A Comparative Study of Failover Schemes for IaaS Recovery

Wasiq Noor Ahmad Qasmi, Taimur Siddiqui, Muhammad Karam Shehzad, Adnan Iqbal, Muhammad Saqib Ilyas
Namal College, Mianwali, Pakistan
{wasiq.noor, taimur2013, karam.shehzad, adnan.iqbal, saqib.ilyas}@namal.edu.pk

Abstract—Uninterrupted functioning of cloud infrastructure is critical to many everyday personal and professional activities. Preventing natural disasters from affecting cloud services is, therefore, important. Failover mechanism is an important component of any disaster recovery scheme. We compare three commonly used failover schemes analytically as well as empirically and present a Markov chains based generic model of disaster recovery. We implement disaster recovery approaches studied analytically in OpenStack and report results from several experimental settings. Analytical and empirical results suggest that having multi-level redundancy makes it more likely to recover quickly from different disaster scenarios.

Keywords—Disaster Recovery, Failover, Markov Chains, OpenStack

I. INTRODUCTION

Cloud computing has seen wide-spread adoption over the recent past. Due to this proliferation, cloud infrastructure may be considered critical infrastructure. This infrastructure is composed of a diverse set of devices with varying levels of reliability. To prevent outages, operators incorporate redundancy into the system using mechanisms such as redundant power supplies, replicated storage and backup network routes. However, each of these protection mechanisms has its scope and limitations. Natural disasters can wipe out most of these protection mechanisms. In order to provide continued service in the face of natural disasters, Disaster Recovery (DR) sites are deployed, which are replicas of the primary site.

When a disaster is detected, the DR site takes over and starts provisioning the cloud services - a step known as failover. One common method of achieving failover is to point the fully qualified DNS name of the hosted service to the corresponding servers at the DR site. This change, however, is dependant upon several factors including client’s local DNS server’s TTL setting [1]. Another method of achieving failover is to use a frontend server which reroutes client traffic to the DR site. Such schemes provide much quicker failover but fail completely if the frontend fails.

Multi-level failover schemes are also used in which multiple frontends and redundant servers collectively provide disaster recovery service. These schemes alleviate the non-failure assumption of the frontend. Recovery from failure of the primary server is quick, as in frontend based schemes. In case of frontend failure, a DNS based recovery is invoked to redirect clients to the secondary frontend. Thus, the DR system’s reliability and agility is improved, but the overall cost of the system increases.

Most of the prior work in the area of disaster recovery has been client focused. For instance, using cloud services as replicas of local resources to mask the latter’s failure. In this paper, we focus on disaster recovery for an IaaS cloud from the operator’s perspective. The experiences from work on disaster recovery in distributed systems, in general, are applied and evaluated in the specific case of an IaaS cloud.

This paper presents an analytical and empirical comparative study of three failover schemes - DNS based, frontend based and hybrid of both. At first, paper describes simple failure probability models for each scheme. Then a generic Markov chain based disaster recovery model is presented.

This paper also presents a disaster recovery implementation supporting all three failover schemes in OpenStack - a platform to ease the deployment, operation and management of cloud infrastructure [2]. This choice of platform is driven by OpenStack’s significant industry adoption over the past few years. OpenStack is the de facto standard cloud management software which is composed of a collection of services to handle different kinds of tasks such as compute, network, storage and identity management.

Disaster recovery for an OpenStack cloud requires DR support in all OpenStack services. At present, many services in OpenStack lack DR features. Since OpenStack follows a plugable architecture and DR is a real requirement, OpenStack deploying organizations may integrate their own mechanisms for disaster recovery in their deployment of OpenStack. However, we are not aware of any such implementation publicly available for community use. We make our implementation of DR publicly available for broader community use [3].

Our implementation is successfully tested for disaster recovery of Keystone - a critical OpenStack service on which all other services depend. We tested our implementation under various conditions on our campus LAN as well as on Amazon’s cloud infrastructure. Our experimentation shows that disaster recovery may be performed in less than 4 seconds when using a frontend based scheme. The DNS based failover, on the other hand, took several minutes to recover from a disaster. We also found that multi-level schemes involving redundant frontend servers may be used to achieve improved resilience. However, due to DNS based failover for frontend selection, the time to recover from a frontend failure is similar to DNS based approaches.

Through this paper, we make the following contributions:
• Analytical models and evaluation of DNS, Frontend and Hybrid disaster recovery schemes.
• A generic disaster recovery model based on Markov Chains.
• A publicly available implementation of these schemes for OpenStack Keystone service.
• Empirical evaluation of these schemes under various scenarios focusing on failover time.

Before detailed discussion of contributions, the next section covers necessary background information.

II. BACKGROUND

A disaster recovery system has two primary objectives, namely, recovery point objective (RPO) and recovery time objective (RTO). RPO dictates the tolerance level on the amount of data that may be lost in case of a disaster. RTO dictates the tolerance level on the duration of the interval for which the services are disrupted by a disaster. Lower values of RTO translate into quick recovery of system from failure - also known as high availability.

The secondary service is kept in sync with the primary site. When a user issues a data modification request, the change can be made at the primary site and the replicas are notified of the change. The system can wait until the replica confirms that this change has been made before sending a response to the user. This is known as a synchronous scheme [4]. The drawback is that if the primary and replica sites are placed far away, the latency for the response to the user is quite high. The advantage is that the primary and replica can be maintained in perfect lockstep. In case of any error, the change can be rolled back and a negative acknowledgement may be sent to the user. This makes it possible to achieve an RPO of zero.

An alternative is to issue the response to the user as soon as the change has been made at the primary site, without waiting for replication to complete. This is known as the asynchronous scheme [4]. The advantage is that the latency of the response to user is lowered. The drawback is that the replicas may be inconsistent at times, i.e., RPO has to be higher than that for synchronous schemes.

According to the agility of replication between the primary and the replica, DR may be categorized into hot backup, cold backup and warm backup schemes [5]. A hot backup site is one where the replica servers are always available to run applications once the disaster occurs. A warm backup site is one in which the state of the DR site is updated periodically and the stand-by servers take time to assume active state. At any given time, the data at a warm backup site may be slightly stale. A cold backup site is one where backup is taken infrequently and servers are not available to provide DR immediately. The trade off amongst these options is between the cost of the solution, the strength of the guarantee about data loss and the time required to resume operations after a disaster [5]. In this paper, we focus on a comparative analysis of several warm backup based disaster recovery schemes.

A DR scheme must include continuous monitoring of the availability of the primary site. This can be done by passively monitoring heart beat sent by the primary (and the replica) or by periodically polling them. When a failure is detected, the client requests are redirected to the replica at the DR site. This step is known as failover. Traffic redirection for failover can be achieved in various layers of the network stack. This includes overlay network schemes on layer 2 [6]–[8], layer 3 [9]–[11], layer 4 [12], and layer 5 [13]. Software Defined Networking (SDN) can also be used to enable traffic redirection [14], [15].

DR for clouds has not been widely discussed in the research community. Usually, the cloud service providers tie up with disaster recovery vendors to provide DR over cloud where maintenance of backup and disaster site is provided by the disaster recovery provider [16].

There are a few proprietary DR solutions for cloud. VMware Sphere Cloud is a cloud management tool that provides DR as a Service and private clouds disaster recovery solutions [17]. Cloud services such as Amazon Web Services and Microsoft Azure can be used to provide DR for users by replicating their data and applications on the cloud [18]. To the best of our knowledge, currently there is no open source solution publicly available for DR in OpenStack.

III. FAILOVER SCHEMES

In this paper we consider three failover schemes described below:

DNS based failover: In this scheme, failover is provided by updating the service’s DNS entry to point to the IP address of the replica, as shown in Fig. 1a. The service is hosted on multiple sites, preferably on geographically different locations. One of the service hosts is designated as primary site and its IP address is registered with DNS servers. The service is accessed through fully qualified URL which requires a mapping of URL to IP before accessing the service. In case of disaster, the DR mechanism changes DNS records to map the URL to the IP of secondary host.

Frontend based failover: Frontend based schemes make use of two layers for service provision as shown in Fig. 1b. The frontend system is the public interface usually assumed to be fail safe while the second layer consists of actual servers providing the service. The failure of any server can now be dealt with internally and, therefore, failover offers lower latency as compared to the DNS based strategy. However, the system may fail completely if the frontend fails. Variants of frontend based schemes - such as VRRP [19] - are used in practice for disaster recovery in different domains.

Hybrid failover: The frontend based scheme may be made more resilient by having replicated frontends as shown in Fig. 1c. If the primary frontend fails, the secondary frontend takes control. Failover from primary to secondary frontend may be achieved using DNS. If any server fails, the failover is internal and frontend takes care of the transition.

IV. ANALYTICAL MODEL

This section presents different models for failover schemes described in previous section. It starts from system failure probability model for each scheme given the probability of
failure for individual nodes. Then it presents a Markov Chain based model for generic disaster recovery scheme.

A. System failure probability models

A DNS based failover scheme consists of $m$ servers where each server has a failure probability $p_k$. In this scenario, where $p_k$ is same for all the servers, total system failure probability becomes $p_m^k$.

In frontend based schemes, the system fails completely if all the servers fail or the frontend fails. Let $p_k$ be the probability of a server failing and $p_f$ be the probability of the frontend failing, then the probability of complete system failure is:

$$p_f^2(1 - p_f) + p_f$$

The frontend based scheme may be made more resilient by having replicated frontends (Hybrid). If two frontends are used, the probability of complete system failure is given by:

$$p_f^2 + p_k^2 - p_f^2p_k^2$$

Addition of one frontend server has reduced the probability of complete failure by an amount $p_f(1 - p_f - p_k^2)$.

In general, one can use an arbitrary number of service replicas and frontend servers. It can be shown that when using $m$ frontend servers and $n$ service replicas, the probability of complete system failure becomes

$$p_f^m + p_k^n - p_f^mp_k^n.$$  

B. Markov Chain based disaster recovery model

The disaster recovery process can also be modeled as a discrete time Markov chain. In a first order Markov chain, a random variable $X_k$ at the time instant $k$ is considered which can attain values in the set $\{x_1, x_2, \ldots, x_m\}$. The probability that a random variable can attain a state $x_i$ at a particular time instant depends only on the attained state at the previous time instant according to the Markov property of the first order Markov chains i.e. $P(X_{k+1} = x_j | X_k = x_j, X_1 = x_1, \ldots, X_1 = x_m) = P(X_{k+1} = x_j | X_k = x_j) = p_{jj}$ where $p_{jj}$ is the transition probability from the state $x_j$ to the state $x_i$ depicting transition from any state to any other state.

We model disaster recovery process as a discrete time Markov chain consisting of three states - Working, Failed and Repairing. As shown in Figure 3, probability for being in the Working state is given by:

$$\Pi_w = \Pi_w p_{ww} + \Pi_r (1 - p_{rr})$$

Similarly, the probability for being in the Failed and Repairing states are given by the following equations:

$$\Pi_f = \Pi_w (1 - p_{ww}) + \Pi_f p_{ff}$$
$$\Pi_r = \Pi_f (1 - p_{ff}) + \Pi_r p_{rr}$$

Furthermore, it must hold that:

$$\Pi_r + \Pi_w + \Pi_f = 1$$
Failed

Complete system goes down immediately.
Network connectivity for incoming and outgoing traffic is disrupted temporarily.
Keystone becomes non-functional, and therefore OpenStack is inaccessible.

This approach offers more flexibility in terms of disaster recovery deployment as it is also closer to microservice architecture, a widely recommended modern software engineering approach [20]. The other approach could be to provide single disaster recovery service on top of all OpenStack services. This requires identification of common critical components of all the services and performing a collective backend replication and synchronization. As compared to first approach, this may offer less overhead but also lacks flexibility of choosing which OpenStack services to be protected against a disaster.

In this study we follow separate implementation of disaster recovery for each service since it offers more flexibility in terms of disaster recovery deployment. Experiments reported in this paper are based on Keystone (OpenStack’s identity service) using DevStack distribution of OpenStack. Since OpenStack follows a pluggable architecture, it is possible to use a variety of database backends for its services. In this study, we use MySQL because of wide spread adoption.

To provide disaster recovery in Keystone OpenStack, we assume that Keystone is deployed on two geographically distant locations, primary and secondary sites. Only the primary site is active for users initially. The disaster recovery service on each site is further composed of three sub-services: Detection, Synchronization and Recovery Service. Although same three services are available on primary and secondary sites, their functionality differs based on type of site. Below is some description of each of these services.

**Synchronization Service**: At primary site the job of this service is to get workload description periodically and transfer it to secondary site. Since default workload description storage in OpenStack is MySQL, this implementation reads MySQL logfiles and converts them into text based queries. At secondary site, the synchronization service executes these queries therefore synchronizing both sites. The synchronization checkpoint is a cloud administrator defined parameter.

We do not use MySQL's default replication mechanism because of two reasons; 1) using default mechanism results into user’s loss of control over synchronization semantics and 2) different database management systems may use different replication mechanisms - a problem to be faced when database other then MySQL is to be used.

**Detection Service**: Since disasters can be of different types, both the sites have a detection service. The detection service at primary site runs periodic local diagnostics such as Keystone status check, MySQL status check or Network Failure check. The secondary site also runs periodic diagnostics remotely for primary site such as ICMP status check. The type of disasters supported in this implementation are listed in the Table I.

**TABLE I: Types of disaster supported in this implementation**

<table>
<thead>
<tr>
<th>Disaster Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network Failure</td>
<td>Network connectivity for incoming and outgoing traffic中断 temporarily.</td>
</tr>
<tr>
<td>Keystone Failure</td>
<td>Keystone becomes non-functional, and therefore OpenStack is inaccessible.</td>
</tr>
<tr>
<td>Large Scale Disaster</td>
<td>Complete system goes down immediately.</td>
</tr>
</tbody>
</table>
Recovery Service: The disaster recovery service is activated when a disaster is discovered. For each disaster discovered in this implementation requires different treatment in terms of recovery process. For instance, if disaster detection service on primary site discovers that Keystone service is unavailable then recovery service will restart the Keystone and no further action is required in case of successful start. However, if the detection service discovers that everything at the primary site is down then the secondary site is to be made active by running the Keystone service and declaring it the primary site. This is where failover schemes come into play. Depending upon failover mechanism selected by the cloud administrator, the required action is completed accordingly. Just like other components of OpenStack, configurations can be provided by setting user provided parameters in relevant configuration files.

This implementation is publicly available at [3]. To our knowledge, this is the only publicly available disaster recovery implementation for OpenStack services.

VI. EXPERIMENTAL SETUP

For empirical evaluation of schemes considered in this study, we have implemented all the schemes in OpenStack as described in previous section.

We have tested our implementation in two different scenarios. Detailed tests were conducted in LAN environment on local machines representing frontends, primary servers and secondary servers. In addition to these machines, another machine was used to launch user queries to measure downtime from user’s perspective. Periodically, synthetic failures were injected in to the primary machine and recovery time was observed. These experiments were repeated at least 15 times for a single failover scheme.

To test the functionality in a more realistic manner in the context of disaster recovery, we used virtual machines hosted on geographically dispersed data centers of Amazon Web Services. In particular, we used datacenters in Ireland, Sydney, Singapore and Tokyo. For instance, to test the proposed scheme with two Keystone servers and two frontends, we used all four sites. The Ireland and Sydney sites hosted both Keystone servers - primary and secondary. The Singapore and Tokyo sites hosted both frontends - primary as well as secondary.

VII. RESULTS

For each of the scenarios mentioned in the previous section, we have evaluated DNS based, the single and dual frontend based schemes using different types of disasters. In this paper, we report the results of network disaster of a site. The network disaster is emulated using a forced shutdown of network connectivity on a given machine through a remote script. Similarly complete shutdown of a site is emulated using complete shutdown of machine hosting the relevant service. We have also experimented with other disasters such as shutdown of sub-services which can be resumed using local diagnosis. Such disasters are not discussed here since recovery is local and does not require any failover.

In each experiment, we first established a working setup whereby a remote client is continuously accessing the Keystone service. Then, we create the disaster and note down the time when the service becomes unavailable. Meanwhile, the client continues to attempt to access the Keystone service periodically. We record the time when the service becomes available again. The duration of the interval for which the Keystone service is unavailable to the client is noted.

The total downtime consists of disaster detection time $t_d$ and failover time $t_o$. Since the disaster detection time is constant in all the schemes, we report here the failover time only. Table II gives the failover time for experiments conducted on our local machines. This table describes average failover time for at least 15 different experiments for a single scheme.

The results in Table II suggest that the frontend based scheme incurs lowest delay when there is a Keystone server failure (less than a second), however, it fails to recover if the frontend fails. On the other hand, the DNS based scheme has a large failover time - 131 seconds on average - due to slow DNS binding procedure. In contrast, the hybrid scheme also incurs similar time on average if there is a frontend failure. Scheme using two frontends can also recover from simultaneous failure of frontend and an actual server with a minor increase in average recovery time 135 seconds. However, the scheme would fail if either all the frontends or all the servers fail.

TABLE II: Average Failover Time (in seconds) in Local Testbed

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Server Failure</th>
<th>Frontend Failure</th>
<th>Server and Frontend Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNS</td>
<td>132</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Frontend</td>
<td>0.7</td>
<td>Fails to recover</td>
<td>Fails to recover</td>
</tr>
<tr>
<td>Hybrid (2 Frontends)</td>
<td>0.7</td>
<td>131</td>
<td>135</td>
</tr>
</tbody>
</table>

Even though the average recovery time for schemes using DNS is between 130 and 135 seconds, individual experimental results show lot of variance with 37 and 238 seconds reported as the minimum and maximum times for DNS. Results for Hybrid scheme in all the scenario containing frontend failure show similar variations but with higher minimum value. As compared to these schemes, frontend scheme shows consistent 0.7 seconds if a primary server fails.

Since disaster recovery makes more sense when the primary and secondary sites are geographically dispersed, testing in a local environment is not sufficient. Therefore, we also conducted tests with AWS EC2 instances in four different AWS zones. The cities where these instances were hosted alongwith their role in the test are given in Table III.

TABLE III: AWS datacenters with given roles

<table>
<thead>
<tr>
<th>Site</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Singapore</td>
<td>Secondary Frontend</td>
</tr>
<tr>
<td>Tokyo</td>
<td>Primary Frontend</td>
</tr>
<tr>
<td>Ireland</td>
<td>Primary Keystone Server</td>
</tr>
<tr>
<td>Sydney</td>
<td>Secondary Keystone Server</td>
</tr>
</tbody>
</table>

The results from the AWS testbeds - shown in Table IV - show the same trends as the results for the local implement-
tation. However, the failover time has changed a little. In the case of DNS based scheme, it has decreased from the local testbed. We believe that the decrease is due to closer proximity of the EC2 instance to the DNS servers. In case of frontend based scheme, the failover time has increased a little - from $< 1$ to $< 5$ seconds. We believe that this increase is due to increased geographical distance between the servers. The average round trip time between the four machines was only a few milliseconds in the local deployment whereas it was hundreds of milliseconds for the EC2 instances owing to the large geographical distances.

It has been reported that many Wall Street companies have DR sites with sixty to seventy kilometers of downtown Manhattan [6]. This choice must have been made to achieve low failover time, unlike the numbers that we obtained. However, the presence of primary and DR sites in such close proximity makes them more susceptible to being affected by the same natural or artificial disaster.

![Table IV: Average Failover Time (in seconds) in AWS Testbed](image)

<table>
<thead>
<tr>
<th>Keystone Server Failure</th>
<th>Frontend Failure</th>
<th>Keystone Server and Frontend Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNS</td>
<td>57</td>
<td>N/A</td>
</tr>
<tr>
<td>Frontend</td>
<td>$&lt; 5$</td>
<td>Fails to recover</td>
</tr>
<tr>
<td>Proposed (Hybrid)</td>
<td>$&lt; 4$</td>
<td>85</td>
</tr>
</tbody>
</table>

### A. Discussion

If a cloud administrator has no choice but to place its primary/secondary servers in geographically confined area then any approach involving DNS - pure DNS and/or Hybrid - will result into longer failover time. The frontend based approach is recommended in this scenario for low latencies.

If a cloud administrator has the option of deploying its primary/secondary servers in globally distant locations then approaches involving DNS may work better provided chosen locations are in close proximity of DNS servers.

### VIII. Conclusion

We have implemented disaster recovery features in OpenStack Keystone servive based on three different failover techniques. Evaluation of our implementation was carried out on a local virtualized environment as well as on EC2 instances. Our evaluations show that when using two replicas each for the Keystone service and the frontend redirection server can result in failover within a few seconds in case of Keystone failure. DNS based schemes however show greater variation in failover time. Current implementation is based on single OpenStack service and MySQL. We are already working to extend this implementation for other services and backends.

We have also provided some basic stochastic models of disaster recovery in cloud services. These models consider overall system failure probability for a given failover mechanism. However, these models do not yet consider system failure as well as latency simultaneously, which can be achieved by posing it as an optimization problem.

### References