High Availability for Carrier-Grade SIP Infrastructure on Cloud Platforms

Jong Yul Kim and Henning Schulzrinne
Department of Computer Science
Columbia University
New York, NY 10027
{jyk,hgs}@cs.columbia.edu

March 19, 2012

Abstract

SIP infrastructure on cloud platforms has the potential to be both scalable and highly available. In our previous project, we focused on the scalability aspect of SIP services on cloud platforms; the focus of this project is on the high availability aspect. We investigated the effects of component fault on service availability with the goal of understanding how high availability can be guaranteed even in the face of component faults. The experiments were conducted empirically on a real system that runs on Amazon EC2. Our analysis shows that most component faults are masked with a simple automatic failover technique. However, we have also identified fundamental problems that cannot be addressed by simple failover techniques; a problem involving DNS cache in resolvers and a problem involving static failover configurations. Recommendations on how to solve these problems are included in the report.

Contents

Chapter 1

Introduction

Real-time communications systems - systems that provide interactive multimedia communications among its users - need to be resilient to failures. However, these are complex systems built with multiple components, so knowing how to implement a highly available service on top of cloud platforms is not straightforward. In this project, we studied the consequence of component failures on the function of the overall system; on whether failure of components result in a failure of the overall system to process new incoming calls. Then, we considered how to build a highly available cloud-based system in which new calls are accepted even with component faults.

Our approach to failure analysis is macroscopic: the focus is on how component failures affect the entire system and its functionality to accept new calls, not on how individual components within the system are made more resilient. Individual components may fail for many reasons, e.g. due to hardware, software, or virtual machine faults (see Chapter 2), but our main interest is not on how they fail nor on how to make the individual components more resilient. Our goal is to understand what happens to the entire system *when* a component fails so that we can make the system function even in the face of individual component faults.

Our analysis deals with single component faults. The result of our analysis, in Chapter 4, is the illustration of the chain of events that occurs, starting from a component fault and leading towards success or failure of the system to process new incoming calls. Once such a chain of events has been identified, we can consider ways to block it or mask the fault so that it does not lead to the service failure.

In our experiments, we terminate a single component within the system to investigate the details of how it affects other components in the system. The component that fails, i.e., the root cause of system failure, fails completely since we're shutting down the process. In other words, our experiment is with faulty components that show fail-stop behavior, i.e., when a component fails it stops completely. There are no assumptions about surviving components. They may be partially operating, leading to partial system failure. Investigation of how partial failure in the a faulty component can affect the overall system is left as future work.

The failure experiments are conducted on a prototype system that is representative of a class of real-time communications systems that use the Session Initiation Protocol (SIP) for signaling and the Real Time Protocol (RTP) for media delivery. Our system is built with components and architecture commonly found in SIP/RTP systems. By analyzing such a representative system, our aim is to make our results generally applicable to a whole class of systems instead of being applicable only to our specific implementation. Both the methodology of our experiments and the representative system is explained further in Chapter 3.

Lastly, based on the results from our analysis, we make recommendations on how to make real-time communications service more resilient to component failures in Chapter 5.

Our contributions in this project are the following:

1. As far as we're aware, this is the first attempt to analyze the effects of VoIP component faults on SIP services that run on cloud platforms.
2. The fault analysis exposed some interesting problems that, as far as we're aware, has not been solved for VoIP services on cloud platforms.
3. By actually implementing a real system on a cloud platform, we are able to share our experiences about building such a system.

Chapter 2

Why do individual components fail?

As noted in Chapter 1, finding the reasons that components fail is not the main focus of the project. However, it is beneficial to find out about why they do fail since this informs and enriches the results that we will get from our experiments. This chapter is dedicated to understanding why components fail.

2.1 Terminology

In this report, we use the terminology of fault, error, and failure as defined in Avizienis et al. [9]. *Failure* is “an event that occurs when the delivered service deviates from correct service”. An *error* is defined as a deviation from correct state, that is, an incorrect state of the system. A *fault* is the cause of an error. In short, a fault causes a system to go into error and the error may lead to system failure.

This chapter is about individual components. Therefore, the term “failure” in this context does not mean the failure of an entire real-time communications system but a failure of an individual component. Likewise, “error” is a deviant state within an individual component and “fault” is the cause of such error.

In the rest of the report, these terms are used in a different level of abstraction where “failure” is the failure of an entire real-time communications system, “error” is a deviant state of the entire system, and “fault” is the cause - a component failure - of a system error.

2.2 Faults that lead to component failure

According to Gray [11], potential cause of system failure can be categorized into hardware, software, environment, and administration faults. The categories of faults described in Avizienis et al. [9] are more elaborate but they can be reduced to the four categories used by Gray. We follow Gray's category describe component faults.

Under each fault category, we explain

- why it happens,
- how it affects the component,
- how often it happens, and
- how to prevent the fault or mask its effects.

2.2.1 Hardware faults

Hardware faults arise from malfunctioning or non-functioning CPU, memory, disk drive, network card, motherboard, or power supply. These, in turn, could be caused by bad manufacturing (infant mortality), physical degradation (end-of-life mortality), or abuse such as overclocking a CPU [15]. In some cases, natural phenomena such as cosmic rays or solar flares can interfere with the proper operation of hardware [9].

Hardware problems may be diagnosed at the early booting stage with Power On Self Tests (POTS). The faults that are not detected could lead to unexplainable software behavior or operating system crashes.

Gray [11] reports that, after subtracting infant mortality, hardware faults are a minor contributor to total system outages[1]. The reason is that machines are made fault-tolerant by redundancy, examples being RAID systems and dual power adapters, so that a hardware fault does not immediately lead to component failure.

Virtualization may further increase a component's hardware fault-tolerance. Patnaik et al. [17] shows how to use virtual machine migration in an active-passive configuration to recover from faults. The technique is promising as a way to recover from hardware faults. When the hardware that hosts the active virtual machine fails, the passive virtual machine on a different hardware takes over. Extensive checkpointing is performed during runtime so that buffers and memory state at the operating system level, including the state and buffer of the network stack if needed, are identical between active and passive virtual machines. Patnaik et al. [17] used this technique on SIP/RTP components and reports complete recovery for signaling but mixed results for media streams due to packet delays and losses during continuous checkpointing.

2.2.2 Software faults

Software faults occur because of bugs that were not fixed while developing the software or by bugs newly introduced in an upgrade. Bugs have a wide range

[1]Gray's study was done in 1985 for a line of fault-tolerant Tandem computer systems. The relative frequency of faults still seems to be valid, as shown in a 2003 paper by Oppenheimer et al. [16], in which the authors investigated why internet services fail. Although these are two different systems, analyzed at different levels of abstraction, the relative frequency of faults that lead to system failure were similar.

of effects on software such as hanging or crashing the software, degrading performance, or corrupting data.

In a study of bugs in two popular open source software [12], bugs were categorized, based on root cause, into memory bugs, concurrency bugs, or semantic bugs. Memory bugs are in code that mishandle memory objects, such as a double free or referencing a null pointer. Concurrency bugs appear in multi-threaded or multi-processor environment where the indeterminancy of the order and the atomicity of operation can lead to deadlocks and race conditions. Semantics bugs refer to application-level error in code, such as an incorrect implementation of a specification or an unresponsive button. Neglecting an exception, such as a full hard disk, is also considered a semantic bug.

Gray [11] categorizes bugs as Bohrbugs and Heisenbugs, based on the relative frequency of bug manifestation. Bohrbugs are easily detected because they manifest with regular frequency. In a well-tested software, most Bohrbugs are fixed before the software is shipped. On the other hand, Heisenbugs are harder to detect because they do not occur regularly and may disappear when the software environment changes or the software is restarted. It takes time to fix Heisenbugs. However, restarting the software is a good way to get rid of Heisenbug manifestation.

There are lots of literature, practices, and tools on how to reduce bugs in software. Code organization principles, bug detection at compile time and runtime, software testing, garbage collection are some of the examples of techniques that can help programmers reduce bugs in software, although none by itself is effective.

In Gray's report [11], software faults are a major contributor to system outages in Tandem computers, second to administration faults. Furthermore, writing concurrent programs that take advantage of multicore hardware is difficult, and even patches for concurrency bugs can be buggy due to the difficulty in reasoning with non-determinism [13]. With increasing software complexity, it seems that software faults will continue to be a major contributor to component failures.

2.2.3 Environment faults

Environment faults encompass power failures and network failures. These failures may occur because of problems at the power company, in case of a power failure, or ISP, in case of a network failure. Natural disasters such as earthquakes and tornadoes, and man-made disasters such as fire are a few of the reasons that environment faults can occur.

Redundancy is a core principle in dealing with environment faults. Having backup power sources, such as Uninterruptable Power Supply (UPS), and multiple upstream network providers is common practice in data centers. Also, geographic redundancy of servers helps to continue service despite local natural and man-made disasters.

Environment faults are the least contributor to system outages [11], but when it happens, it affects a large proportion of components in the local area.

2.2.4 Administration faults

Most administration faults are from normal maintenance such as upgrading the hardware or software, performing preventive maintenance procedures, adding or removing a component, or configuring a component [11] [16]. It is known that making changes to a component is risky, even if the human operator is competent, when new hardware or software has to be installed [11]. Also, configuration is often difficult the more interconnected and complex the component is. Furthermore, human operators occasionally make mistakes, like typing a wrong command or unplugging the wrong module.

According to Gray [11], administration faults are the largest source of failures. The reason for this, according to Oppenheimer et al. [16], is that administration faults are harder to mask than software or hardware faults. Redundancy does not mask administration faults as well as hardware faults and to a lesser extent, software faults. For example, a mis-configured component may not work properly, even though the hardware and software within the component are fault-tolerant.

Gray [11] suggests that making self-configured systems with minimal maintenance and a simple and consistent operator interface is essential to reduce administrative faults. Installing tested hardware and software and then avoiding unnecessary changes may also be a good strategy.

2.3 Relevance to our experiments

As mentioned in Chapter 1, root-cause components are shutdown completely in our experiments. What kind of faults makes the component fail completely, like the shutdown we're introducing artificially? An obvious example is referencing a null pointer, at which the process is forced to terminate. At this point, the component is no longer functioning.

Sometimes the distinction is subtle. For example, when there is a network card fault, the component is inaccessible from other entities but other parts of the component are still functioning correctly. However, in the viewpoint of other entities, this component is no longer functioning. Therefore, this is also a case of a complete failure.

Other examples of faults that lead to complete component failure are: misconfigurations that disable the component's main function or communication with other entities, power shutdown, and deadlocks.

In our experiments, the root-cause component could have failed from any of the faults above. We do not make the distinction since our focus is on what happens to state of the entire system after a component fails; whether the error is visible externally, how many other components are affected, and what are the ways to recover the system.

Chapter 3

Methodology

3.1 Overview

As mentioned earlier, the goal of the project is to expose the details of how a single component faults affect the call processing functionality of real-time communications system. In this chapter, we delve into what class of system is under investigation, and how we designed and conducted the experiment. Results from these experiments are explained in the next chapter.

Section 3.2 describes the system under investigation. The section starts with an explanation of the real-time communications system from the highest level of abstraction (the whole system as a blackbox), continues towards the middle level of abstraction (architecture and components), and ends with implementation details of our specific prototype. The adoption of a commonly found architecture and the use of open-source products is intentional: the analysis of failures in such a system would be beneficial to a wider group of designers and practitioners. Although we're forced to choose a specific implementation of the system, the results of our experiments are intended to be generally applicable to SIP/RTP systems, although the focus is on systems that use cloud platforms.

The next section describes the method and the end product of investigation. The end product of the investigation is an explanation of a chain of events, triggered by a component fault, and how it ultimately affects the whole system. Three main points of interest are: which of the surviving components are affected, how they are affected, and whether the fault leads to an external system failure. The chain of events are constructed by following these steps:

1. A single fault is triggered manually by terminating a running process of the selected component.
2. The state of the surviving components and the state at the system boundary are captured and observed for a period of time.
3. Any change in state are noted and used to construct the event chain.

Each of these steps are explained in more detail later.

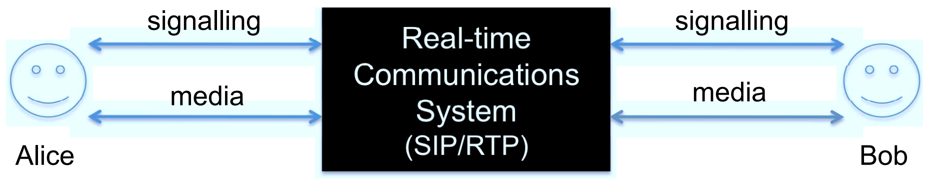

Figure 3.1: The system at the highest level of abstraction - as a blackbox.

3.2 The system under investigation

3.2.1 System as a blackbox

At the highest level of abstraction, the entire real-time communications system is a black box that interacts with users, as shown in Figure 3.1. This view of the system allows us to define the functions of the system and its behavior during failure.

Functions

This class of system handles both signaling and media delivery for users. On the signaling side, functions provided are:

1. registration of users,
2. session establishment between users, and
3. re-negotiation of an established session.

On the media delivery side, the functions provided are:

1. media delivery for users behind Network Address Translation (NAT) devices,
2. conferencing, and
3. lawful intercept of media.

More advanced functions, such as call transfer or music-on-hold, may be provided by the system. However, in this project, we chose to investigate how component faults affect the most basic functions such as the ones mentioned above.

System behavior during failure

At this level of abstraction, system failure results in failure to process new incoming calls. Disregarding communication faults between the users and the system, the following behavior shows that the system is in an error state:

- unresponsiveness, where the user's request times out
- returning an error response, such as 5xx responses or a "408 Request Timeout" response
- no media flow from the system

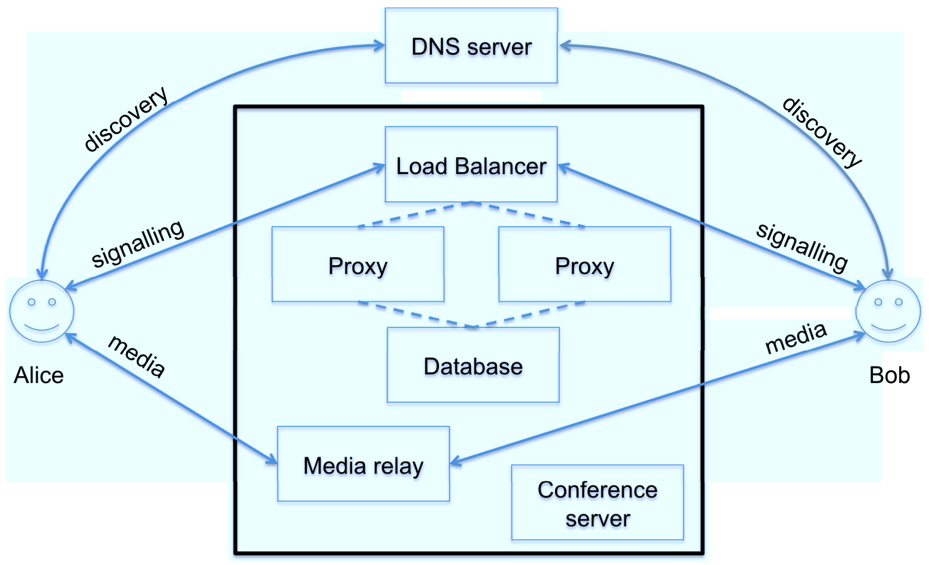

Figure 3.2: The system under investigation has a common architecture found in many real-time communication systems.

As specified in RFC 3261 [18], any 5xx response is an indicator of system failure.

"408 Timeout" generated by the system is also a symptom of system failure. In our experiments, the client is always available to receive calls and there are no communication faults between the client and the system. Therefore, a Timeout response is a indication of error inside the service provider's system.

The system handles media delivery as well; therefore, a lack of media packets from the system is an indicator of system failure.

During the experiment, responses and media packets are recorded and used as an indicator of overall system failure.

3.2.2 System architecture and components

As shown in Figure 3.2, the architecture follows a design found in many real-time communication systems, with components such as the DNS server, load balancer, proxy, database, media relay, and conference server. This type of architecture is common because it allows the service to scale using inexpensive commodity hardware and software.

The DNS server directs user requests to one of the load balancers, which statelessly forwards requests to the proxy. The proxy provides the main signaling functions of the system. As mentioned in the previous section, the functions are registration, session establishment, and re-negotiation of sessions. The database stores user information, such as the contact information, necessary for the proxy to operate correctly.

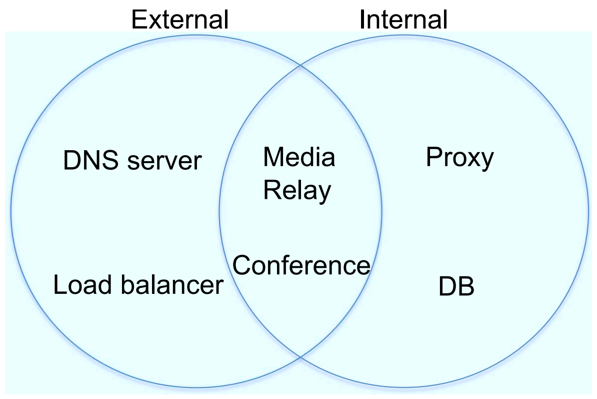

Figure 3.3: Components classified by system boundary. External components are ones that are in direct contact with clients. Media relay and conference servers are exposed to clients only as the session is being initiated.

The media relay provides media traversal for clients behind Network Address Translation (NAT) devices and is also used for lawful intercept. For all sessions established, media goes through the relay regardless of the activation of lawful intercept or whether the client is behind a NAT device. Doing so simplifies session routing logic and may be useful for future value-added features, but it comes at the cost of delays to end-to-end media packets and the use of more bandwidth.

Conference servers provide conferencing feature for sessions of three parties or more.

System boundary

As Figure 3.3 shows, the load balancer and the DNS server are the two components form the external system boundary where clients are in direct contact. As such, a fault in the external system boundary can directly affect the client.

Proxies and database servers are never exposed to the client. However, a fault in these components may propagate to the external boundary. A resilient system blocks or masks fault propagation.

Media relays and conference servers are not exposed to the client initially, but are exposed during session negotiation. The status of the four components within the external boundary circle determines whether the system is responsive to client requests.

Component states

The state of each component, relevant to the failure experiment, is the following:

Component	What constitutes its state
DNS server	DNS records of load balancers
Load balancer	The dispatch(proxy) list and the client connection list.
Proxy	Transaction success or failure? State can be observed by response type or lack of response.
Database	Up-or-down status only since we do not add or delete users during our experiments.
Media relay	Current RTP clients
Conference server	Current conference calls
Client	Session success or failure? State can be observed by DNS queries and SIP requests generated by the client.

System state is an aggregate of each state of the components within the system. A system error state is a system state where the state of each component is not congruent with the rest of the components in the system. For example, if a load balancer's dispatch list contains a dead proxy, its state is not congruent with the (dead) state of the proxy. Therefore, it is an error state which leads to service failure. The detail augments the service failure indicator, such as 5xx responses, for possible reasons of system failure.

Dependency between components

Figure 3.4 shows the dependency between components in the system. The dependency graph is a natural consequence of architecture design. The arrows show the direction in which one component affects another. For example, the operational status of database components affect the proxy, or, the proxy is dependent on the status of database components.

This graph is important when reconstructing the chain-of-events that occur after the initial fault(s). For example, a proxy fault can only directly affect the load balancer, so any state change in the rest of the components must be a side effect of a proxy fault.

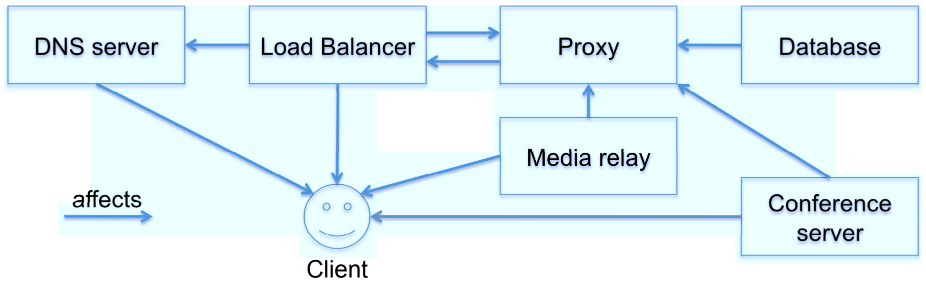

Figure 3.4: Dependency between components.

3.2.3 Implementation of components

The system was implemented using open-source products such as the SIP Express Router (SER) [8], Cassandra [3], RTPproxy [7], Asterisk [4], and PowerDNS [6]. The following table shows which open-source products were used for each component.

Component	Open-source implementation	Version
DNS server	PowerDNS	2.9.22
Load balancer	SIP Express Router	3.2.0
Proxy	SIP Express Router[1]	3.2.0
Database	Cassandra	0.7.0
Media relay	RTPproxy	1.2.1
Conference server	Asterisk	1.6.0.5

The eccentricities of our system lies in the fact that it uses a key-value store database, Cassandra [3], instead of a relational database like MySQL [5]. However, this fact has little effect on the failure analysis since, as shown in the component dependency graph in Figure 3.4, the database is not dependent on any other component. From the proxy component's point of view, there is no difference in key-value store or relational database to its operation as long as the data it needs is served correctly.

Configuration for scalability and redundancy

The components are configured to dynamically adapt to changes in the system. For example, when a new load balancer is added to the system, a new DNS record is generated and added to the DNS server. This allows clients to connect to the new load balancer when it's ready to serve requests.

Components are also configured for redundancy. There are at least two of each components. However, this does not necessarily mean that the components are acting as a pair-wise backup buddy. The surviving component will accept new transactions but old transactions on the faulty component are lost.

Operation on a cloud platform

The system operates on the Amazon Elastic Compute Cloud (EC2) platform [2]. Amazon EC2 allows operators to initiate or terminate components through their web interface and through their APIs. The only component outside Amazon EC2 is the DNS server, which runs on a hardware housed within Columbia University.

3.3 Method of investigation

The following steps are taken for each experiment.

[1]The SIP Express Router was modified to work with Cassandra as its database backend.

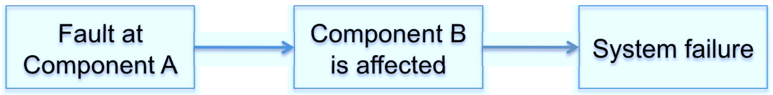

Figure 3.5: Chain of events triggered by a fault at Component A.

1. System preparation
2. Inducing component fault
3. Recording and gathering the state of components
4. Analysis and construction of the chain-of-events

3.3.1 System preparation

The system is initiated with 1 DNS server, 2 load balancers, 2 proxies, 2 database servers, 2 media relays, and 2 conference servers. Each component runs on a separate virtual machine instance in Amazon EC2.

3.3.2 Inducing component fault

Experimentation begins with a manually induced component fault. The component under test is terminated manually by killing the main process. There are six components in the system; therefore, there are six experiments for the single component fault.

3.3.3 Recording and gathering the state of components

The relevant states of each component are explained in Section 3.2.2. The state information are recorded, either as a log file or a packet trace.

3.3.4 Analysis and the construction of an event chain

Figure 3.5 shows an abstract example of an event chain.

This kind of cause-and-effect chains is the result of analyzing the state changes in the components.

Chapter 4

Single Component Faults and Its Effects

How does a single component fault affect service availability? Single component faults may be masked to a certain extent by running redundant components in the system. However, redundancy of components does not guarantee service availability. With simple automatic failover mechanisms, redundant components can be better utilized to mask component failures. In this chapter, a series of analysis on how a single component fault can lead to service failure, with and without automatic failover mechanisms, are presented in Sections 4.1 and 4.2 respectively. Analysis results are shown in cause-and-effect graphs, starting with a component fault and ending in failure or success of processing new calls.

There are some issues that cannot be solved with simple failover mechanisms employed in our experiments. These are explained in Section 4.3. Discussions about the limitations of our analysis and whether the results are specific to our system or are general to classes of systems are also included in that section.

4.1 Effects of a single component fault without automatic failover

The effects of each type of component fault on service availability vary significantly. In the set of results described in this section, the components are redundant and so there is one component that survives the faulty component but there is no automatic failover mechanism. These results serve as a comparative base case for subsequent analysis of component faults with automatic failover.

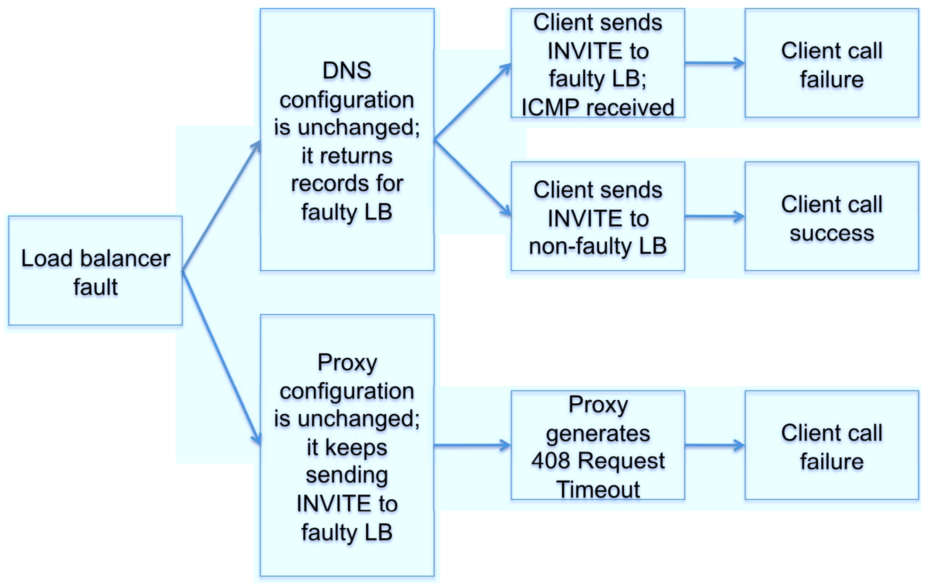
Load balancer fault
DNS configuration is unchanged; it returns records for faulty LB
Client sends INVITE to faulty LB; ICMP received
Client call failure
Client sends INVITE to non-faulty LB
Client call success
Proxy configuration is unchanged; it keeps sending INVITE to faulty LB
Proxy generates 408 Request Timeout
Client call failure

Figure 4.1: Effects of load balancer fault without automatic failover.

4.1.1 Load balancer fault

Figure 4.1 shows the effects of a load balancer fault. Two other components are affected by the fault: DNS server and proxy.

The top half shows that without automatic failover, the DNS configuration remains unchanged after the fault. Whether a new incoming call succeeds or fails depends on client behavior. If the client tries one of the load balancers, fails, and stops, then the call fails. Of course, the client could be lucky and choose a load balancer that is alive. If the client is smart, it would try all the load balancers returned by the DNS server before giving up.

The bottom half shows that the proxy is also affected by the fault. When a proxy relays a SIP INVITE message to the recipient, the proxy may have chosen a faulty outbound load balancer. In this case, the proxy repeatedly sends the SIP INVITE message to the faulty load balancer and eventually times out. The proxy generates a 408 Timeout reply, which traverses the Via header chain to the original sender and the call fails.

4.1.2 DNS server fault

The DNS server is the first component that a client contacts for service discovery. An ideal DNS server replies to the client's NAPTR and SRV queries with the name, address, and port of available load balancers.

Figure 4.2 shows what happens when the DNS server fails. Much depends on the behavior of the DNS resolver, usually located close to the client. Ironically in this case, an agile DNS resolver which is quick to refresh its cache to get up-to-date information may actually hurt the client's chance of making a successful call. This happens because the DNS resolver cannot reach the authoritative DNS server and lets the client know that it is unable to locate any server. On the other hand, a DNS resolver that retains stale records in its cache may be beneficial to the client since load balancers may still be alive even though the DNS records are stale.

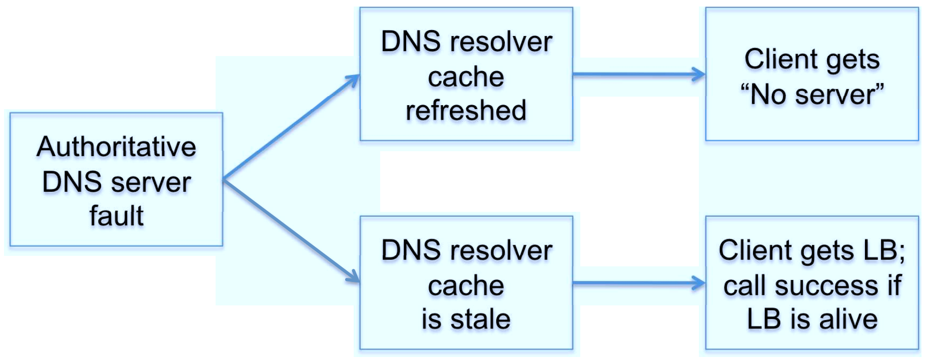
Authoritative DNS server fault
DNS resolver cache refreshed
Client gets "No server"
DNS resolver cache is stale
Client gets LB; call success if LB is alive

Figure 4.2: Effects of DNS server fault.

4.1.3 Proxy fault

Proxies are the main workhorse of the system. If a proxy fails and recovery steps are not taken, a new call destined for that proxy eventually fails. This is shown in Figure 4.3.

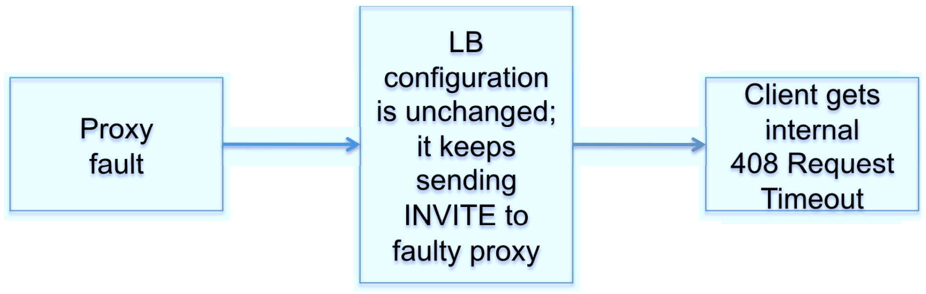
Proxy fault
LB configuration is unchanged; it keeps sending INVITE to faulty proxy
Client gets internal 408 Request Timeout

Figure 4.3: Effects of proxy fault.

4.1.4 Database fault

A database instance contains user registration information, which is vital to relaying calls to its intended recipient. As Figure 4.4 shows, a database server fault results in the proxy generating a 500 Internal Server error. This leads to service failure.

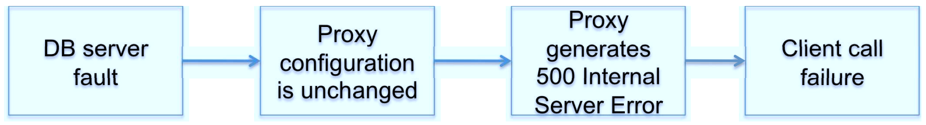
DB server fault
Proxy configuration is unchanged
Proxy generates 500 Internal Server Error
Client call failure

Figure 4.4: Effects of DB fault.

4.1.5 Media relay fault

Media relay failover mechanism is implemented by default in the proxy; therefore, there is no cause-and-effect analysis for this fault.

4.1.6 Conference server fault

Figure 4.5 shows the effects of conference server fault on the proxy. The proxy, without any failover mechanism, keeps sending the SIP INVITE message to the conference server and ultimately times out. It then generates a 408 Request Timeout message, which is sent to the client.

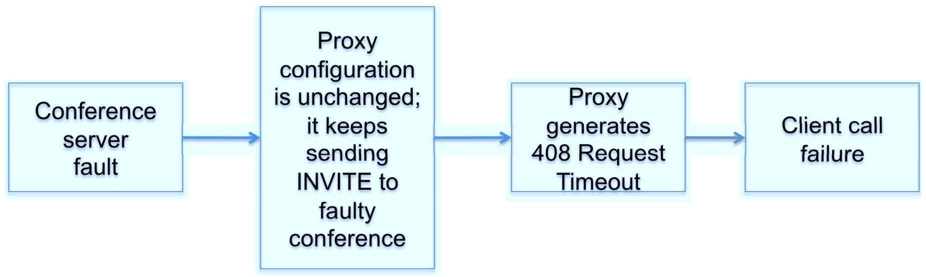
Conference server fault
Proxy configuration is unchanged; it keeps sending INVITE to faulty conference
Proxy generates 408 Request Timeout
Client call failure

Figure 4.5: Effects of conference fault.

4.2 Effects of a single component fault with automatic failover

This section shows results of fault experiments where components had some form of automatic failover capability. What the failover capability looked like, and the result of the experiment are described next.

4.2.1 Load balancer fault

There are two components that are affected by a load balancer fault: DNS server and proxy, as shown in Figure 3.4. The top portion of Figure 4.6 shows that while the records on authoritative DNS server are updated automatically, the client experience depends on whether the DNS resolver's cache is refreshed. As opposed to the case where automatic recovery was not implemented, an updated cache results in call success since faulty load balancers are removed from the DNS answer section. The lower portion of Figure 4.6 shows that the proxy server was not successful in failing over to the other load balancer. The proxy generates a 408 Request Timeout message, which is relayed to the caller by traversing the Via header list. The reason for an unsuccessful failover attempt is that the outbound load balancer is statically configured in the proxy.

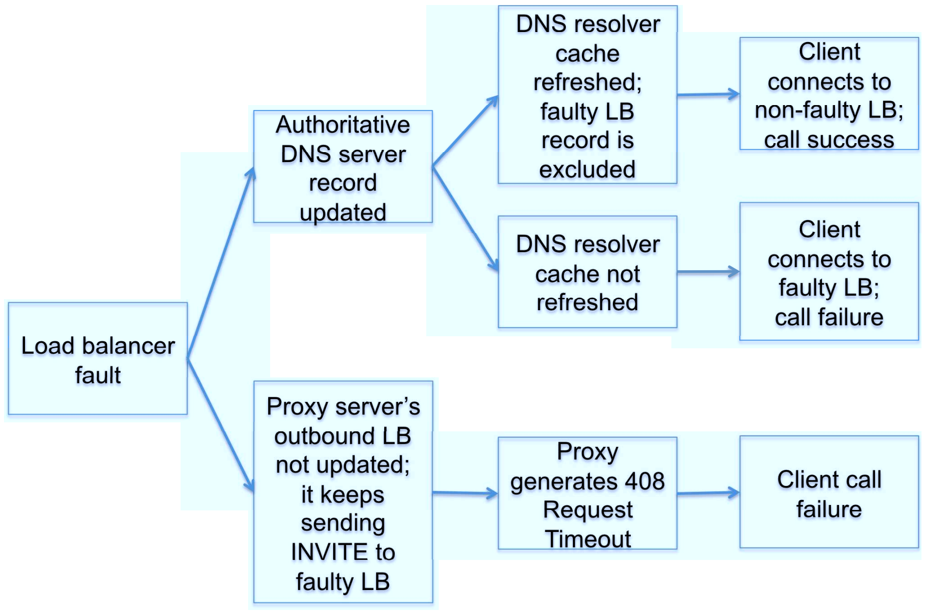
Load balancer fault
Authoritative DNS server record updated
DNS resolver cache refreshed; faulty LB record is excluded
Client connects to non-faulty LB; call success
DNS resolver cache not refreshed
Client connects to faulty LB; call failure
Proxy server's outbound LB not updated; it keeps sending INVITE to faulty LB
Proxy generates 408 Request Timeout
Client call failure

Figure 4.6: Effects of load balancer fault with failover mechanism.

4.2.2 DNS server fault

We did not experiment with authoritative DNS server faults. Authoritative DNS servers are usually setup with a primary server and one or more secondary servers. This redundancy is typically required when registering a new domain. Therefore, a DNS server fault would be recovered with normal DNS operation.

4.2.3 Proxy fault

With automatic failover, a proxy fault leads to the reconfiguration of the dispatch list in all load balancers in the system. After the reconfiguration, load balancers no longer send messages to the faulty proxy, so all new incoming calls are processed correctly. This is shown in Figure 4.7.

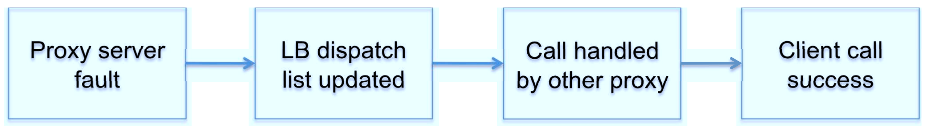

Figure 4.7: Effects of proxy fault.

4.2.4 Database fault

The proxy is the only component affected by a database fault. The proxy recovers from a database fault by connecting to another database instance, if the original instance fails. This leads to successful processing of new incoming calls, as shown in Figure 4.8.

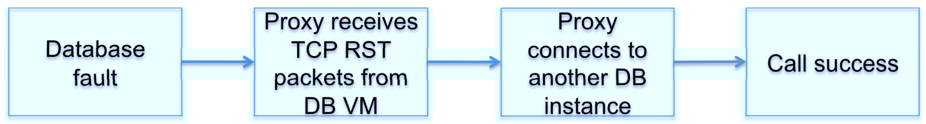
Database fault
Proxy receives TCP RST packets from DB VM
Proxy connects to another DB instance
Call success

Figure 4.8: Effects of DB fault.

4.2.5 Media relay fault

Media relay fault only affects the proxy. The proxy recovers from a media relay fault by connecting to the other media relay, just like when it deals with a database fault, which leads to successful processing of new incoming calls. Figure 4.9 shows the cause-and-effect diagram of a media relay fault.

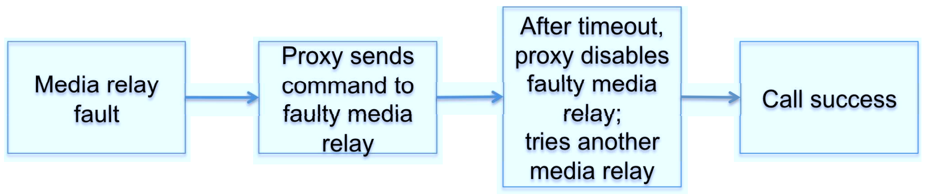
Media relay fault
Proxy sends command to faulty media relay
After timeout, proxy disables faulty media relay; tries another media relay
Call success

Figure 4.9: Effects of media relay fault.

4.2.6 Conference server fault

Like the previous faults, a conference server fault affects only the proxy. The proxy tries the faulty conference server first. But after sending the SIP INVITE message until retransmission timeout, the proxy sends the SIP INVITE message to the other conference server, which accepts the call and processes it. This is shown in Figure 4.10.

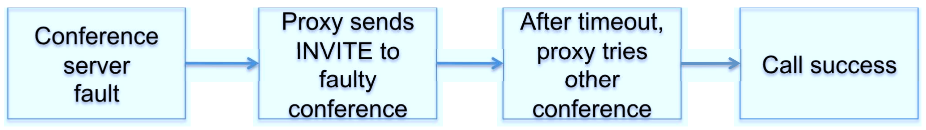
Conference server fault
Proxy sends INVITE to faulty conference
After timeout, proxy tries other conference
Call success

Figure 4.10: Effects of conference server fault.

4.3 Discussion

With an automatic recovery scheme, most problems with processing new incoming calls were solved. One failure was due to stale DNS resolver cache, which is outside the control of the voice service provider. The other failure was due to a static configuration, which could not be updated while the proxy was running. These problems and a few other issues discovered during the experiments are discussed in this section.

4.3.1 DNS issues

Resolver cache

DNS resolver cache has a big role in whether a new call can be successfully delivered to the servers or not. However, the resolver is outside the control of the voice service provider and this makes it a hard problem to solve.

It would be interesting to see how much of the resolvers in the wild actually adhere to TTL values mandated by authoritative servers. If most resolvers respect TTL values, and refreshes the cache accordingly, the problem may not be a big issue for the service providers. At this point, we're not sure how big a problem this may be.

A way to side step the issue might be to distribute clients that use all DNS answers before giving up on the service. As shown in the results on load balancer faults and authoritative DNS server faults, even if the cache does not respect TTL values and the records are stale, some of the load balancers that the (stale) records point to may actually be alive. Therefore, a client that tries to use all DNS records may still be able to get service.

Redundant authoritative servers

It was explained earlier in Section 4.2.2 that multiple authoritative DNS servers are configured so that any one fault does not bring down the entire service. In this project, we didn't experiment with a redundant setup of authoritative DNS servers but we note that an update to the zone file in a primary DNS server must be synced with all the backup DNS servers. In this section, how this may be done using normal DNS operations or otherwise, is explained.

DNS zone transfer is a standard DNS record replication procedure that uses the AXFR protocol [14]. The protocol is initiated by the secondary DNS server, which connects to the primary server for any changes to the zone file, and if there are changes to the serial number, it requests for all records and receives them. There is a problem with this approach because secondary servers may not be synced before the primary server fails. To deal with this problem, primary servers can send a NOTIFY message [20] to secondary servers.

Another method for DNS record replication is to use a file sync program, such as rsync, or use the replication mechanisms of the database management system itself. Also, in our project, the Load Scaling Manager can be used to propagate record changes to all authoritative DNS servers. However, these mechanisms are only possible if we have update privileges to the secondary servers. Other organizations may agree to host a secondary server for us, but whether we would be given update privileges is a separate matter.

4.3.2 Static configuration issues

When a statically configured component fails, it is hard for a dependent component to recover from that fault. An example was shown in Figure 4.6 where a load balancer fault resulted in a system failure; the faulty load balancer was statically configured in a proxy as the outbound load balancer. Since the proxy configuration could not be updated on-the-fly, it failed to relay the SIP INVITE message to the recipient.

This issue is relevant for all other types of faults as well, even the ones where our results show that the system was successful in dealing with faults. The limitation of our experiment is that the recovery mechanism was statically configured. For example, for conference server faults, the proxy was configured to try one first, and if there is no reply try the other. The configuration was static in the sense that IP addresses of the primary and backup conference servers are hard-coded into the recovery mechanism.

There are two problems with this kind of static configuration: first, it is not scalable, and second, it is not ideal for cloud platforms where VM instances and the IP address assigned to the VM instance may be gone forever. The first point is clear. If we had 100 media relay instances, we would have to statically configure all of them in each proxy. If a media relay is permanently dead, there is no way to take it out of the proxy configuration unless we kill the proxy, reconfigure it, and then restart it. Obviously, this is not a good management strategy.

The second point is where the problem becomes bigger for statically configured components. If a dependent component fails, and a new replacement component is started with a new VM instance, it would have a new IP address. The new component cannot be used unless configurations can be updated dynamically.

4.3.3 Connection reuse problem

During the experiments, we found a problem with using multiple load balancers. The problem is shown in Figure 4.11. In the Figure, Client B has a TCP connection to the bottom load balancer. When Client A sends a SIP INVITE message destined for Client B, the message first goes through the top load balancer and is processed by the proxy. The proxy looks up the registration data for Client B and sends the message to its outbound load balancer, which happens to be the top load balancer. However, the top load balancer cannot relay the message to Client B since it does not have an existing TCP connection to Client B; and it drops the message. The same problem would occur if Client A or Client B was behind a restrictive NAT.

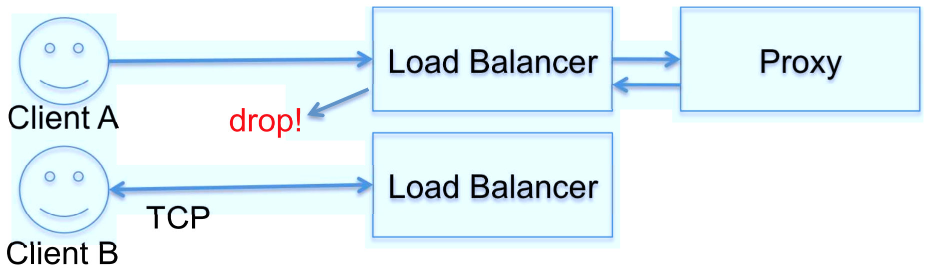

Figure 4.11: Connection reuse problem illustrated.

This problem may be side-stepped by using a single powerful load balancer or a active-standby configuration for redundancy.

If multiple load balancers are to be used, one potential approach is for the load balancers to share client connection information, and relay the message to whichever load balancer is responsible. A similar approach is that if the proxy can view the client connection information, the proxy may be able to choose the outbound load balancer based on the information.

Chapter 5

Recommendations for a resilient system

In a resilient system, the effects of a fault does not reach system boundary and is therefore not experienced by the client. One way we looked at masking the fault is for a dependent component to automatically contact a non-faulty component. The results were presented in Chapter 4.

However, as it was noted in the discussions section of Chapter 4, there are fundamental problems not addressed by the current system architecture and components. One problem is the DNS cache problem, which is outside the control of service providers. Another problem is that some components, e.g. proxies, are statically configured and its configurations cannot be modified on-the-fly.

In this chapter, we recommend some approaches that may lead to a more resilient system.

5.1 Smarter client

In our experiments, we ignored the effects of client software on service availability. The reason is that the service provider may not have direct control on what kind of client software their customers are allowed to use. However, the client software could play a role in enhancing service availability. This section describes some techniques that the client software can use to enhance service availability.

5.1.1 Being aggressive about contacting load balancers

As shown in Figure 4.1 and Figure 4.6, the client we used in our experiments gave up after one try to reach a load balancer although it had received DNS records for both the faulty load balancer and the non-faulty one. It tried the first record that pointed to a faulty load balancer, didn't succeed in getting a response, then stopped trying altogether. If the client software had been aggressive about contacting all load balancers, it would have succeeded in making a new call.

5.1.2 Remembering load balancers

In Figure 4.2, the client fails to make a call because the DNS resolver returns a “No Server” message. In this case, the client could remember the last load balancer that successfully responded to its request and try to contact that load balancer. Since the cause of the issue is a faulty authoritative DNS server, it is likely that the load balancer is still available to process new calls.

However, this kind of client behavior may backfire if applied to other cases. For example, in the case where a load balancer is being retired for maintenance and left out of the DNS records, a client that remembers and keeps using this load balancer will eventually fail. Therefore, client software should use this in situations where it can’t contact any load balancer. DNS resolver returning a “No Server” message fits this situation.

5.1.3 Re-registering after load balancer failure

Due to the connection reuse problem, as shown in Figure 4.11, a load balancer fault leads to a ’lost’ client, meaning that it can’t be reached anymore if the client is behind NAT or connected by TCP. The client had registered properly but cannot receive any message from the system. In this case, the client can help the situation by re-registering with another load balancer.

5.2 Dynamic configuration of failover components

One of our findings from the experiments is that static configuration of failover components is problematic for fault recovery when utilizing the cloud platform. There are existing techniques that may be helpful in allowing components to configure it during runtime in a dynamic environment. These are described in this section.

5.2.1 Runtime reload of configuration

Load balancers in our system already have a method of reloading configuration during runtime. A load balancer can read the dispatch file while it’s running and then send messages only to the proxies that are listed in the file. However, applying this technique to other software requires code-level changes.

5.2.2 Reliable Server Pooling

Reliable server pooling (RSerPool) [10] is an IETF standard for managing server clusters. It allows servers to join or leave a group, and servers in the group are monitored for liveness by management nodes called “Pool Registrars” (PR). Each group has a name with which the client can request the service of the group. The name is called “Pool Handle” (PH). The client first resolves the PH and gets a list of PRs. PRs are responsible for selecting available servers in the group and sending the list to the client. The client uses the list to connect to servers. If a server fails, the client connects to another server in the list.

This can be applied to components on cloud platforms. For example, the media relays can form a group and get a name called “media relay group”. The proxy is configured with a group name instead of a list of IP addresses. When a proxy needs to select a media relay, it would resolve the name “media relay group” and receive a Pool Registrar. The Pool Registrar would build a list of selected media relays and present it to the proxy. The proxy can then use the list to pick media relay. If a media relay fails, it can pick another from the list or ask the PR for another list.

RSerPool is a software building block for components. As such, it is only useful if the components must be built from scratch or can be augmented at the code level with RserPool.

5.2.3 Sharing a single contact address

Syzmaniak et al. [19] describes a way for multiple servers to share a single contact address by utilizing mobile IP version 6 (MIPv6). This can be applied to the problem of configuring failover components. The benefit of using a single contact address is clear: it allows static configuration up front, and failovers are handled dynamically behind the scenes.

MIPv6 supports this kind of behavior through an entity called “home agent” and by separating two addresses: a contact address and a care-of address. The contact address is the single address that is shared by a group of servers. The care-of address is the an address used by an individual server. The home agent’s task is to receive the message destined for the contact address and relay it to one of the individual server, while keeping track of which sender was redirected to which server. In a cluster of servers, one of the servers can be the home agent for all the other servers. The home agent handles failover transparently.

Applying this to voice services, each group of components can have a single contact address. For example, all conference servers may share a single contact address. The proxy can then be statically configured with just one IP address, i.e. the contact address of conference servers. One of the conference servers acts as a home agent and relays all messages to individual conference servers. If a conference server fails, the home agent can then start sending new incoming calls to another one that’s available.

How to handle a home agent failure is not clear. Another server can become a home agent but state in the original home agent will be lost. This is not a problem for new incoming calls.

5.3 Summary

There were two problems that were not solvable with simple automatic failover used in this project: the DNS cache problem and the static configuration problem. The first problem may be solved by making clients smarter, that is, more aggressive in trying to contact load balancers, remembering load balancers, and proactively re-registering in case of a load balancer fault. The second problem may be solved by using components that allow dynamic configurations, by building components that use RSerPool, or by using "home address" techniques from mobile IP version 6. These solutions have potential to make the system more resilient to component faults.

Chapter 6

Related Work

Oppenheimer et al. [16] looked into why internet services fail. The authors used similar categories as those of Gray [11] - node hardware, network hardware, node software, network software, environment, operator error, overload, or unknown - to describe faults that lead to internet service failure. However, their method of investigation is different from ours. They perused problem reports generated by service operators after major service failures to find out how and why services fail.

A real world example of a fault that brought down many components occurred in Amazon's Elastic Compute Cloud (EC2) service [2]. According to the post-mortem report [1], the root cause of the service failure was a configuration error during normal network scaling activity. Traffic destined for high-capacity primary network was temporarily switched to a low-capacity secondary network that was designed only for data replication between storage components. This resulted in storage components being completely isolated from each other, triggering an aggressive re-mirroring storm after the network was restored. The re-mirroring storm ultimately affected other parts of the system, bringing down larger parts of the service not directly affected by the storm.

Chapter 7

Conclusion

As SIP and RTP-based real-time communication services migrate to cloud platforms, scalability and high availability of such services will become a question of interest. In our previous project, we've studied the scalability aspect. In this project, we have focused on high availability; we haveanalyzed how single component faults affect service availability and identified several problems that a service provider would likely face when deploying such systems on cloud. One problem is with DNS resolvers that cache records from authoritative servers. Another problem is with using components that do not allow failover targets to be dynamically configured on-the-fly. It has been shown through experiment results that a simple failover strategy works in a small-scale system. However, this is not a good solution for cloud platforms where scale can easily vary.

Since we implemented a real system on Amazon EC2 and experimented on that system, we were able to share our results and experiences to write recommendations on how to build a system resilient to component faults. The two recommendations are to use smarter clients for DNS and load balancer faults, and to use dynamic configuration tools when deploying components on cloud platforms.

In summary, the main outcomes of the project are:

1. Cause-and-effect diagrams that came out of the analysis of single component faults.
2. Identification of problems unique to real-time communications services on cloud platforms.
3. Recommendations on how to solve the identified problems.

As future work, we intend to continue looking into the problems that were identified and search for better solutions with the grand vision of building an autonomous, massively scalable, and highly available SIP infrastructure.

Bibliography

[1] *Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region*, 2011 (accessed January, 2012). `http://aws.amazon.com/message/65648/`.

[2] *Amazon EC2*, 2012 (accessed March, 2012). `http://aws.amazon.com`.

[3] *The Apache Cassandra Project*, 2012 (accessed March, 2012). `http://cassandra.apache.org/`.

[4] *Asterisk*, 2012 (accessed March, 2012). `http://asterisk.com/`.

[5] *MySQL*, 2012 (accessed March, 2012). `http://mysql.com/`.

[6] *PowerDNS*, 2012 (accessed March, 2012). `http://pdns.org`.

[7] *RTPproxy*, 2012 (accessed March, 2012). `http://rtpproxy.org/`.

[8] *The SIP Router Project*, 2012 (accessed March, 2012). `http://sip-router.org/`.

[9] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1:11–33, January 2004.

[10] Thomas Dreibholz and Erwin P. Rathgeb. RSerPool - providing highly available services using unreliable servers. In *Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications*, EUROMICRO '05, pages 396–403, Washington, DC, USA, 2005. IEEE Computer Society.

[11] Jim Gray. Why do computers stop and what can be done about it? Technical Report TR 85.7, Tandem Computers, 1985.

[12] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, ASID '06, pages 25–33, New York, NY, USA, 2006. ACM.

[13] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 329–339, New York, NY, USA, 2008. ACM.

[14] P. Mockapetris. Domain Names - Concepts and Facilities, RFC1034, 1987.

[15] Edmund B. Nightingale, John R. Douceur, and Vince Orgovan. Cycles, cells and platters: an empirical analysisof hardware failures on a million consumer pcs. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 343–356, New York, NY, USA, 2011. ACM.

[16] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do internet services fail, and what can be done about it? In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association.

[17] D. Patnaik, A. Bijlani, and V.K. Singh. Towards high-availability for IP telephony using virtual machines. In *Internet Multimedia Services Architecture and Application(IMSAA), 2010 IEEE 4th International Conference on*, pages 1 –6, Dec. 2010.

[18] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. Sip: Session initiation protocol, 2002.

[19] M. Szymaniak and G. Pierre. A single-homed ad hoc distributed server. Technical Report IR-CS-013, Vrije Universiteit, 2005.

[20] P. Vixie. A mechanism for prompt notification of zone changes (DNS NOTIFY), RFC1996, 1996.