that the replication middleware implements a mechanism that ensures a consistent reconfiguration (e.g., [20, 21]).

Notice that, for a replica replacement, the membership needs to be updated twice, first adding the new replica and then removing the old one. We intentionally use this two-step approach for all replacements, because it simplifies the architecture and is necessary to guarantee services' liveness and fault tolerance.

4 Implementation

We implemented the FITCH architecture and two representative services to validate it. The services are a crash fault-tolerant (CFT) web service and a consistent BFT key-value store.

FITCH. Cloud resource managers are the components responsible for deploying and destroying service VMs, following requests from the adaptation manager. Our prototype uses OpenNebula, an open source system for managing virtual storage, network and processing resources in cloud environments. We decided to use Xen as a virtualization environment that controls the physical resources of the service replicas. The deployment agent is implemented as a set of scripts that runs on a separated privileged VM, which has no interface with the untrusted domain.

A service gateway maintains information about the service group. In the stateless service, the service gateway is a load balancer based on Linux Virtual Server [29], which redirects client requests to application servers. In our stateful service implementation, an Apache Tomcat web server provides a basic service lookup.

Our adaptation manager is a Java application that processes adaptation requests and communicates with cloud RMs and the service gateway to address those requests. The communication between the adaptation manager and cloud resource managers is done through OpenNebula API for Java. Additionally, the communication between the adaptation manager and the service gateway is done through secure sockets (SSL).

In our implementation, each application server is a virtual machine running Linux. All software needed to run the service is present in the VM image deployed at each service instance. Different VLANs, in a Gigabit Ethernet switch, isolate data and control planes.

Stateless service. In the replicated stateless web service, each client request is processed independently, unrelated to any other requests previously sent to any replica. It is composed of some number of replicas, which have exactly the same service implementation, and are orchestrated by a load balancer in the service gateway, which forwards clients requests to be processed by one of the replicas.

Stateful service. In the key-value store based on BFT state machine replication [8], each request is processed in parallel by all service replicas and a correct answer is obtained by voting on replicas replies. To obtain such replication, we developed our key-value store over a Byzantine state machine replication library called BFT-SMaRt [5]. For the purpose of this paper, it is enough to know that BFT-SMaRt employs a leader-based total order protocol similar to PBFT [8] and that it implements a reconfiguration protocol following the ideas presented by Lamport *et al.* [20].

Adaptations. We implemented three adaptation solutions using the FITCH infrastructure. Both services employ *proactive recovery* [8, 24] by periodically replacing a replica

by a new and correct instance. This approach allows a fault-tolerant system to tolerate an arbitrary number of faults in the entire service life span. The window of vulnerability in which faults in more than f replicas can disrupt a service is reduced to the time it takes all hosts to finish a recovery round. We use *horizontal scalability* (adding and removing replicas) for the stateless service, and we analyse *vertical scalability* (upgrading and downgrading the replicas) of the stateful service.

5 Experimental Evaluation

We evaluate our implementation in order to quantify the benefits and the impact caused by employing dynamic adaptation in replicated services running in cloud environments. We first present the experimental environment and tools, followed by experiments to measure the impact of proactive recovery, and horizontal and vertical scalability.

5.1 Experimental Environment and Tools

Our experimental environment uses 17 physical machines that host all FITCH components (see Table 1). This cloud environment provides three types of virtual machines – *small* (1 CPU, 2GB RAM), *medium* (2 CPU, 4GB RAM) or *large* (4 CPU, 8GB RAM). Our experiments use two benchmarks. The stateless service is evaluated using the WS-Test [26] web services microbenchmark. We executed the echoList application within this benchmark, which sends and receives linked lists of twenty 1KB elements. The stateful service is evaluated using YCSB [11], a benchmark for cloud-serving data stores. We implemented a wrapper to translate YCSB calls to requests in our BFT key-value store and used three workloads: a read-heavy workload (95% of GET and 5% of PUT [11]), a pure-read (100% GET) and a pure-write workload (100% PUT). We used OpenNebula version 2.0.1, Xen 3.2-1, Apache Tomcat 6.0.35 and VM images with Linux Ubuntu Intrepid and kernel version 2.6.27-7-server for x86_64 architectures.

5.2 Proactive Recovery

Our first experiment consists in replacing the entire set of service replicas as if implementing software rejuvenation or proactive/reactive recovery [8, 18, 22, 25]. The former is important to avoid software ageing problems, whereas the latter enforces service's fault tolerance properties (for instance, the ability to tolerate f faults) [24].

Component	Qty.	Description	Component	Qty.	Description
Adaptation Manager	1	Dell PowerEdge 850 Intel Pentium 4 CPU 2.80GHz	Client (YCSB)	1	Dell PowerEdge R410 Intel Xeon E5520
Client (WS-Test)	5	1 single-core, HT 2.8 GHz / 1 MB L2 2 GB RAM / DIMM 533MHz	Service Gateway	1	2 quad-core, HT 2.27 GHz / 1 MB L2 / 8 MB L3 32 GB / DIMM 1066 MHz
Cloud RM	3	2 x Gigabit Eth. Hard disk 80 GB / SCSI	Physical Cloud Host	6	2 x Gigabit Eth. Hard disk 146 GB / SCSI

Table 1. Hardware environment.

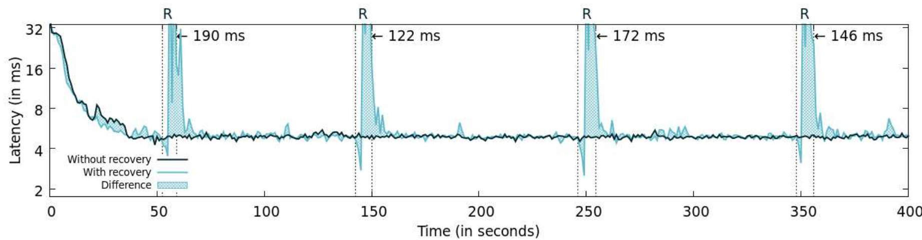


Fig. 2. Impact on a CFT stateless service. Note that the y-axis is in logarithmic scale.

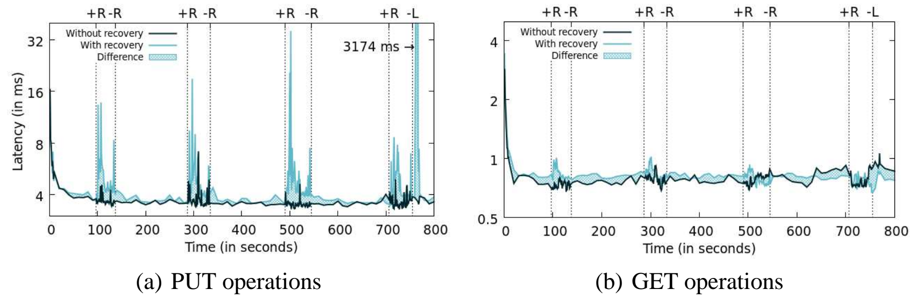
The idea of this experiment is to recover all n replicas from the service group, one-by-one, as early as possible, without disrupting the service availability (i.e., maintaining $n - f$ active replicas). Each recovery consists of creating and adding a new replica to the group and removing an old replica from it. We perform n recoveries per experiment, where n is the number of service replicas, which depends on the type and number of faults to be tolerated, and on the protocol used to replace all replicas. The time needed to completely recover a replica can also vary for each reconfiguration protocol.

Impact on a CFT stateless web service. Our first application in this experiment is a stateless web service that is initially composed of 4 large (L) replicas (which can tolerate 3 crash faults). The resulting latencies (in ms) are presented in Fig. 2, with and without the recovering operations in an experiment that took 400 s to finish. In this graph, each replica recovery is marked with a "R" symbol and has two lines representing the beginning and the end of replacements, respectively. The average of service latency without recovery was 5.60 ms, whereas with recovery was 8.96 ms. This means that the overall difference in the execution with and without recoveries is equivalent to 60% (represented in the filled area of Fig. 2). However, such difference is mostly caused during replacements, which only happens during 7.6% of the execution time.

We draw attention to three aspects of the graph. First, the service has an initial warm-up phase that is independent of the recovery mechanism, and the inserted replica will also endure such phase. This warm-up phase occurs during the first 30 s of the experiment as presented in Fig. 2. Second, only a small interval is needed between insertions and removals, since the service reconfigures quickly. Third, the service latency increases 20- to 30-fold during recoveries, but throughput (operations/s) never falls to zero.

Impact on a BFT stateful key-value store. Our second test considers a BFT key-value store based on state machine replication. The service group is also composed of 4 large replicas, but it tolerates only 1 arbitrary fault, respecting the $3f + 1$ minimum required by BFT-SMaRt. Fig. 3 shows the resulting latencies with and without recovery, regarding (a) PUT and (b) GET operations. The entire experiment took 800 s to finish.

We are able to show the impact of a recovery round on service latency by keeping the rate of requests constant at 1000 operations/s. In this graph, each replica recovery is divided into the insertion of a new replica (marked with "+R") and the removal of an old replica (marked with "-R"). Removing the group leader is marked with "-L".



(a) PUT operations (b) GET operations

Fig. 3. Impact on a BFT stateful key-value store. Note that the y-axis is in logarithmic scale.

The average latency of PUT operations without recovery was 3.69 ms, whereas with recovery it was 4.15 ms (a difference of 12.52%). Regarding GET operations, the average latency without recovery was 0.79 ms, whereas with recovery was 0.82 ms (a difference of 3.33%).

We draw attention to six aspects of these results. First, the service in question also goes through a warm-up phase during the first 45 s of the experiment. Second, the service needs a bigger interval between insertion and removal than the previous case because a state transfer occurs when inserting a new replica. Third, the service loses almost one third of its capacity on each insertion, which takes more time than in the stateless case. Fourth, the service stops processing during a few seconds (starting at 760 s in Fig. 3(a)) when the leader leaves the group. This unavailability while electing a new leader cannot be avoided, since the system is unable to order requests during leader changes. Fifth, client requests sent during this period are queued and answered as soon as the new leader is elected. Finally, GET operations do not suffer the same impact on recovering replicas as PUT operations do because GET operations are executed without being totally ordered across replicas, whereas PUT operations are ordered by BFT-SMaRt's protocol.

5.3 Scale-out and Scale-in

Horizontal scalability is the ability of increasing or reducing the number of service instances to follow demands of clients. In this experiment, we insert and remove replicas from a stateless service group to adapt the service capacity. The resulting latencies are presented in Fig. 4. The entire experiment took 1800 s to finish, and the stateless service processed almost 4 million client requests, resulting in an average of 2220 operations/s. Each adaptation is either composed of a replica insertion ("+R") or removal ("-R").

Since all replicas are identical, we consider that each replica insertion/removal in the group can theoretically improve/reduce the service throughput by $1/n$, where n is the number of replicas running the service before the adaptation request.

The service group was initially composed of 2 small (S) replicas, which means a capacity of processing 1500 operations/s. Near the 100 s mark, the first adaptation was performed, a replica insertion, which decreased the service latency from 30 ms to 18 ms.

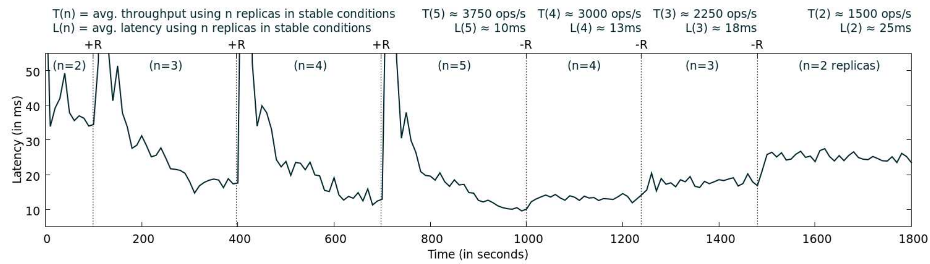


Fig. 4. Horizontal scalability test.

Other replica insertions were performed near the 400 s and 700 s marks, increasing the service group to 4, and to 5 replicas, and decreasing the service latency to 13 ms, and to 10 ms, respectively. The service achieved its peak performance with 5 replicas near the 900 s mark, and started decreasing the number of replicas with removals near the 1000 s, 1240 s and 1480 s, until when increases the service latency to 25 ms using 2 replicas.

Dynamically adapting the number of replicas can be performed reactively. A comparison between the service capacity and the current rate of client requests can determine if the service needs more or fewer replicas. In case of large differences, the system could insert or remove multiple replicas in the same adaptation request. Such rapid elasticity can adapt better to large peaks and troughs of client requests. We maintained the client request rate above the service capacity to obtain the highest number of processed operations on each second.

The entire experiment would cost $0.200 on Amazon EC2 [2] considering a static approach (using 5 small replicas), while with the dynamic approach it would cost $0.136. Thus, if this workload is repeated continuously, the dynamic approach could provide a monetary gain of 53%, which is equivalent to $1120 per year.

5.4 Scale-up and Scale-down

Vertical scalability is achieved through upgrade and downgrade procedures to adjust the service capacity to client's demands. It avoids the disadvantages of increasing the number of replicas [1], since it maintains the number of replicas after a service adaptation.

In this experiment, we scale-up and -down the replicas of our BFT key-value store, during 8000 s. Each upgrade or downgrade operation is composed of 4 replica replacements in a chain. The service group comprises 4 initially small replicas (4S mark). Fig. 5 shows the resulting latencies during the entire experiment, where each "Upgrading" and "Downgrading" mark indicates a scale-up and scale-down, respectively. We also present on top of this figure the "(4S)", "(4M)" and "(4L)" marks, indicating the quantity and type of VMs used on the entire service group between the previous and the next marks, as well as the average latency and number of operations per second that each configuration is able to process.

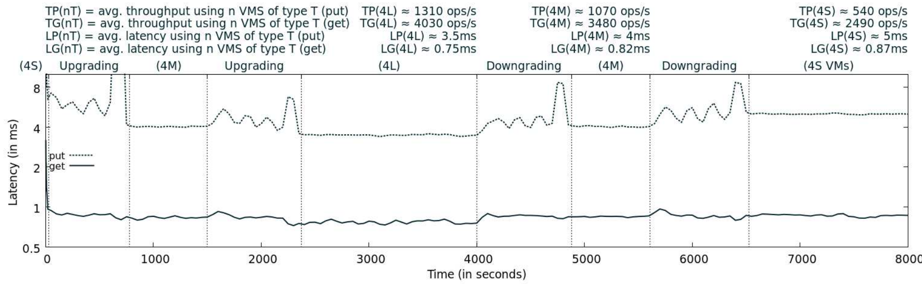


Fig. 5. Vertical scalability test. Note that y-axis is in logarithmic scale.

The first adaptation was an upgrade from small (S) to medium (M) VMs, which reduced PUT latency from near 6 ms to 4 ms and GET latency from almost 0.9 ms to 0.82 ms. The second round was an upgrade to large (L) VMs. This reduced the PUT latency from 4 ms to 3.5 ms and the GET latency from 0.82 ms to 0.75 ms. Later, we performed downgrades to the service (from large to medium and from medium to small), which reduced the performance and increased the PUT latency to almost 5 ms and the GET latency to 0.87 ms.

The entire experiment would cost $2.84 on Amazon EC2 [2] considering the static approach (using 4 large replicas), while the dynamic approach would cost $1.68. This can be translated into an economical gain of 32%, the equivalent to $4559 per year.

6 Related Work

Dynamic resource provisioning is a core functionality in cloud environments. Previous work [7] has studied the basic mechanisms to provide resources given a set of SLAs that define the user's requirements. In our work, cloud managers provide resources based on requests sent by the adaptation manager, for allocating and releasing resources.

The adaptation manager is responsible for performing dynamic adaptations in arbitrary service components. These adaptations are defined by an adaptation heuristic. There are several proposals for this kind of component [6, 14, 17], which normally follow the "monitor-analyse-plan-execute" adaptation loop [19]. Their focus is mostly on preparing decision heuristics, not on executing the dynamic adaptations. Such heuristics are based mainly on performance aspects, whereas our work is concerned with performance and dependability aspects. One of our main goals is to maintain the service trustworthiness level during the entire mission time, even in the presence of faults. As none of the aforementioned papers was concerned about economy aspects, none of them releases or migrate over-provisioned resources to save money [28].

Regarding the results presented in these works, only Rainbow [17] demonstrates the throughput of a stateless web service running over their adaptation system. However, it does not discuss the impact caused by adaptations in the service provisioning. In our evaluation, we demonstrate the impact of replacing an entire group of service replicas, in a stateless web service, and additionally, in a stateful BFT key-value store.

Regarding the execution step of dynamic adaptation, only Dynaco [6] describes the resource allocation process and executes it using grid resource managers. FITCH is prepared to allow the execution of dynamic adaptations using multiple cloud resource managers. As adaptation heuristics is not the focus of this paper, we discussed some reactive opportunities, but implemented only time-based proactive heuristics. In the same way, proactive recovery is essential in order to maintain availability of replicated systems in the presence of faults [8, 15, 22, 24, 25]. An important difference to our work is that these systems do not consider the opportunities for dynamic management and elasticity as given in cloud environments.

Dynamic reconfiguration has been considered in previous work on group communication systems [10] and reconfigurable replicated state machines [20, 21]. These approaches are orthogonal to our contribution, which is a system architecture that allows taking advantage of a dynamic cloud infrastructure for such reconfigurations.

7 Conclusions

Replicated services do not take advantage from the dynamism of cloud resource provisioning to adapt to real-world changing conditions. However, there are opportunities to improve these services in terms of performance, dependability and cost-efficiency if such cloud capabilities are used.

In this paper, we presented and evaluated FITCH, a novel infrastructure to support the dynamic adaptation of replicated services in cloud environments. Our architecture is based on well-understood architectural principles [27] and can be implemented in current data centre architectures [3] and cloud platforms with minimal changes. The three basic adaptation operations supported by FITCH – add, remove and replace a replica – were enough to perform all adaptation of interest.

We validated FITCH by implementing two representative services: a crash fault-tolerant web service and a BFT key-value store. We show that it is possible to augment the dependability of such services through proactive recovery with minimal impact on their performance. Moreover, the use of FITCH allows both services to adapt to different workloads through scale-up/down and scale-out/in techniques.

Acknowledgements. This work was partially supported by Fundação para a Ciência e Tecnologia (FCT), through the CloudFIT project (PTDC/EIA-CCO/108299/2008) and by EU's Seventh Framework Program (FP7/2007-2013), through the Tclouds project (ICT-257243). We thank the anonymous reviewers of DAIS 2013 for their helpful comments about this paper.

References

1. Abd-El-Malek, M. *et al.*: Fault-scalable Byzantine fault-tolerant services. In: Proc. of SOSP (2005)
2. Amazon Web Services: Amazon Elastic Compute Cloud (Amazon EC2) (2006), http://aws.amazon.com/ec2/
3. Barroso, L., Hölzle, U.: The datacenter as a computer: An introduction to the design of warehouse-scale machines. Synthesis Lectures on Computer Architecture 4(1) (2009)

4. Bessani, A., Correia, M., Quaresma, B., André, F., Sousa, P.: Depsky: dependable and secure storage in a cloud-of-clouds. In: Proc. of EuroSys (2011)
5. Bessani, A. *et al.*: BFT-SMaRt webpage., http://code.google.com/p/bft-smart
6. Buisson, J., Andre, F., Pazat, J.L.: Supporting adaptable applications in grid resource management systems. In: Proc. of the IEEE/ACM Int. Conf. on Grid Computing (2007)
7. Buyya, R., Garg, S., Calheiros, R.: SLA-oriented resource provisioning for cloud computing: Challenges, architecture, and solutions. In: Proc. of Cloud and Service Computing (2011)
8. Castro, M., Liskov, B.: Practical Byzantine fault tolerance and proactive recovery. ACM Trans. Comput. Syst. 20(4) (2002)
9. Chandra, T., Griesemer, R., Redstone, J.: Paxos made live - an engineering perspective. In: Proc. of the PODC (2007)
10. Chen, W., Hiltunen, M., Schlichting, R.: Constructing adaptive software in distributed systems. In: Proc. of ICDCS (2001)
11. Cooper, B. *et al.*: Benchmarking cloud serving systems with YCSB. In: Proc. of SOCC (2010)
12. Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., Warfield, A.: Remus: high availability via asynchronous virtual machine replication. In: Proc. of the NSDI'08 (2008)
13. DeCandia, G. *et al.*: Dynamo: Amazon's highly available key-value store. In: Proc. of SOSP (2007)
14. Dejun, J., Pierre, G., Chi, C.H.: Autonomous resource provisioning for multi-service web applications. In: Proc. of the WWW (2010)
15. Distler, T. *et al.*: SPARE: Replicas on Hold. In: Proc. of NDSS (2011)
16. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM 35 (1988)
17. Garlan, D. *et al.*: Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer 37(10) (2004)
18. Huang, Y., Kintala, C., Kolettis, N., Fulton, N.: Software rejuvenation: analysis, module and applications. In: Proc. of FTCS (1995)
19. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1) (2003)
20. Lamport, L., Malkhi, D., Zhou, L.: Reconfiguring a state machine. SIGACT News 41(1) (2010)
21. Lorch, J. *et al.*: The SMART way to migrate replicated stateful services. In: Proc. of EuroSys (2006)
22. Reiser, H., Kapitza, R.: Hypervisor-based efficient proactive recovery. In: Proc. of SRDS (2007)
23. Schneider, F.: Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Comput. Surv. 22(4) (1990)
24. Sousa, P., Neves, N., Verissimo, P.: How resilient are distributed f fault/intrusion-tolerant systems? In: Proc. of DSN (2005)
25. Sousa, P. *et al.*: Highly available intrusion-tolerant services with proactive-reactive recovery. IEEE Trans. on Parallel and Distributed Systems (2010)
26. Sun Microsystems: Web services performance: Comparing Java™ 2 enterprise edition (J2EE™ platform) and .NET framework. Tech. rep., Sun Microsystems, Inc. (2004)
27. Veríssimo, P.: Travelling throught wormholes: Meeting the grand challenge of distributed systems. In: Proc. of FuDiCo (2002)
28. Yi, S., Andrzejak, A., Kondo, D.: Monetary cost-aware checkpointing and migration on amazon cloud spot instances. IEEE Trans. on Services Computing PP(99) (2011)
29. Zhang, W.: Linux virtual server for scalable network services. In: Proc. of Linux (2000)