# An Evaluation Framework for Highly Available and Scalable SIP Server Clusters

Jong Yul Kim
Computer Science Dept
Columbia University, USA
jyk@cs.columbia.edu

Gregory W. Bond
AT&T Labs Research
Florham Park, NJ USA
bond@research.att.com

Eric Cheung
AT&T Labs Research
Florham Park, NJ USA
cheung@research.att.com

Thomas M. Smith
AT&T Labs Research
Florham Park, NJ USA
tsmith@research.att.com

Henning Schulzrinne
Computer Science Dept
Columbia University, USA
hgs@cs.columbia.edu

## ABSTRACT

SIP server clusters provide scalability and high availability for SIP applications. The nature of telecommunications services dictates requirements for these capabilities that generally exceed what is necessary in the domain of web applications. There are a number of clustering solutions available to address these requirements, but due to significant differences in architecture and the lack of common terminology, the solutions are difficult to compare. In this paper, we propose system requirements that are necessary to achieve scalability and high availability. We then propose an abstract cluster architecture containing four common architectural elements. Solutions from three different vendors are described in terms of these abstract elements. The solutions are compared based on an evaluation framework that includes a number of criteria, such as robustness to failures of different types and numbers, overhead required to manage failures, and scaling efficiency.

## 1. INTRODUCTION

As VoIP deployments grow, service providers need to be able to handle increasing demand for their services while at the same time making sure that the service is highly available. There are two general approaches to addressing these problems. One is to scale up, which means supplementing existing SIP server hardware so that it handles more load, and using advanced software engineering methods to make the server software more resilient. The other is to scale out, which means adding more server hardware instances and organizing them into a SIP server cluster. As a group, the servers in a cluster can process more requests and also recover from server failures. Scaling out is the approach currently favored by service providers since it offers better price-performance than scaling up [7]. As a result, many SIP server vendors offer highly available SIP server cluster solutions.

While scaling out has also been adopted in the web services domain, there are important differences to consider between the telecom and web services domains. For most web applications it is acceptable to expect the user to resubmit a browser request (refresh the page) in the event that an HTTP server fails. In contrast, it is generally considered unacceptable for an in-progress phone call to be dropped when a SIP server fails, forcing the user to re-dial and re-establish their connection. In short, telecom service is expected to be more reliable than web service. Also, interactions between SIP servers and clients are session-based and long-lived. All but the simplest telecom services (e.g., proxying and registration) require the SIP server to maintain state for the duration of each call serviced.

The expectation of high reliability combined with the need to maintain long-lived state mean that the requirements for supporting highly available and scalable telecom services differ significantly from those for web services. Given the relative nascence of SIP server technology, there exists no standard approach to satisfying these requirements. Instead, SIP server vendors have developed approaches based on different architectures and technologies. As a result, there is no straightforward way to ascertain relative strengths and weaknesses of these approaches.

The results presented in this paper were motivated by a need to compare differing solutions to high availability and scalability for SIP server clusters. Being able to directly compare solutions is necessary to support making an informed decision when choosing a solution for a particular system deployment. It also provides insight into more fundamental problems associated with supporting high availability and scalability.

In order to achieve this motivating goal, this paper proposes a framework for evaluating highly available SIP server clusters. The framework consists of a set of system requirements and an abstract architecture.

The framework's requirements, described in Section 2, are properties that are necessary to support high reliability and scalability for any ideal SIP server cluster solution.

Related work is presented in Section 3.

The framework's abstract architecture, described in Section 4, exposes the salient elements of SIP server cluster solutions. The architecture provides a common terminology for describing a solution and a means for direct comparison across solutions.

In Section 5, we show how the abstract architecture maps to three different vendors' SIP server cluster solutions and how the constituent abstract elements interoperate for each solution.

Finally, in Section 6, we provide a detailed comparative evaluation of the three SIP server cluster solutions with respect to our framework's requirements. By doing so we are able to provide insight into the relative strengths and weaknesses of different approaches for providing high availability and scalability in SIP server clusters. The ability to compare these solutions also serves to demonstrate the efficacy of our proposed evaluation framework.

## 2. REQUIREMENTS OF A SIP SERVER CLUSTER

It is helpful to see what an ideal SIP server cluster would look like from the outside, that is, from the SIP client's point of view. At the highest level of abstraction, a SIP server cluster is simply a SIP application: either a proxy, a user agent client, a user agent server, or a back-to-back user agent.

An ideal application should "just work" all the time. This means that the client is not aware of any failure within the SIP server cluster (highly available) and that any number of clients and any call rate are supported by the application (scalable).

The rest of this section lists a number of criteria that fit under these two overarching requirements. We will use them to evaluate different designs and implementations of SIP server clusters. Furthermore, different deployments may use this list to help specify detailed requirements based on their specific circumstances.

### 2.1 High Availability

In order to differentiate varying degrees of availability, we adapt the "Availability Level Spectrum", originally developed for transactional servers by the analyst IDC [4], to telecom services. The availability level (AL) ranges from AL1 to AL4. AL1 means that in the event of a failure, the SIP service is unavailable until the failure is repaired. A service at AL2 remains available. Although calls are interrupted, they can be re-established if the users call in again. At AL3, all calls stay up but some ongoing operations may need to be repeated. At the highest AL4, any failure is transparent to the users.

This paper discusses SIP server clusters that implement the higher levels, AL3 and AL4, which require that the states of SIP dialogs and/or transactions are protected.

Detailed criteria related to the broad high availability requirement include:

**Types of failures** A cluster should be able to recover from various hardware and software failures, including rack failure, machine failure, and software failure.

**Simultaneous failures** A cluster is more robust if it can handle a higher number of failures occurring at the same time.

**Shift in workload upon failure** When an element fails, the remaining elements must take over the load. Therefore, each element must reserve some capacity in anticipation of a failure. The redistribution of load on the failed element should be as even as possible such that this capacity overhead can be low and resources can be used efficiently during normal operations.

**Performance degradation after failure** In the event of a failure, the overall system performance should not degrade substantially. For example, any additional latency in message processing due to time spent to recover the dialog or transaction state should be low.

**Loss of protection after failure** As described above, clusters support different numbers of simultaneous failures. After this number is reached and before the failed elements are repaired, high availability is compromised and another failure will result in permanent loss of state. Ideally, a cluster can automatically adapt and restore protection using the surviving elements.

### 2.2 Scalability

The scalability requirement mandates that any performance bottlenecks within the cluster can be handled with relative ease such that it can be scaled out to handle more load by adding resources. For example, if the problem is determined to be lack of CPU power to process SIP messages, it should be operationally easy to add extra servers to the cluster. Likewise, the cluster should make it operationally easy to add additional storage in case the total number of sessions it can handle is bounded by memory. For network bandwidth scaling, multiple clusters can be deployed.

Detailed criteria related to the broad scalability requirement include:

**Scaling increment** It should be possible to add resources at small increments to permit fine-grained tuning of the cluster.

**Operational cost of scaling** It should be straightforward to scale a cluster such that operational costs are kept low for the deployers.

**Scaling efficiency** A cluster should allow additional resources to be utilized as fully as possible. Also scaling out should consume as little additional network bandwidth as possible.

## 3. RELATED WORK

Previous work in this area has focused on specific SIP services at a lower level of availability. For example, [13] and [6] examine SIP servers that provide registrar and routing proxy functions. Because of the nature of the service, only application data (endpoint registrations) needs to be protected against failure. SIP dialog and transaction states are not protected by these techniques, so services that require statefulness will be compromised by mid-call failures. For example, a proxy server that also logs call duration will not operate properly in the event of a mid-call failure. In terms of availability level, the solutions described in these references only achieve AL2.

Another work of interest studies the use of virtualization technology [9]. Through virtual machine replication and migration provided by Xen and Remus, SIP applications are protected from failures of the physical host machines

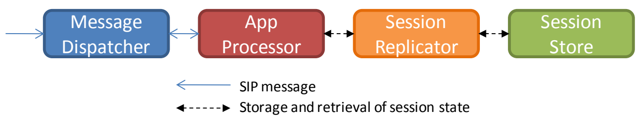

**Figure 1: Abstract elements of a SIP server cluster**

on which they run. However, this solution does not protect against failures in the operating system or application software. This is because the approach provides complete system state capture and replication, and as such will propagate application errors to the backup [3].

## 4. ABSTRACT ARCHITECTURE

Because there is no standard terminology to describe components of a SIP server cluster, different implementations describe their components using different terminology, thus making it difficult to understand their commonalities and differences. In order to understand and compare the functionality of SIP server cluster implementations, we propose an abstract architecture that both captures the common functionality across different SIP server cluster implementations, and highlights their differences. By casting different SIP server cluster implementations in terms of the proposed architecture, we are able to directly compare them.

In creating the abstract architecture, our goal was to maintain a level of abstraction that was detailed enough to reveal salient differences with respect to the requirements specified in Section 2, but not so detailed that less important implementation differences were revealed. The architecture is also intended to be general enough to be applied to future SIP server cluster implementations. As shown in Figure 1, the resulting architecture consists of four distinct abstract elements: the *message dispatcher*, the *app processor*, the *session replicator*, and the *session store*. As will be shown in Section 5, it is possible for an abstract element to map to more than one implementation element. Furthermore, it is possible for an abstract element to map to more than one host machine. We now describe each abstract element in turn.

### 4.1 Message Dispatcher

The message dispatcher receives SIP messages from the environment and dispatches them to app processor elements. Upon receiving a message from the environment, the message dispatcher algorithm determines which app processor should receive the message. For example, the algorithm may dispatch messages in a round-robin fashion, or it may dispatch to a particular app processor based on the hash of fields present in the message. The dispatcher may also add or manipulate message headers prior to dispatching. When an app processor goes out of service due to unexpected failure or scheduled maintenance, the dispatcher will cease sending messages to the failed app processor and send them to other app processors until the app processor comes back online. A message dispatcher may support app processor *affinity*, also known as *sticky routing*; that is, routing subsequent messages in the same SIP dialog to the same app processor that the initial request was routed to. In this case, the message dispatcher is said to be *SIP aware*. To counter host machine failure, the message dispatcher is deployed redundantly on different host machines. Message dispatcher instances normally monitor each other for failure. When a dispatcher fails, another dispatcher will take over for the failed one. Similarly, when a dispatcher comes online, it will take over some of the load from another instance.

### 4.2 App Processor

The app processor executes service logic in response to receiving a message from the message dispatcher. For example, upon receiving an initial request, an app processor may react by instantiating service logic to support voicemail service. In processing the message, the app processor needs to create and store *session state*. Session state is an umbrella term that refers to the three types of state that an app processor may maintain: *transaction state*, *dialog state*, and *application state*. Transaction and dialog state refer to the state of an individual SIP transaction and dialog, respectively. Application state refers to higher level state that service logic may maintain. For example, if the voicemail service logic were implemented as a state machine, then the current machine state would be maintained as application state.

Processing a message may result in messages being sent from the app processor, for example, to endpoints or media servers. In some message dispatcher configurations these messages may be sent via the dispatcher. App processors in a cluster are distributed across different host machines in anticipation of host failure. In some configurations, more than one app processor may run on the same host machine in order to better utilize the host machine resources. App processor instances may monitor one another for failure, or a message dispatcher may monitor them. In either case, a message dispatcher must be notified when an app processor fails or comes online. When an app processor fails, the app processor that takes over must be able to access the session state of the failed app processor; it does this via a session replicator.

### 4.3 Session Replicator

The session replicator is responsible for storing and retrieving instances of session state originally created and maintained by the app processor. The session replicator stores session state replicas on session store instances. The session replicator presents a simple interface to the app processor. The app processor *puts* a session state instance in order to store the instance, and it *gets* a session state instance by providing the replicator with a session state identifier, where the identifier is normally known a priori by the app processor.

In storing session state, the session replicator is responsible for serializing the state in preparation for transport to the (remote) session stores. When a session becomes invalid, for example, when a call is torn down, the replicator is responsible for purging the session stores of the associated session state. When a session store goes out of service, due to planned downtime or unexpected failure, the session replicator is able to access previously stored sessions from a replicated session store. While the session replicator is responsible for replicating an app processor's session state, the session replicator itself normally maintains no state of its own. This simplifies failover since another session replicator can take over from a failed session replicator without having to inherit the state of the failed replicator. Normally, the majority of the session replicator functionality is co-located

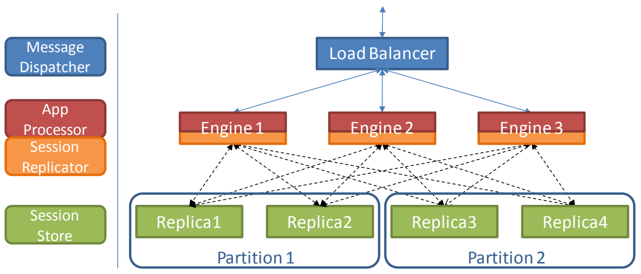

**Figure 2: Oracle Communications Converged Application Server Cluster**

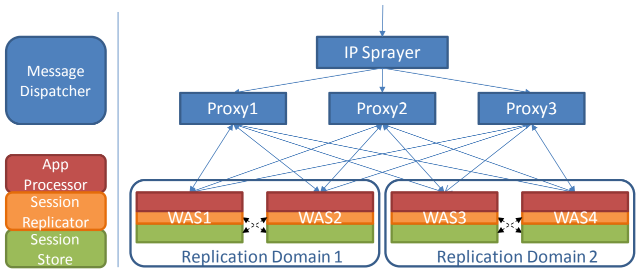

**Figure 3: IBM WebSphere Application Server Cluster**

with an app processor. This means that session replicators normally rely on the failover and recovery mechanisms that support app processors. For those replicator functions that may not be co-located with an app processor, such as replicating session state across session stores, a self-monitoring approach akin to that used by the message dispatcher may be used.

### 4.4 Session Store

A session store is responsible for maintaining session state stored by the session replicator. Session stores are distributed across different host machines in anticipation of host failure. Session stores may monitor one another for failure, or the session replicator may monitor them. Session store replicas are maintained in replication groups, where each session store in a replication group maintains identical information. When a session store fails, the appropriate replicators are notified and the replicators redirect access to another store in the failed store's replication group. When a session store comes online, the appropriate replicators can resume accessing the store.

## 5. DESCRIPTION OF COMMERCIAL SIP SERVER CLUSTERS

In this section, we describe the architecture of three SIP application server clusters using the four abstract elements introduced earlier. The server cluster implementation details described here have been obtained from publicly available documentation. All of these clusters support the Java SIP servlets specification [2]. Figures 2, 3, and 4 show the different cluster designs. On the right-hand side of each figure is the actual cluster design with terms used in the product manual. The left-hand side shows the corresponding abstract elements: the message dispatcher, app processor, session replicator, and session store. The solid lines show SIP messages and dotted lines show session storage and retrieval.

### 5.1 Oracle Communications Converged Application Server

The Oracle Communications Converged Application Server (OCCAS) architecture, as shown in Figure 2, is broken down into three entities. A *load balancer* is an implementation of the message dispatcher. An *engine* is a software process that combines the app processor and the session replicator. A *replica* is a software process that implements an in-memory session store. Replicas are grouped into *partitions*, and each partition is responsible for a subset of all active sessions in the cluster.

An engine does not include the session store element, therefore it is stateless. When there is any change in session state, the engine updates the replicas. To process messages, an engine retrieves session state from replicas. The advantage of a stateless engine is that if an engine fails, any other engine can resume the session by retrieving state from the session store. However, there is a cost of always having to store and retrieve state in each invocation. To reduce this overhead, engines are configured by default to cache session states locally.[1]

The session replicator is contained within the engine, meaning that the engine, not the replica, is responsible for replicating session states. An engine's session replicator works this way: when an engine needs to store session state, a hashing algorithm is applied to the session state to select a partition in which to store the session state data. The engine then writes the session state to *each* replica within that partition.[2] Therefore, replicas within a partition contain identical copies of the session state. This ensures high availability: if a replica fails, the engine can retrieve state from another replica in the same partition. Adding more replicas to the partition protects against multiple replica failures. However, each partition is limited to a maximum of three replicas[2] due to higher bandwidth usage between engines and replicas as the number of replicas increase.[3]

As for the load balancer, any type can be used since the engines are stateless and are able to process any message by retrieving the latest state from a partition. However, a SIP-aware load balancer would be able to take advantage of the local cache in the engines.[1]

### 5.2 IBM WebSphere Application Server

The IBM WebSphere Application Server (WAS), as shown in Figure 3, supports high availability by providing two cooperating components. The first component, labeled `WAS1` to `WAS4`, is referred to as the *SIP container*, and provides the app processor, session store, and session replicator functions. The second component, labeled `Proxy1` to `Proxy3`, is simply referred to as a *SIP proxy*, which provides part of the message dispatching function.

The SIP containers are grouped into *replication domains*, statically assigned by the administrator. Containers within a replication domain are in reciprocal primary-backup relationships: the containers back each other up. Up to three

[1] [8] page 6-1
[2] [8] page 1-6
[3] [8] page 3-3

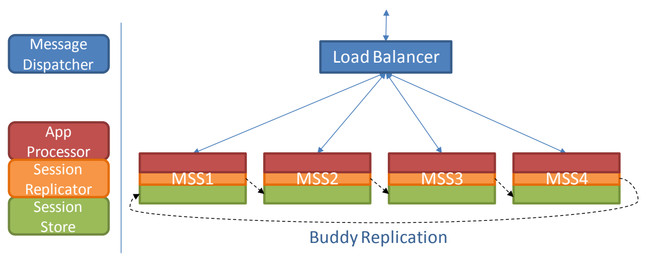

**Figure 4: Mobicents SIP Servlets Cluster**

containers can be grouped in a replication domain, though IBM recommends two containers per replication domain for performance reasons.[4]

Session state is held by the container and replicated from the primary to the backups as required; thus, the session store and session replication functions are also provided by the container. The session replicator within the container connects to its peers in the same replication domain. The replicator sends periodic heartbeats over this connection; failure conditions are detected by sensing connection termination. Once failure is detected, active sessions in the session store of the failed container are reassigned to the session store of another container in the same replication domain, and the change is reported to the SIP proxies. Since the same session state is maintained by all members of the replication domain, no migration of data is required at failure time, and the backup container can begin processing immediately.

The SIP proxy servers ensure that incoming messages are directed to the appropriate SIP container based on two criteria. First, the appropriate replication domain is determined by inspecting certain headers of the incoming message. If the message is part of a known session, then the message is proxied to the primary SIP container corresponding to that session. If the message does not correspond to a known session, then the proxy chooses one of the available replication domains and proxies the message to a SIP container in that domain.

Multiple SIP proxies can be employed for scaling and redundancy; each SIP proxy is capable of routing messages to the proper SIP container. When multiple proxies are used, they are fronted by a load balancer which the IBM product manual calls an *IP sprayer*. This load balancer does not need to be SIP aware. Unlike the SIP container and SIP proxy, which are tightly coupled, the IP sprayer may or may not be supplied by IBM, as its only task is to distribute load (to SIP proxies) using any desired algorithm. Thus the message dispatching function is split between the IP sprayer (for distribution) and the SIP proxy (for affinity). All incoming messages pass through the IP sprayer and one of the proxies; outgoing messages bypass the IP sprayer.

## 5.3 Mobicents SIP Servlets Server

Mobicents SIP Servlets Server (MSS) [11] is an open source SIP servlet container that runs on top of Apache Tomcat application server [1] (MSS for Tomcat) or Red Hat JBoss application server [10] (MSS for JBoss). Since MSS for Tomcat does not support clustering, the description below applies only to MSS for JBoss.

[4] *Replicating SIP Sessions* section in [5]

Mobicents provides a *SIP load balancer* and a *SIP servlets server*. To avoid confusion with the general meaning of these terms, we will call them *Mobicents load balancer* and *Mobicents server* respectively. The Mobicents load balancer implements the message dispatcher. The Mobicents server is a single software process that internally includes, among multiple parts, two components, the *Mobicents container* and the *JBoss Cache*. The Mobicents container implements the app processor and a small part of the session replicator function, namely the object serialization/deserialization. JBoss Cache [14] is an application-agnostic distributed cache which is used by the Mobicents container for reliable storage of session states. JBoss Cache implements the session replicator and session store functions.

In the default configuration of a Mobicents server, JBoss Cache instances find and communicate with each other using a shared multicast address and cluster name. The cache instances use 1 Buddy Replication, which means that session state is replicated to the cache instance that is "next in line" as shown in Figure 4. The cache instance that holds the replicated session state is called "buddy".

JBoss Cache provides the appearance of one large cache for all Mobicents servers in the cluster. When a Mobicents container needs to store session state, it internally calls a method provided by the JBoss Cache API. JBoss Cache makes sure that the session state is replicated to the buddy cache instance for high availability. When a Mobicents server fails, any other Mobicents server can take over since its JBoss Cache instance can retrieve the replicated session state from the buddy cache.

Due to this property of JBoss Cache, the Mobicents servers are essentially stateless. Therefore, like OCCAS, any type of load balancer can be used. The suggestion from Mobicents is to use a SIP-aware load balancer such as the Mobicents load balancer to make efficient use of the local JBoss Cache instance.[5]

## 6. EVALUATION OF CLUSTERS

In this section, we evaluate the three SIP server clusters in terms of high availability and scalability requirements described in Section 2. Although the clusters have different architectures and implementation elements, they can be compared directly as they have been decomposed into the four common abstract elements as described earlier in Section 5.

## 6.1 High Availability

A cluster supports high availability of sessions if it implements the session replicator abstract element. Without the session replicator, a session store failure leads to lost sessions. Such clusters can only achieve availability levels up to AL2. The session replicator is implemented in all three clusters described in Section 5. Session state is replicated between replicas in OCCAS, between SIP containers in WAS, and between Mobicents servers' cache buddies in MSS. Therefore, all three clusters can recover from a session store failure, thereby achieving availability level AL4.

We use a common deployment topology as shown in Figure 5 to compare the different cluster architectures. In the figure, there are two racks with two machines inside each rack. Each machine contains two abstract elements: the app processor and the session store. Pairs of session store

[5] [12] Section 5.3 Load Balancer

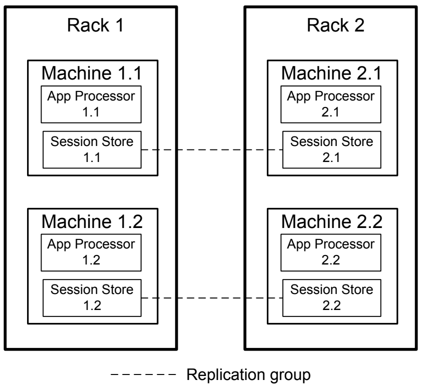

**Figure 5: Common deployment topology for evaluation**

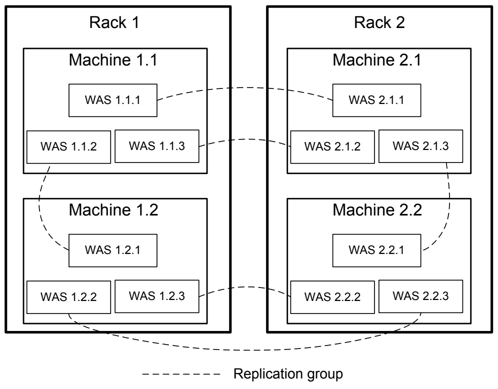

**Figure 6: WAS configuration that achieves 33% failure overhead**

elements form a replication group, which is deployed across different machines and racks. This reflects real-world strategies to ensure session recovery in case of failure. The focus of our evaluation is on high availability of sessions; therefore, the message dispatcher is not included in the figure. The function of session replicators is indicated by dashed lines that pair session stores into replication groups.

When applying the common topology to a specific cluster, we can replace the app processor and session store with the actual implementation element that contains the respective abstract elements. For example, applying this to OCCAS, app processors would become engines and session stores would become replicas. The two replicas in a replication group would form a partition. In WAS, each SIP container contains an app processor and a session store; thus, there is one SIP container per machine. SIP containers are paired in a replication domain across different machines and racks. Mobicents servers also contain both app processor and session store so there is one Mobicents server per machine.

Table 1 shows the summary of the high availability evaluation that follows.

### 6.1.1 Types of failures

Clusters using the common topology are protected from single process failures, single machine failures, and single rack failures since the implementation elements that contain the session store are deployed on different machines and racks. The same strategy can be applied to blades and blade enclosures since a blade failure is analogous to a machine failure and a blade enclosure failure is analogous to a rack failure. In OCCAS and WAS, the administrator can statically assign elements within a replication group to run on different racks. By default, Mobicents servers try to find a buddy on a different machine but not necessarily on a different rack. The JBoss Cache configuration property called *buddyPoolName* can be set to pair buddies on different racks [14], thereby achieving the replication group arrangement shown in Figure 5.

### 6.1.2 Simultaneous failures

Moving beyond the common topology, all clusters protect against multiple failures of session stores. By adding more session stores to a replication group, the cluster can tolerate more session store failures. However, there is a tradeoff: adding more session stores increases network traffic for session replication. In OCCAS and WAS, there is a limit of three session stores per replication group. Therefore, these clusters can recover from up to two session store failures. In MSS, the number of session stores per replication group is configurable to more than three, but at the cost of increased network traffic.

### 6.1.3 Shifts in work load after failure

Next, we evaluate how much the work load shifts to other implementation elements when there is a failure. Consider the effect on each cluster when `Rack 1` fails in Figure 5: in all three clusters, the load on all machines in `Rack 2` would double.

Clusters react differently when a machine fails. Let's assume that all app processors are handling the same amount of work load when `Machine 2.1` fails. Also, for OCCAS and MSS, the message dispatcher is assumed to evenly distribute the load to the remaining app processors in case of an app processor failure. The second assumption is not needed for WAS because the SIP proxy dispatches to another container in the replication domain.

In OCCAS, the number of engines is reduced from four to three so each remaining engine needs to handle 33% more load. In contrast, the number of partitions remains the same and each partition continues to handle the same number of sessions. The top partition is left with one replica in `Machine 1.1`. This replica's load for *put* operations does not change because the replica is still storing the same number of sessions. The replica's load for *get* operations doubles since the replica must handle load that normally would have gone to the failed replica.

In WAS, the replication domain affected by the machine failure would have one SIP container left. The SIP container needs to handle 100% more load while containers in the unaffected replication domain continue handling equal load as before. The overall result is that `Machine 1.1` handles 100% more load while `Machine 1.2` and `Machine 2.2` handle no extra load.

In MSS, existing and new sessions will be distributed evenly among the remaining three Mobicents servers as in OCCAS. For existing sessions, the Mobicents server selected by the load balancer pulls the replicated session state from the dead server's buddy and replicates that session to its own buddy. The replication is done lazily: a server does

| Evaluation criteria | OCCAS | WAS | MSS |
|---|---|---|---|
| Types of failure it can recover from | Process failure, machine failure, and rack failure | Process failure, machine failure, and rack failure | Process failure, machine failure, and rack failure |
| Recovery from simultaneous failures | Partitions can recover from up to two replica failures. | Replication domains can recover from up to two SIP container failures. | The number of replication buddies is configurable. Additional buddy leads to increased traffic. |
| Load shift after failure | Load is distributed among remaining engines across the cluster. In partitions, *get* operations are handled by remaining replicas. | Remaining SIP containers in a replication domain share load. Containers in other replication domains do not share load. | Load is distributed among remaining Mobicents servers across the cluster. |
| Additional message processing latency after failure | One-time latency | No additional latency | Probability of one-time latency increases proportional to cluster size. |
| Session loss | Happens when all replicas in a partition fail. Partitions are statically assigned. | Happens when all SIP containers in a replication domain fail. Replication domains are statically assigned. | Happens when all replication buddies fail, but buddy relationships dynamically adjust to failure. |

**Table 1: Summary of high availability evaluation**

not pull the session state unless it needs to handle the session. After this lazy replication process, the selected server becomes the new owner of the session. The dead server's buddy deletes the session from its cache. For its own session states, it finds a new replication buddy. This way, session states are automatically replicated to other cache instances. The overall result is identical to OCCAS.

As mentioned earlier, in this topology, WAS requires all containers to have 100% over-capacity to cope with a container failure. To reduce the over-capacity on containers, the administrator can add another container to the replication domain so that the remaining two containers handle 50% extra load instead of 100%. To reduce the over-capacity on machines, the administrator can assign more than one SIP container process to the machine. For example, to reduce the over-capacity to 33% as in OCCAS or MSS, the administrator could run three SIP containers on each of the four machines and assign partitions that span two different machines as shown in Figure 6. However, this kind of scheme would quickly become unwieldy as the cluster scales. Besides, this scheme does not protect sessions from rack failure. For example, if `Rack 1` fails, sessions handled by `WAS 1.1.2` and `WAS 1.2.1` cannot be recovered.

To summarize, in both rack failures and machine failures, the remaining processes must handle more load. In the case of machine failures, OCCAS and MSS require the same over-capacity but WAS requires more over-capacity.

#### 6.1.4 *Additional message processing latency after failure*

WAS SIP containers already store session states of other containers in the same replication domain. The backup SIP container only needs to activate the session states that belong to the failed container. Therefore, there is no additional message processing latency in WAS after a failure. On the other hand in OCCAS, when an engine fails, the load balancer picks another engine to take over the session. The selected engine does not have any session state of the failed engine in its local cache and must fetch session state from a replica to process the message. This leads to one-time additional latency. In MSS, the Mobicents server selected by the load balancer to take over a session of the failed server may be a buddy which already has the session state in its local cache instance. For example, if `Machine 2.1` in Figure 5 fails there is a 33% chance that the buddy server in `Machine 1.1` will be selected. In this case, there is no additional latency in message processing. In the other 67% of cases where a buddy is not selected, there is a one-time additional latency. The probability that a buddy is selected is inversely proportional to the cluster size.

Therefore, WAS has no additional latency, OCCAS has a one-time latency, and MSS has a one-time latency for a higher proportion of calls as the cluster grows.

#### 6.1.5 *Exposure to loss of sessions before recovery*

All clusters have a potential to lose session state before recovery. OCCAS and WAS both support automated process restarts for quick recovery from process failures. However, there may be times when automated recovery is not possible, such as when there is a rack failure or a machine failure. During these incidents where an administrator has to intervene, OCCAS and WAS clusters are vulnerable to session loss. For example, all session states in an OCCAS partition are lost if, after a replica fails, the other remaining replica also fails before the administrator has a chance to recover from the first failure. This leads to service failure for those sessions. The is because OCCAS and WAS clusters have replication groups statically assigned; therefore, session states are not protected until the failed session stores in the same group are restarted.

As mentioned before, MSS adapts dynamically to failures through lazy replication and selection of new replication buddies. This allows MSS to restore protection to machine failures and rack failures automatically even without restarting failed servers in the same replication group. Therefore, the odds of session loss in MSS are lower than in OCCAS or WAS.

### 6.2 Scalability

| Evaluation criteria | OCCAS | WAS | MSS |
|---|---|---|---|
| Scaling increments | Engine, or<br>Partition ($2 \sim 3$ replicas) | Replication domain<br>($2 \sim 3$ SIP containers) | Mobicents server |
| Operational costs | Adding an engine or a partition is easy. | Adding a replication domain involves careful planning for extra load after failure. | Servers with identical configurations automatically form a cluster. |
| Scaling efficiency | CPU-intensive units and memory-intensive units scale separately, so resource usage can be optimized. | Incremental unit adds both CPU and memory - leads to redundant resource. | Incremental unit adds both CPU and memory - leads to redundant resource. |
| Replication traffic | Sent to all replicas in a partition | Sent to other containers in a replication domain | Sent to other replication buddies, but can update field instead of whole session |

**Table 2: Summary of scalability evaluation**

How much load a cluster can handle is limited by the amount of parallel processing power, memory, or the network bandwidth. As mentioned in Section 1, scaling out is the more common approach when handling increased load, and all three clusters achieve scalability by scaling out. A summary of the evaluation that follows can be found in Table 2.

### 6.2.1 *Scaling increments*

The three clusters scale in different increments. For the purpose of comparative analysis, let's assume that the processing speed of the abstract element app processor is $r$ transactions per second and that any implementation element that contains an app processor has the processing speed $r$. Also, let's assume that the maximum number of sessions stored in the abstract element session store is $s$ and that any implementation element that contains a session store has the same upper bound on the number of sessions it can store. We also assume that session replication is configured to be done in pairs; two replicas per partition in OCCAS, two SIP containers per replication domain in WAS, and two replication buddies in MSS.

In OCCAS, an engine is CPU-intensive while a replica is memory-intensive. Scaling out an OCCAS cluster involves either scaling the engine to increase processing power or scaling the partition to increase session storage. The two limiting factors can be addressed separately. When the bottleneck is processing power, more engines can be added. Likewise, more partitions can be added when the bottleneck is the amount of memory needed to store active sessions. An extra engine adds $r$ transactions per second. An extra partition adds $s$ sessions storage. Adding a partition entails adding two replicas deployed in separate machines and racks.

Scaling out a WAS cluster involves adding more replication domains, each of which handles a non-overlapping subset of sessions. The total work load of the cluster is divided into $n$ replication domains. As stated in section 5.2, replication domains consist of SIP containers, that is, an app processor and a session store combined into a single process. Depending on the nature of the work load, a SIP container can be limited by processing power $r$ or memory space $s$. A replication domain processes up to $2r$ transactions per second since it contains two SIP containers. Session storage in a replication domain is still $s$ since sessions are shared equally by all SIP containers within the domain.

MSS is similar to WAS in that each Mobicents server contains both app processor and session store. Unlike WAS, a cluster can be incremented one Mobicents server at a time instead of one replication group at a time. This is because replication groups are not statically assigned: servers can dynamically select buddies. Each additional Mobicents server adds $r$ transactions per second of extra processing power and $s/2$ extra session storage, assuming that each server replicates with one buddy.

Overall, MSS has the smallest scaling granularity ($r$ and $s/2$), followed by OCCAS ($r$ or $s$), and then WAS ($2r$ and $s$).

### 6.2.2 *Operational cost of scaling*

Scaling out a cluster is not just a matter of adding more machines. It involves careful planning to ensure high availability.

In OCCAS, this process involves grouping replicas into partitions and placing them on separate racks or machines. Engines can be added with relative ease because they do not affect high availability as much as the replicas.

Mobicents servers are more convenient when scaling. As long as the servers run with identical configurations, new Mobicents servers will add themselves to the cluster automatically. However, MSS lacks an easy way for servers to be aware of racks. The JBoss Cache instances will pick buddies that are on different machines but they may be in the same rack. Support for configuration on a per-rack basis would make scaling easier.

WAS clusters need more administrator attention than OCCAS and MSS. Adding a new replication domain is simple but, as mentioned earlier, a failure in a WAS machine may result in 100% extra load on the remaining machine. The administrator may need to add more resources to machines to handle the extra load or may have to carefully place processes in machines so that the extra load does not exceed a certain amount. In either case, WAS needs more advance planning as the cluster scales up.

### 6.2.3 *Scaling efficiency*

In terms of processing power and memory efficiency, OCCAS has a more flexible architecture since CPU-intensive and memory-intensive elements scale separately. WAS and MSS are less efficient because adding an incremental unit means adding both processing power and memory space. This leads to either redundant CPU power or redundant memory space.

To compare network bandwidth usage for replication traffic, let's assume that a session replicator writes to two session stores. The write operation incurs session replication traffic to one SIP container in the same replication domain (WAS) or to one buddy JBoss Cache instance (MSS). In OCCAS, the write operation incurs replication traffic to two replicas in a partition, which is twice as much as others. JBoss Cache is further optimized to update a modified field within a session state instead of replicating the entire session state. Given the same network bandwidth for a cluster, MSS will scale most efficiently, followed by WAS and then OCCAS.

## 7. CONCLUSION AND FUTURE WORK

We have developed a framework for evaluating SIP server clusters. The framework includes system requirements and evaluation criteria that can be used to specify detailed requirements for different deployments. The framework also includes an abstract architecture defined at an appropriate abstraction level to help with understanding and comparing different designs and implementations. Each of the four abstract elements — the message dispatcher, the app processor, the session replicator, and the session store — has distinct and essential functions. In analyzing three SIP server cluster solutions, we have demonstrated the effectiveness of the abstract architecture. This paper shows that the abstract elements in each solution achieve their functions differently. For example, how session data is stored, replicated, and partitioned in the session store element is different in each solution. As well, this paper shows that each solution follows a different design in mapping these abstract elements to actual software components. Identifying these salient differences helps analyze and explain the high availability and scalability characteristics of the solutions.

While all three solutions examined can provide the highest Availability Level 4, the analysis has revealed significant differences in their characteristics and how well they meet the evaluation criteria. For example, the performance degradation upon a single machine failure, network bandwidth required for state replication, and scaling flexibility, are quite different in the solutions examined. It is hoped that this analysis will help practitioners select a suitable solution for their specific environments. It is also hoped that the evaluation framework will aid future analyses as well as motivate alternative designs.

We have not considered failure of an entire site in this paper. Disaster recovery based on geographical redundancy and replication will be examined in the future. Also as future work, we plan to conduct benchmarking to compare the performance of different designs under normal and failure conditions. We also plan to apply the evaluation framework to analyze highly available systems that utilize peer-to-peer networking.

## 8. REFERENCES

[1] Apache Software Foundation. Apache tomcat. `http://tomcat.apache.org/`.

[2] BEA. SIP servlet API version 1.1, 2008. Java Community Process JSR 289. http://jcp.org/en/jsr/detail?id=289.

[3] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: high availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 161–174, Berkeley, CA, USA, 2008. USENIX Association.

[4] E. M. Farr, R. E. Harper, L. F. Spainhower, and J. Xenidis. A case for high availability in a virtualized environment (HAVEN). In *The Third International Conference on Availability, Reliability and Security*, pages 675–682, 2008.

[5] IBM. *IBM WebSphere Application Server, Network Deployment, Version 7.0.* `http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/`.

[6] G. Kambourakis, D. Geneiatakis, S. Gritzalis, C. Lambrinoudakis, T. Dagiuklas, S. Ehlert, and J. Fiedler. High availability for SIP: Solutions and real-time measurement performance evaluation. *International Journal of Disaster Recovery and Business Continuity*, 2010.

[7] M. Michael, J. Moreira, D. Shiloach, and R. Wisniewski. Scale-up x scale-out: A case study using Nutch/Lucene. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1 –8, 2007.

[8] Oracle. *Oracle Communications Converged Application Server Configuration Guide Release 4.0*, 2008. `http://download.oracle.com/docs/cd/E13153_01/wlcp/wlss40/pdf/configwlss.pdf`.

[9] D. Patnaik, A. Bijlani, and V. K. Singh. Towards high-availability for IP telephony using virtual machines. In *IMSAA-10*, 2010.

[10] Red Hat Middleware. JBoss application server. `http://www.jboss.org/jbossas/`.

[11] Red Hat Middleware. Mobicents SIP servlets. `http://www.mobicents.org/products_sip_servlets.html`.

[12] D. Silas, J. Deruelle, V. Ralev, I. Ivanov, and J. Morgan. *Mobicents: SIP Servlets User Guide*, 2011. Release 4.0. Available at [11].

[13] K. Singh and H. Schulzrinne. Failover, load sharing and server architecture in SIP telephony. *Comput. Commun.*, 30:927–942, March 2007.

[14] M. Surtani, B. Stansberry, G. Zamarreño, and M. Markus. *JBoss Cache Users' Guide, Release 3.1.0 Cascabel*, 2009. `http://docs.jboss.org/jbosscache/3.1.0.CR1/userguide_en/pdf/userguide_en.pdf`.