Chapter 5
Anti-fragile Cloud Solutions

To better understand how to achieve anti-fragility to downtime, the chapters of Part II discuss how to realize the four design principles and the one operational principle from Chap. 4 in different types of systems. The current chapter focuses on how to realize the principles in customer-facing web-scale solutions in the cloud. Much of the discussion is based on design and operational patterns described by Nygard [35] and Netflix's realization of these patterns in its cloud-based streaming service. YouTube videos (http://youtube.com/watch?v=jCanhyFDopQ, https://youtube.com/watch?v=dekV3Oq7pH8) document that the development teams at Netflix used the principles described in Chap. 4 to build and operate an anti-fragile system.

5.1 Choice of System Realization

We initially consider the advantage of realizing a web-scale solution in a public cloud compared to a traditional private datacenter. For simplicity, we consider a generic cloud infrastructure offering services to manage virtual machines, data storage, networking, and monitoring. The generic cloud platform is divided into regions, each with multiple availability zones. The zones correspond to different datacenters. All virtual machines run on commodity hardware. Failures happen routinely due to the infrastructure's huge number of servers, storage units, and network equipment [13, 14]. While a company or a government can build a private cloud infrastructure, it is less expensive to use a public cloud, because its cost is divided among many customers. The pay-as-you-go pricing model makes a public cloud especially attractive to startup companies that want to compete with established companies having their own infrastructures.

The architecture of a system models the major components and the important relations between them [52]. Figure 5.1a sketches the architecture of a web-scale solution running in a private datacenter without cloud technologies. This server-side application is said to be *monolithic* because it is built as one entity with a single

K.J. Hole, *Anti-fragile ICT Systems*, Simula SpringerBriefs on Computing 1,
DOI 10.1007/978-3-319-30070-2_5

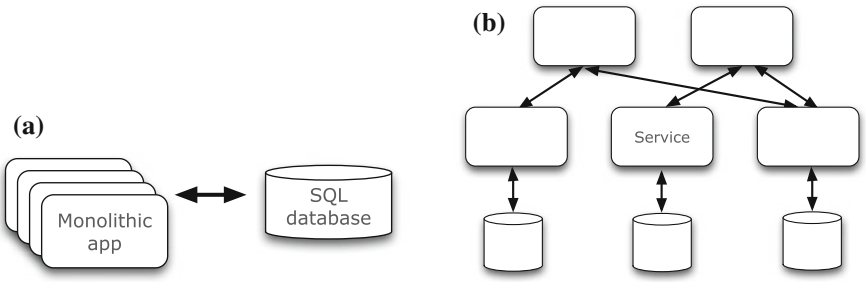

Fig. 5.1 **a** Replicated monolithic application running on multiple servers in a private datacenter. All application copies use the same database. **b** Identical application functionality realized by self-contained services running on a cloud platform and storing data in databases replicated over multiple zones

executable [53, 54]. One or more load balancers (not shown) distribute requests to replicated executables that run on multiple servers. All application copies use the same database. Figure 5.1b sketches a service-oriented architecture (SOA) with layers of self-contained services running on a cloud platform. Together, the services provide the same functionality as the monolithic application. The services use individual databases replicated in multiple zones. Each service is scaled individually by running multiple copies.

The *availability* of a web-scale solution is measured by the percentage of time it is accessible to users. A high availability of 99.99 %, referred to as four nines of availability, corresponds to about 53 min of downtime each year. A solution's *scalability* refers to the number of concurrent users who are having a positive experience and its *performance* refers to the experience of individual users, often measured by response time (latency) [14]. A customer-facing web-scale solution must have high availability to avoid customer dissatisfaction, high scalability to support tens of millions of customers, and good performance to quickly respond to the real-time requests of each customer.

Monolithic (non-cloud) solutions with multiple load balancers and many servers have good scalability up to a point, beyond which scalability becomes exceedingly difficult. The strongly connected modules in the software layer and the high integration of subsystems in the hardware layer also ensure low-latency communication. However, tight integration in both layers leads to propagating failures resulting in insufficient availability. As we shall see, SOA in the cloud provides an efficient way to leverage the redundancy and diversity needed to break the strong dependencies in monolithic solutions. Furthermore, server virtualization on a massive hardware platform supports almost unlimited (horizontal) scalability and the use of multiple cloud regions facilitates low-latency service throughout the world.

5.2 Modularity via Microservices

To achieve anti-fragility to downtime, it is not enough to move a monolithic solution into a cloud. It is vital to build a cloud-native solution that takes full advantage of the cloud's properties [13, 14]. In particular, the choice of application architecture is vital to achieve a high degree of anti-fragility to downtime. SOA introduces modularity in the form of well-defined and self-contained services in the software layer. In the hardware layer, the cloud supports modularity by having many availability zones and by assigning multiple zones to each region, where the regions cover different parts of the world.

Netflix's streaming application is based on a fine-grained SOA with *microservices* running in the Amazon Web Services (AWS) cloud. Each of the microservices focuses on doing one thing well. They are combined to provide the needed functionality. In early 2014, the Netflix solution had roughly 600 microservices running side by side in each cloud region. The services are responsible for handling customer-facing requests via a few edge services. The large geographical spread of AWS's regions enables Netflix to offer low-latency, high-throughput media streaming in many countries.

While there is no generally agreed upon definition of microservices, it is possible to describe common properties [53, 54]. A microservice encapsulates a well-defined functionality of value in a business context. The functionality fulfills a single purpose. A microservice runs as a separate process with fast startup and shutdown times. Services can be tested, upgraded, and replaced independently of each other. Finally, a microservice manages its own data. Together, microservices separate the functionality of a large application into highly independent chunks of code. They communicate via a standardized set of simple protocols. The services can be written in different languages and utilize different storage technologies. As we shall see, microservices enhance fault tolerance, enable an application to scale, and allow a solution to evolve.

5.3 Weak Links via Circuit Breakers

Virtual machines running (micro-) services are modules in the cloud's software layer. To stop and start the virtual machines without significantly degrading the user experience, they need to be autonomous and stateless. Application state must be stored externally to the machines. If the application state is distributed over many storage devices, then it is possible to upgrade these hardware devices without halting the application.

Weak links are implemented using the *circuit breaker* pattern to ensure that the services are weakly connected [35]. No service contacts another service directly; instead, a service is called via a circuit breaker. The circuit breaker quickly detects when a service develops a problem and open s the circuit (breaks the weak link) to stop the problem from propagating to other services and to provide calling services with a default fallback response. The circuit closes after the problem is fixed. Because

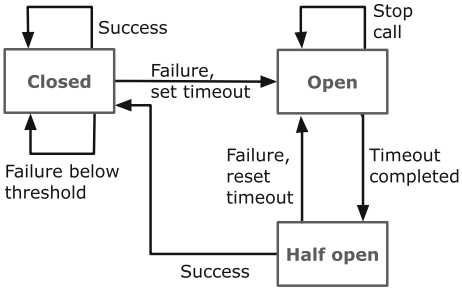

Fig. 5.2 State diagram for a generic circuit breaker

the circuit breaker fails fast, it controls the failure mode, facilitating the graceful degradation of a system's functionality to limit the damage to users. Note that the circuit breaker prevents any positive feedback loop from escalating a local failure into a systemic failure.

Figure 5.2 shows a state diagram of a generic circuit breaker. In the normal closed state, the circuit breaker is closed and a calling service is allowed to connect to the called service. If there is a failure, the circuit breaker records it. Once the number, or frequency, of failures reaches a certain threshold, the circuit trips and opens the circuit. When the circuit is open, all calls are stopped. After a certain time in the open state, the circuit breaker moves to the half-open state, which allows the next calling service to connect to the called service. If this trial call succeeds, then the circuit breaker returns to the normal closed state. However, should the trial fail, the circuit breaker returns to the open state until another timeout elapses. To learn more about circuit breakers, the interested reader should study Netflix's open source implementation of the pattern (http://github.com/Netflix/Hystrix/wiki).

5.4 Redundancy Provided by the Cloud

The cloud supports redundancy at the virtual machine, zone, and region layers. At the virtual machine layer, a web-scale solution runs redundant machines with timeout and failover, as illustrated in Fig. 5.3. Multiple virtual machines run the same (micro-) service in a single availability zone. Figure 5.3a depicts a service depending on one of the redundant services. When the contacted redundant service times out, another is queried. The arrows show the direction of the dependencies. Failure of an instance is often due to power outage in the hosting rack, a disk failure, or a network partition that cuts off access. When there is a software bug or network failure, all instances are affected and a (non-personalized) default response is necessary to contain the error. Careful analysis is needed to determine the appropriate response. Figure 5.3b illustrates the timeout with a default fallback response.

At the zone layer, failure in one zone should not affect the operation of other zones. Multiple zones in a single region provide redundancy, as shown in Fig. 5.4. The

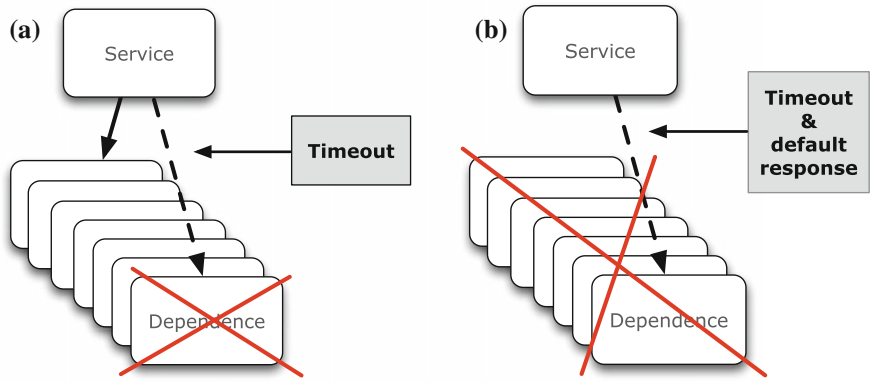

Fig. 5.3 **a** When a service instance times out, another instance is queried. **b** If all instances fail, then there is a default response. The *arrows* show the direction of dependency

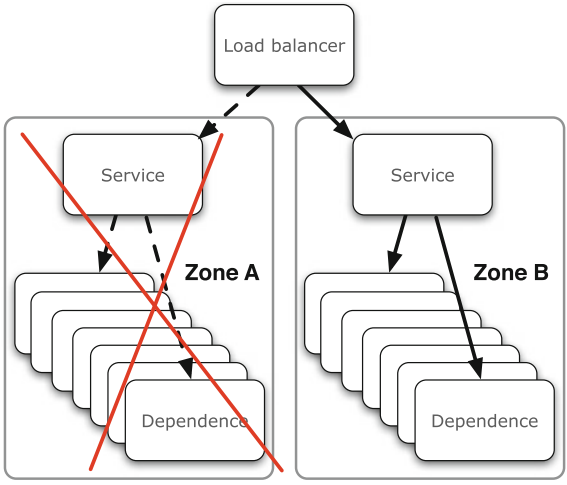

Fig. 5.4 Use of multiple zones to isolate a zone failure

redundancy isolates the consequences of firmware failures, certain serious software bugs, power failures, and severe network failures that take down a whole zone. A web-scale solution should use multiple zones in each region. To compensate for a failed zone, the solution could scale up the remaining zones or introduce a new zone. Note that the load balancer is a single point of failure in Fig. 5.4.

A whole region could fail due to configuration issues, bugs in the infrastructure code, bugs in the application code, and failures in the load balancer. A failed region should not affect other regions. Figure 5.5 depicts two regions, where a server for the domain name system (DNS) splits the traffic load in two halves. A solution should switch users to a new region when needed.

While the redundancy of executable code is important, the data replication obtained by storing the same data on multiple storage devices is critical to achieve

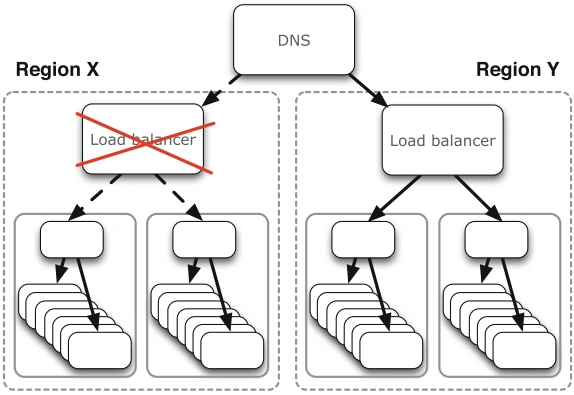

Fig. 5.5 Use of multiple regions to isolate a region failure

high availability. A cloud infrastructure with a highly redundant network and data storage provides both high availability and extremely high durability, that is, only a tiny probability of data loss. Netflix goes to great lengths to ensure the availability and durability of its data. First, Netflix uses an Apache Cassandra database (http://cassandra.apache.org) that stores data in three zones per region. Cassandra provides NoSQL persistent data storage with eventual consistency [55]. It also supports asynchronous cross-region replication.

Furthermore, Netflix stores backups in the Amazon Simple Storage Service (S3), which is designed to provide 99.99 % availability and 99.999999999 % durability of data objects over a given year, though there is no service-level agreement for durability. The S3 service redundantly stores data in multiple facilities and on multiple devices within each facility. An application is first informed about successful storage after the data are stored across all facilities (http://aws.amazon.com/s3/details). Finally, Netflix copies data in S3 to a storage service run by another cloud provider.

5.5 Diversity Enabled by the Cloud

Two software programs are diverse if they have (nearly) the same functionality but different designs or implementations, that is, different machine code [24, 56]. We utilize software diversity to isolate failures by switching between diverse codes, especially when introducing updated services.

Since a web-scale solution supports users throughout the world, there is no good time to take down the whole system to upgrade its software. An alternative is to introduce new code by keeping both old and new code running and switch user requests to the new code. An early version of an updated service is called a *canary*, referring to a canary in a coal mine. The stability of a canary cannot be fully evaluated before it is exposed to a heavy traffic load in a production system. Figure 5.6 illustrates a simple

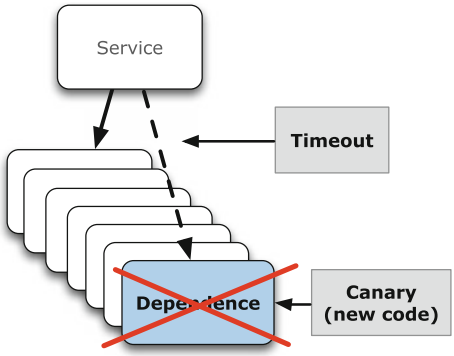

Fig. 5.6 A simple canary push

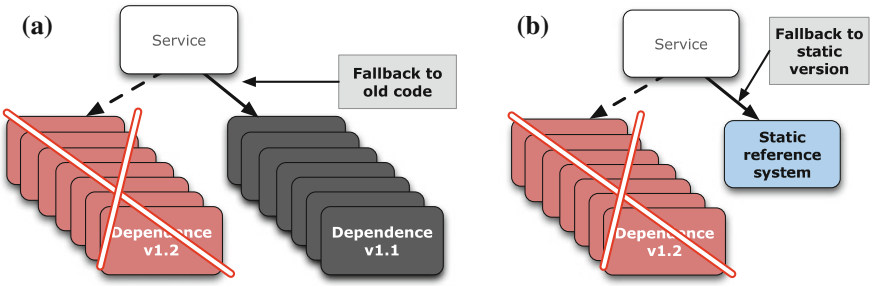

Fig. 5.7 **a** *Red–black* push. **b** Fallback to the static version

canary push where a single instance of an updated service is put into production. If a failure occurs, the system switches back to the old service.

It is possible to extend a simple canary push to include many instances of an updated service. Figure 5.7a illustrates a so-called red–black push where multiple instances of an updated service are needed to carry the traffic load. Instances of the old service are also running to ensure that the system handles peak load if there is a problem with the new code. Note that the cloud facilitates this process because it is easy to double the use of resources for a limited period, for example, a 24-h cycle.

Several versions of a service may contain a "time bomb" that only goes off after a long period. There could be a software bug in both the red and black deployments in Fig. 5.7a or there could be a problem with the data causing several versions of the code to fail. As shown in Fig. 5.7b, it is possible to independently author a static service with simple functionality that delivers a minimal solution when all recent versions of the code fail.

5.6 Fail Fast Using Software Tools

To protect and extend companies' market share, web-scale solutions must support rapid scaling and innovation. Since the rate, or frequency, of hardware failures increases as more hardware is added and the rate of software failures increases as the rate of change grows, frequent hardware and software failures are inevitable in web-scale solutions. The engineers at Netflix do not wait for failures to happen. Instead they use a collection of tools, called the *Simian Army*, to deliberately introduce failures into their production system to quickly learn about vulnerabilities and then make changes to ensure that the vulnerabilities do not cause systemic failures (http://techblog.netflix.com/2011/07/netflix-simian-army.html).

The Chaos Monkey tool disables randomly selected virtual machines to make sure the Netflix solution survives this common type of failure without any customer impact. Latency Monkey introduces random latencies between services to simulate network degradation and to ensure that services tolerate latency spikes and other networking issues. The shutdown of a low-level dependency can lead to a longer timeout at a higher layer, causing a cascading failure. Because there is no general answer to this multi-level dependency problem, each case must be carefully studied. Chaos Gorilla generates zone failures and Chaos Kong generates region failures to test that the system survives such rare incidents with a huge impact.

Netflix's preferred approach to failure detection supports Taleb's [9] well-founded claim that it is impossible to predict all rare incidents with a huge negative impact in complex adaptive systems. Instead of trying to predict gray swans, Netflix simply tests its system on a continuous basis to maintain isolation of local failures as the system changes, especially to avoid propagating failures causing downtime. Netflix engineers run the Simian Army tools during the business day to learn about vulnerabilities in the system and to address any immediate problems. If appropriate, the engineers build an automatic recovery mechanism to deal with a newly discovered vulnerability, so that next time a failure occurs no user will notice.

Since silent failures inhibit learning, failures must be detected to prevent a system from becoming increasingly fragile over time. It is necessary to monitor the system's behavior, especially behavioral changes due to system updates. Netflix has built a telemetry system that monitors many different aspects of the system behavior. As an example, a tool using telemetry data determines whether a canary is doing well. There also exist monkeys to monitor the system, such as Security Monkey (http://techblog.netflix.com/2014/06/announcing-security-monkey-aws-security.html) and Conformity Monkey (http://techblog.netflix.com/2013/05/conformity-monkey-keeping-your-cloud.html). This extensive monitoring allows Netflix to constantly adjust its system to keep within the bounds of normal operation.

5.7 Top–Down Design and Bottom–Up Tinkering

Before leaving Netflix, we consider the collective impact of the five principles. The four design principles essentially prescribe a *top–down* (reductionist) design approach breaking a new system down into modules and then adding weak links between the modules, thus isolating the impact of local failures. Finally, redundancy and diversity are added to further limit the impact of local failures.

A company $\mathcal{C}$ with a system of weakly connected modules that are constantly monitored and tested according to the fail fast operational principle has a competitive advantage over a company $\mathcal{D}$ with a system of strongly connected modules. Company $\mathcal{C}$ can tinker with its system in a *bottom–up* manner without causing system downtime due to the propagation of local failures. At Netflix, many engineering teams constantly innovate their services without any central coordination of new releases. The teams introduce new features and product enhancements rapidly and frequently. It is hard for company $\mathcal{D}$ to maintain a similar high rate of innovation because the many strong dependencies between the modules in their system require the preparation of large coordinated software releases.

Together, the five principles facilitate local decision making in highly independent developer teams. These teams need not schedule common software releases as long as they inform all affected teams about any increased use of computational resources and changes to programming interfaces.

5.8 Discussion and Summary

Netflix's web-scale implementation of its media streaming solution with anti-fragility to downtime is evidence that the cloud facilitates the implementation of the design principles of modularity, weak links, redundancy, and diversity and the fail fast operational principle presented in Chap. 4. The generality of the cloud-based realization presented indicates that other large solutions can benefit from Netflix's approach. The next chapters present more systems for which the cloud simplifies the creation of anti-fragility to downtime. Part III shows that the principles also create anti-fragility to malware spreading.

Although the cloud represents a golden opportunity to develop and operate anti-fragile systems, it is not a panacea. A highly competent cloud provider must be selected, preferably with cloud regions throughout the world. Since a major security breach is unacceptable, it is particularly important that the cloud platform limit the consequences of attacks. Finally, any anti-fragile application must be able to handle a situation in which all datacenters in a region go down at the same time.

It is interesting to observe how the design and operational principles together enable safe bottom–up tinkering without the central coordination of different development teams. This advantage can accelerate innovation compared to more

traditional software development methods requiring large coordinated software releases.