Functional Specification
Converged Load Balancer

Author(s): pankaj.jairath@sun.com joel.xc.binnquist@ericsson.com

Version: 1.3

Change Log

Version	Comments	Date	Author
0.1	First cut of FS covering the overall scope.	07/03/07	Pankaj Jairath
0.2	Updated 2.4.1: Added information about ucr.xml, 2.4.2: modified description of SIP load balancing and 2.5.2: updated description of SIP container changes	07-08-21	Joel Binnquist
0.3	Included use cases and pseudo code	07-08-23	Joel Binnquist
0.4	Updated based on Ramesh P and Kshitiz S review comments	08/24/07	Pankaj Jairath
0.5	Updated based on changes due 0.2	08/27/07	Pankaj Jairath
1.0	Draft	08/27/07	Pankaj Jairath
1.1	Added Appendix section 8.3	09/10/07	Pankaj Jairath
1.2	Updated based on the review comments received.	10/01/2007	Pankaj Jairath
1.3	Updated with modifications done during development	5/13/2008	Joel Binnquist

1 Introduction

<List proposed feature(s). Introduce the basic vocabulary. Why is this interesting? List capabilities that may be normally expected, but are not being supported. Are there any limitations and caveats that need to be disclosed?>

The Converged Load Balancer (CLB) is responsible for forwarding SIP/SIPS//HTTP/HTTPS requests to a set of “SailFin instances” belonging to one or more clusters. The CLB uses proxy API [6] to forward requests to instances such that sticky requests for a particular session are delivered to the same instance that serviced the first request for that session. The CLB also supports SIP/HTTP session failover. What this implies is that the CLB has the ability to forward a SIP/SIPS/HTTP/HTTPS request to a healthy instance, if it detects that the target instance for a given request is unavailable.

It is delivered as a pluggable component into the SailFin request processing stack; on top of Grizzly proxy connector. While the Grizzly provides request forwarding, proxy'ing, across multiple instances, the CLB will provide high availability by supporting forwarding of requests to instances in a cluster, using health check mechanisms for determining the availability of instances and providing support for graceful termination (or quiescing) of instances to support online upgrade.

1.1 Reference Documents

Reference Document	Location (URL)
[1] SailFin Architectural Overview Document	http://wiki.glassfish.java.net/attach/FunctionalSpecsOnePagers/sailfin_umbrella.pdf
[2] SailFin Requirement Document	TDB
[3] DTD for converged-loadbalancer.xml	This Document
[4] Converged HTTP	http://wiki.glassfish.java.net/PageInfo.jsp?page=FunctionalSpecsOnePagers/ConvergedHttpSession_FunctionalSpec.doc
[5] SIP Servlet Container	URL not available
[6] Proxy functional specification	http://wiki.glassfish.java.net/PageInfo.jsp?page=FunctionalSpecsOnePagers/lb_proxy_fsd.doc
[7] Session Initiation Protocol	http://www.ietf.org/rfc/rfc3261.txt
[8] Domain DTD	https://sailfin.dev.java.net/documents/sun-domain_1_4.dtd
[9] TCP Protocol	http://www.ietf.org/rfc/rfc675.txt
[10] UDP Protocol	http://www.ietf.org/rfc/rfc768.txt
[11] Grizzly 1.0 API	https://grizzly.dev.java.net/nonav/apidocs/index.html
[12] Grizzly 1.5 API	URL not available
[13] Shoal / GMS API	https://shoal.dev.java.net/nonav/docs/api/
[14] Administration FS	http://wiki.glassfish.java.net/PageInfo.jsp?page=FunctionalSpecsOnePagers/sailfin_admin.doc

[15] HTTP 1.0/1.1	http://www.w3.org/Protocols/rfc2616/rfc2616.html
[16] Java SE 5.0 API java.util.regex	http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/package-summary.html

1.2 Glossary

In order to make concepts discussed easy to understand, this section provides a brief description of the terminology used in this document.

1 Converged Load Balancer (CLB)

Software load balancing component, that facilitates high availability of converged applications [17], by distributing the application requests in a cluster of SailFin instances. While distributing the incoming requests, it also provides for failing over sticky requests, owing to dynamic membership changes in the cluster. SIP and HTTP are the application protocols over which the requests are serviced.

2 Cluster

A group of SailFin instances, which have a homogeneous configuration. A cluster could be configured to act as both the load balancing and application serving tier or either a pure load balancing or pure application serving tier. The deployment topology defines the role of the cluster. The terms servers, instances are used interchangeably when referring to instances of the cluster, in this document.

3 Self Load Balancing Cluster

Deployment topolgy, in which each instance in the cluster serves as an CLB and an application instance. In a self load balancing cluster, each instance would load balance the incoming request or response to other instances in the cluster based on the load balancing policy. This document would commonly refer to such a deployment as a self-loadbalancing cluster.

4 Sticky Request

A request that belongs to a *Session.* The first / initial request serviced by the application usually results in establishing a *Session.* CLB would ensure that such requests are routed to the instance which hosts the session.

5 Proxy

Component *[6]* responsible of proxying the request to another SailFin instance other than the one which received the request. CLB would identify the server to which the request needs to be dispatched / load balanced to based upon its load balancing policy. In case the selected server is same the server on which the request is first received, it would be processed locally. This component is hosted within the connector.

In general, reference to documents mentioned under section 1.1 would be specified by using *[Number]* **notation.** This document might articulate concepts, mechanisms which have been

captured as part approach of defining CLB architecture and functionality; however would not be supported / implemented for the first release of SailFin. These sections would be colored code in RED .

1.3 Supported Features

The following features will be supported by the Converged LB in the first release of SailFin:

- Setup and configure the Converged Load Balancer.
- Forward SIP(S) and HTTP(S) requests to a cluster of SailFin instances.
- Load Balance requests using consistent hashing/round-robin policy, while maintaining stickiness.
- Enable and disable server instances in a cluster.
- Monitor health of server instances.

1.4 Features Not Supported

1.5 Differentiation

There exist no prior version of converged Load Balancer for SIP applications deployed on SaiFin cluster.

2 Design Overview

<Discuss the core concepts and design. Provide conceptual diagrams, if they would be helpful. Show how this sub-system/feature co-exists with other sub-systems. You may write 1-4 pages (can be shorter or longer). This section is should be a map to navigate well documented code!>

2.1 Principles of Operation

In the context of this document, it is very important for reviewers to read the SailFin architectural document *[1]*. This document defines the key high-level concepts for SailFin – the Communication Application Server. So it is recommended that reviewers read this document prior to reviewing this document.

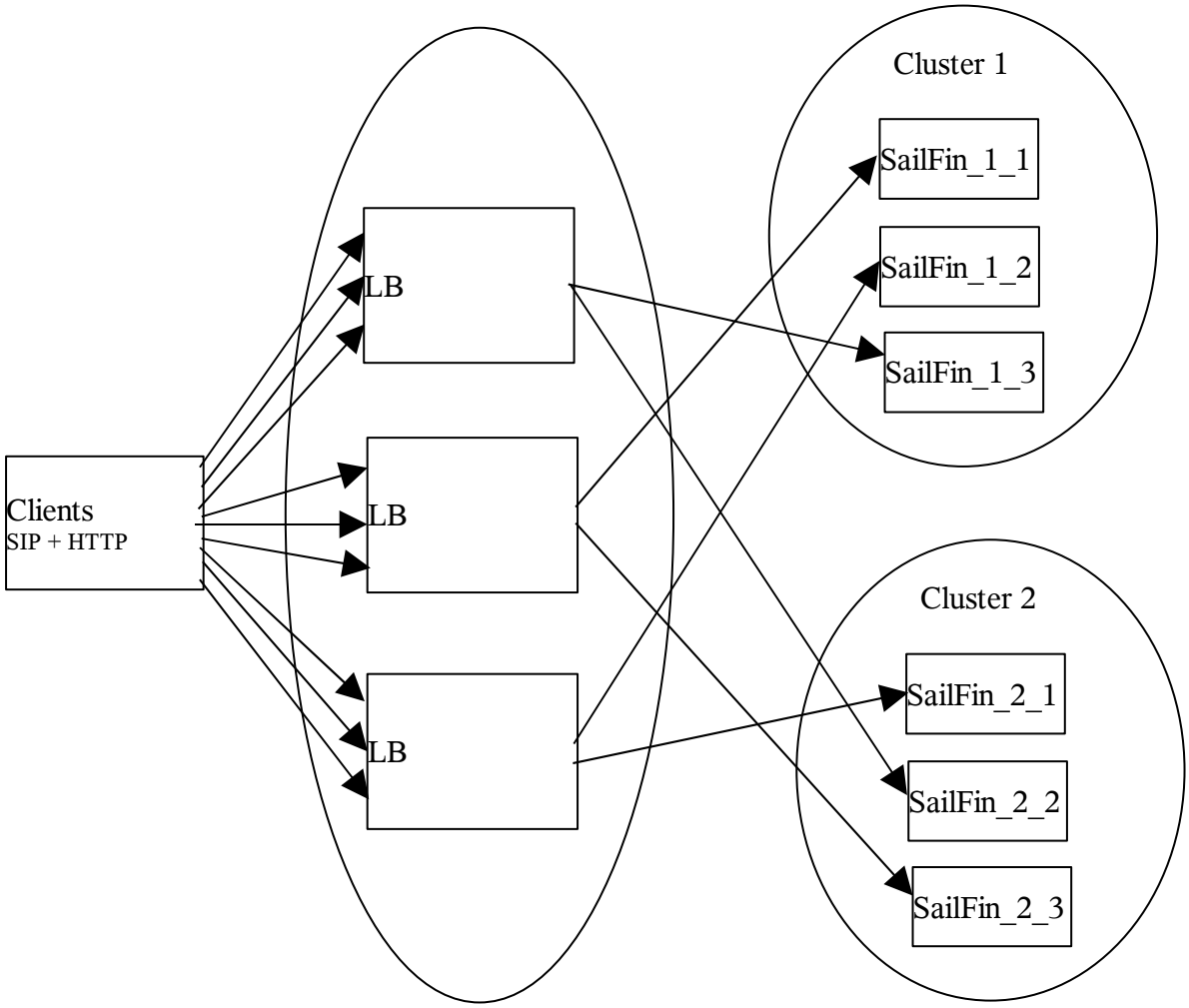


Figure 1. Functional Overview for the Converged LB

The various components that interact to deliver the load balancing feature are shown in Figure 1. They are:

1. Sip application clients over HTTP and SIP protocols
2. SailFin instances configured as converged Load Balancer
3. Cluster of SailFin instances which serve the requests

SailFin instance acting as the CLB receives the SIP clients requests. For each incoming HTTP request, the CLB matches the request URL with context roots of the applications for which the CLB is configured to do load balancing. In the case of SIP requests, there is no such check. Thereafter CLB identifies the list of instances that can service the request. The

CLB selects one server instance from this list based on its load balancing algorithm. Once an instance is selected, the underlaying proxy *[6]* forwards the request to that instance. In the event that the CLB detects that the selected instance for a sticky request is unhealthy, it selects a healthy instance from the same cluster and forwards the request to the selected instance.

These different components can be deployed in variety of ways depending on the customer's needs. A description of the typical deployment scenarios is provided in Section <TBD> of the SailFin Application Server Architectural Overview *[1]*. The typical deployment scenarios are illustrated to provide an understanding of the deployment issues at a broader level, rather than providing exhaustive list of possible deployments. Though CLB will be architectured and designed to support the two-tier deployment, where the first tier is the dedicated CLB, and the second tier being the pure application cluster(s), for SailFin release the single tier topolgy - self-loadbalancing cluster, would be supported.

2.2 Architectural Overview

This conceptual model of the CLB is presented below. The conceptual model identifies the important concepts that influence the CLB functionality. The concepts and their associations are depicted in an UML Collaboration Diagram in Figure 2. We would like to emphasize that the entities depicted in the diagram are “concepts” and not actual classes or components. The description of these concepts is presented below:

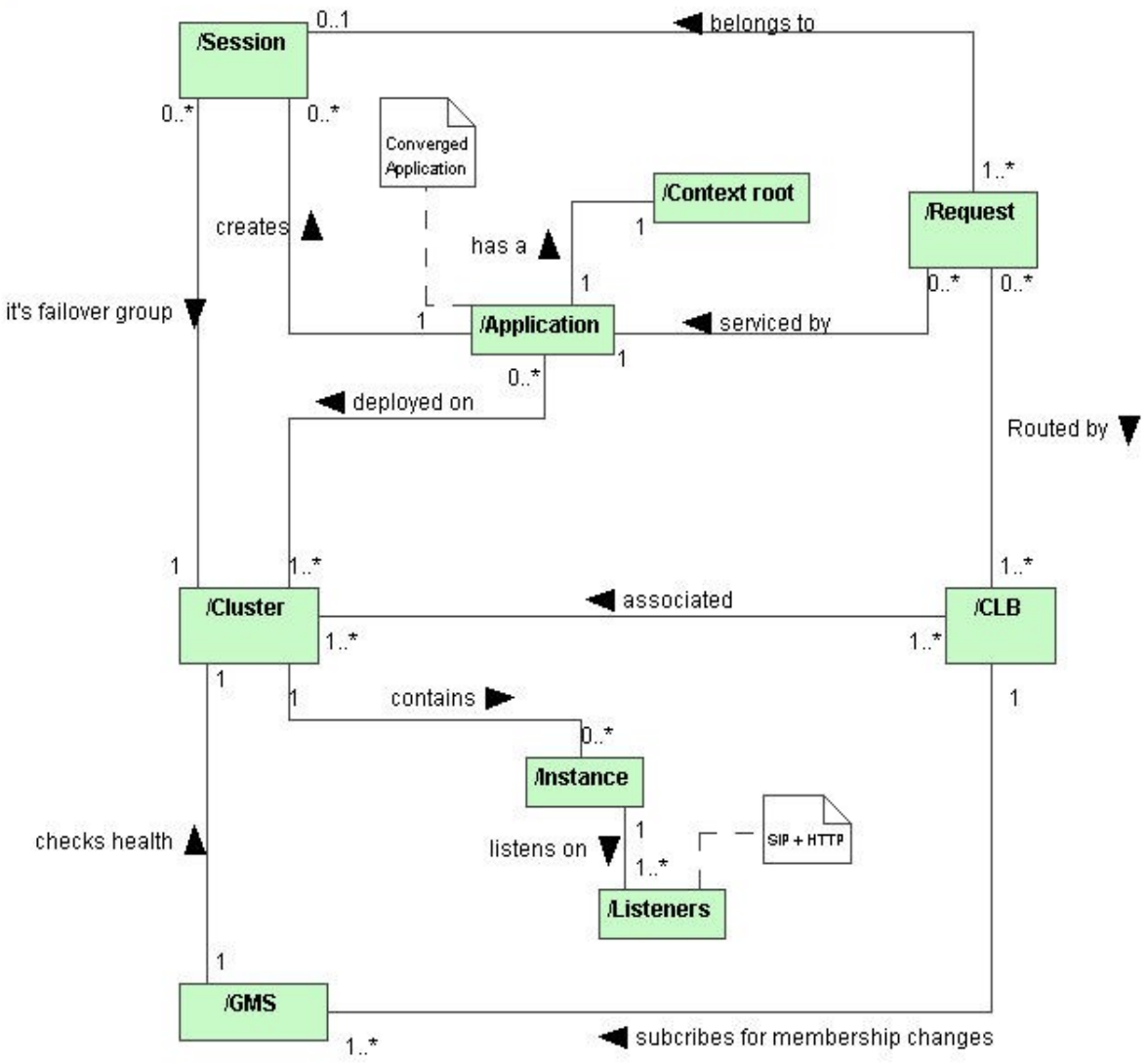


Figure 2. The Conceptual Overview for the Converged Load Balancer System

1 A SailFin C*luster* can be comprised of one or more *instance*s and one or more C*onverged Load Balancer*s. Instances receive requests on *Listeners*. A listener is specified as a combination of the I/P address of the machine on which the instance is running and the Port on which incoming requests are received. Each instance has one or more *Listener*s configured to receive incoming requests.

2 *Application*s are deployed on instances. One Application can be deployed on multiple clusters and a cluster can have multiple applications deployed. A cluster has homogeneous application deployment, i.e., any application deployed on a cluster is deployed on every instance of the cluster.

There is an exception to homogeneity of application, where in it would be possible to deploy an application such that it is disabled in only one instance and enabled in others.

3 Applications are serviced by SIP/SIPS/HTTP/HTTPS *Requests*. In the case of HTTP, each request can be identified by a URL Pattern, for example, /SailFin/TelcoApp/conference.jsp or /SailFin/TelcoApp/main.jsp. The common root of all URL Patterns for a particular application is called the *Context-Root* for that *Application*. For example, /SailFin would be the *Context-Root* for the URL Patterns described above. A SIP request does not have a *Context-Root* associated with it.

4 Each request may belong to a *Session*. A request that is attached to a session is called a **sticky request**. The request is forwarded by the CLB to a listener on a instance.

5 A particular CLB can forward requests to instances on multiple clusters and a single cluster can receive requests from multiple CLB's.

6 Each session is tied to a particular cluster. What this implies is that though an application can be deployed on multiple clusters, **sessions will only be failed over to instances within the same cluster**. In other words, no failover across clusters will be supported by the CLB. Failover occurs when an instance is not available. The *GMS* associated with the cluster determines whether the instances in the cluster are healthy or not.

2.3 Subsystem Overview

The interface overview diagram (Figure 3) depicts the interactions between the CLB and external subsystems. The various association lines are labeled with interface numbers that match the particular entries in the interface table described in Interfaces section. Association line pointing into a subsystem indicate an import into the subsystem whereas an association line pointing out indicate an export out of the subsystem. The dotted line between the subsystem highlights that the two are integral to each other. There are three main subsystems that are part of the CLB:

1 Grizzly - The transport layer and the components that are exported and imported from it are shown in light gray color. This subsystem is already implemented and will be evolved to support UDP as transport layer protocol.

2 Proxy - The reverse-proxy and the components that are exported and imported from it are shown in dark gray color. Grizzly would be evolved to support the proxying of requests to remote instances. This is a new deliverable for SailFin

3 Converged Load Balancer - The converged telco CLB and the components that are exported and imported from it are shown in white color. This is a new deliverable for SailFin.

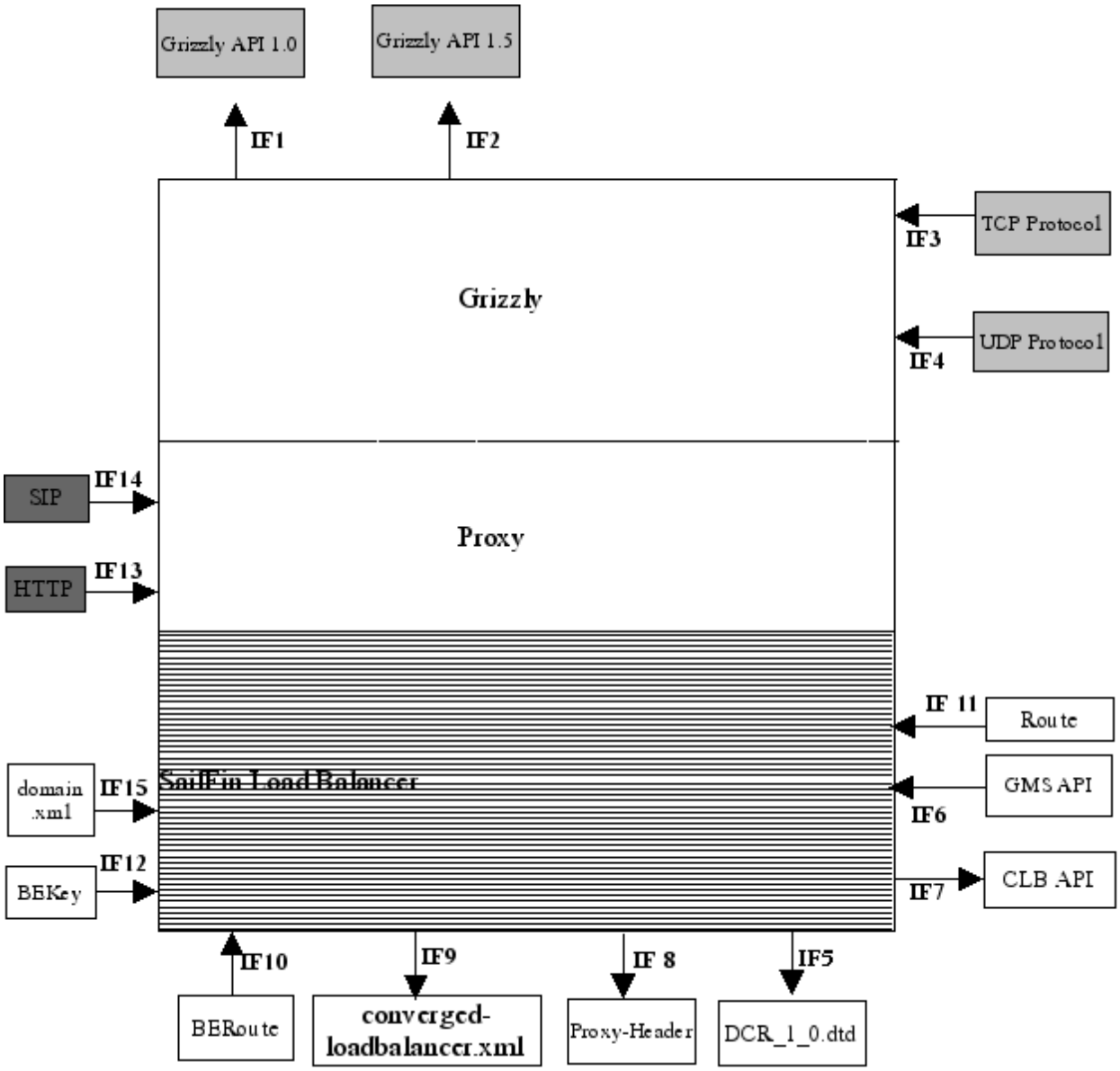


Figure 3. Interface Overview for the Converged LB

2.4 Converged LB Operational Details

The Actors that interact with the LB are:

- The Administrator: who configures and manages the SailFin system.
- The network clients, for SIP and HTTP: that sends requests to the SaiFin system.

In this section, we describe the operation details of the CLB that are triggered by the Actors. There are 2 principal phases in the operational life-cycle of the CLB:

1. CLB Configuration – In this phase, the administrator creates the CLB configuration information that is used in the Runtime phase.

IF11

2. CLB Runtime – The runtime phase is when the CLB receives SIP / HTTP requests and forwards it to the appropriate instance in the cluster based on the configuration (done in steps 2 and 3) and its internal load balancing algorithm.

The details of each of the operational phases are described below.

2.4.1 CLB Configuration

2.4.1.1 Configuration Process

The Administrator uses the administration infrastructure – the CLI / GUI – interface to configure SailFin cluster and set it up for Load Balancing. Subsequently, in the Runtime phase, the CLB loads the configuration information. The contents of XML file and the DTD are described in detail below but to set the context for the next section, we briefly describe the important settings that the Administrator makes in this phase:

1. The Administrator specifies the cluster that needs to be setup for load balancing.

2. While specifying a cluster configuration, the Administrator specifies whether the instances in the cluster can function as a Converged Load Balancer.

3. The Administrator specifies whether the cluster **load balances itself** or **load balances to backend application cluster.**

4. The Administrator specifies whether an instance is enabled or disabled. If the Administrator disables an instance, **while the cluster is operational,** it triggers the **quiescence operation** in the LB. Once an instance is disabled (quiescence period is over), the LB will not send any requests to this instance but instead send the requests to another instance in the cluster. However, **when** the instance is enabled, the LB would **resume sending** requests to the instance.

5. An Administrator specifies converged load balancing policy; whether a hash key shall be extracted from specific headers or if Data Centric Rules shall be used to extract the hash key from the incoming requests. In case Data Centric Rules shall be used the administrator specifies the location of the Data Centric Rule file.

2.4.1.2 CLB Configuration File

This file filters out the cluster deployment topology for which CLB has been configured. Domain administration framework [14] provides CLI and GUI interfaces to support automatic generation of this file. Section 4.1.4 of Admin Functional Specification *[14]* illustrates CLB specific changes to domain DTD.

CLB uses this file to configure its runtime to provide for high availability of the incoming requests and responses of the deployed converged applications.

2.4.1.2.1 sun-converged-loadbalancer_1_0.dtd

The sun-converged-loadbalancer_1_0.dtd file describes the DTD for converged-loadbalancer.xml. The supporting domain/server DTD changes are described in [14]. A hierarchical view of CLB DTD is shown below in Figure 5.

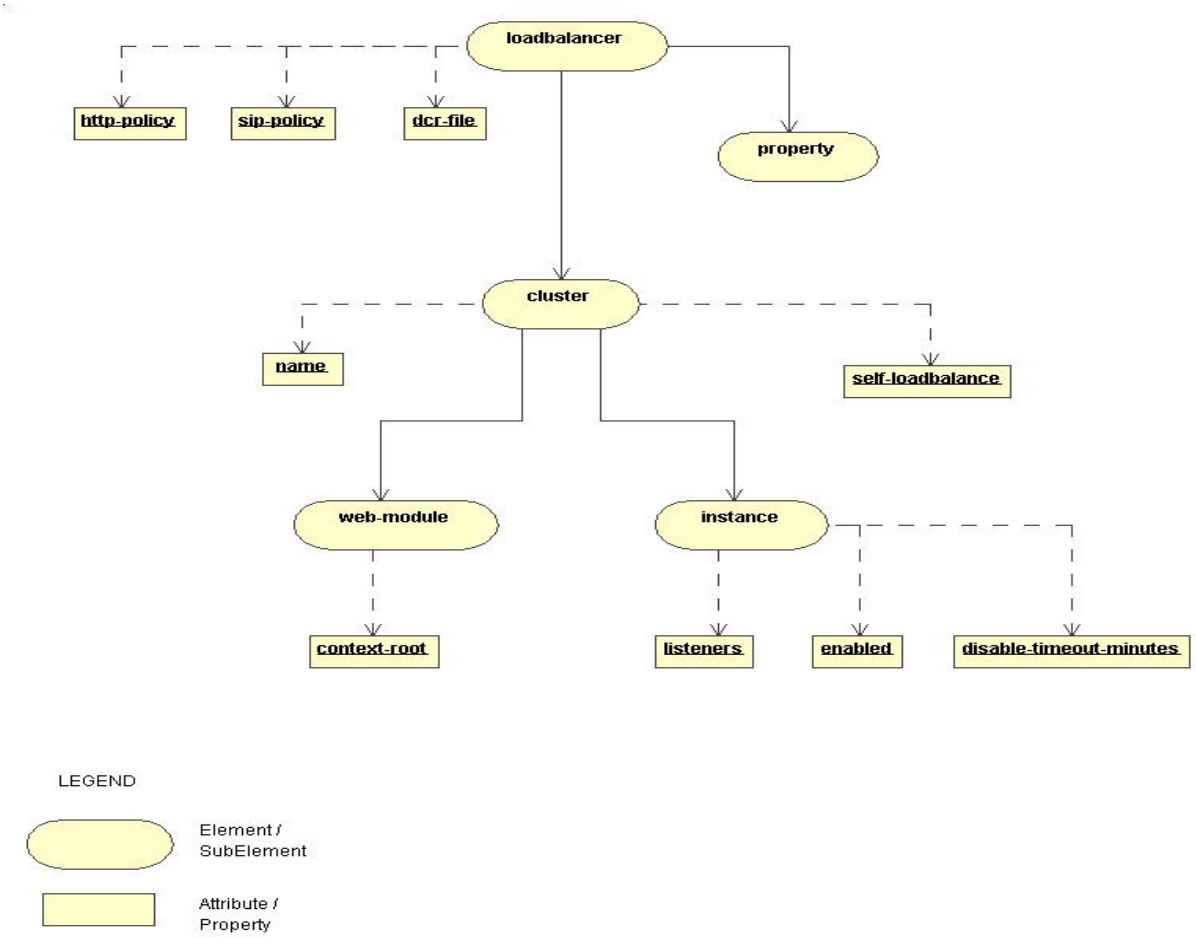


Figure 5. Hierarchical View of sun-converged-loadbalancer_1_0.dtd

A detailed description of for each element/subelement in the DTD is provided below.

loadbalancer

Defines a CLB configuration. This is the root element; there can only be one loadbalancer element in a converged-loadbalancer.xml file.

Attributes

The following table describes attributes of the loadbalancer element.

Attribute	*Default*	*Description*
http-policy	IMPLED having default value of "round-robin"	Specifies the load balancing policy used for the http requests. The default implied value is round-robin.
sip-policy	IMPLIED having default value of	Specifies the parameters on which consistent hashing policy is applied to obtain the hash key.

	“From-tag,To-tag,Call-id”	This can be specified as comma separated values of parameter names to hash on.
dcr-file	"" Empty string	The data centric rules file name. In case this is specified, http-policy and sip-policy would be overridden.

Subelements

The following table describes subelements for the loadbalancer element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

loadbalancer subelements

Element	*Required*	*Description*
cluster	zero or more	Defines the cluster(s) that the CLB caters to.
property	zero or more	CLB specific configuration properties.

cluster

Defines the clusters associated with a particular CLB. This is a subelement of loadbalancer.

Attributes

The following table describes attributes for the cluster element.

cluster attributes

Attribute	*Default*	*Description*
name	none	Required attribute that defines the name of the cluster.
self-loadbalance	Default value of “true”	Specifies whether configured cluster self load balances incoming requests to itself. If its configured to do so, load balancer is an intrinsic component of the participating server instances in the cluster.

Subelements

The following table describes subelements for the cluster element. This is a subelement of loadbalancer.

cluster subelements

Element	*Required*	*Description*
instance	zero or more	Describes the instances in the cluster.
web-module	zero or more	Defines the attributes for the web modules (applications) that are serviced by the LB.

instance

Defines the instances associated with the cluster. This is a subelement of cluster.

Attributes

The following table describes attributes for the instance element.

instance attributes

Attribute	*Default*	*Description*
name	None	Required attribute that defines the name of the instance.
enabled	“true”	Indicates if the instance is enabled or disabled.
disable-timeout-in-minutes	31 minutes	Specifies the *quiescing* time out interval in minutes after which the load balancer no further requests would be forwarded to the instance.
listeners	None	Required attribute that specifies the SIP and HTTP listeners for the instance. This attribute can be used to specify multiple listeners for a instance delimited with a space. For example, "sip://server1:5060 http://server1:8080”

Subelements - None

web-module

Defines the web-module. This is a subelement of cluster.

Attributes

The following table describes attributes for the web-module element.

web-module attributes

Attribute	*Default*	*Description*
context-root	None	Required attribute that defines the the context-root URI for the web module. Web modules are accessed via URIs, for example /SailFin/TelcoApp/conference.jsp or /SailFin/TelcoApp/main.jsp The pattern common to all URIs for an application is called its context-root. For example, /SailFin would be the context root for the URIs described above.

Subelements - None

property

We have decided to use the following properties for the loadbalancer element.

Property	*Default*	*Description*
reload-poll-interval-in-seconds	Default value of "0" which means that no polling will be done	Time interval in seconds at which load balancer would detect if convergerd-loadbalancer.xml timestamp has changed. If it has changed, the CLB would reload it.

2.4.1.2.2 Location of converged-loadbalancer.xml

The default location of converged-loadbalancer.xml is the config directory of the instance.

2.4.1.3 Data Centric Rule (DCR)

The selection of the server instance to forward the request, is based on a hash key. The key is extracted from incoming SIP and HTTP requests according to suitable rules.

All applications share the same rules defined in a common rule file. The rules can be changed during operation. The Data Centric Rule language is defined in XML and similar to the triggering language for mapping requests to servlets [7]. The rule language consists of conditions and variables supporting SIP and HTTP key extraction. In contrast to the servlet mapping language DCR needs to return a value.

Each incoming SIP and HTTP request is matched with the current rule set.The first rule that matches is used for key extraction.

2.4.1.3.1 Data Centric Rule DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- DTD for Data Centric Rules, DCR -->
<!ELEMENT user-centric-rules (sip-rules,http-rules)>

<!ELEMENT sip-rules (operator-condition)*>
<!ELEMENT http-rules (operator-condition)*>
<!ELEMENT operator-condition (operator|condition)>
<!ELEMENT operator (or|and|if)>
<!ELEMENT condition  (header|request-uri|session-case|cookie) >

<!ELEMENT or (operator-condition)*>
<!ELEMENT and (operator-condition)*>
<!ELEMENT if (operator-condition,else?)*>

<!ELEMENT else EMPTY>
```

```
<!ATTLIST else return NMTOKEN #REQUIRED>

<!ELEMENT header (exist|notexist)>
<!ATTLIST header
  name NMTOKEN #REQUIRED
  return NMTOKEN #REQUIRED >

<!ELEMENT request-uri (exist|notexist|match)>
<!ATTLIST request-uri
  parameter NMTOKEN #IMPLIED
  return NMTOKEN #IMPLIED >

<!ELEMENT session-case (equal,session-case-type)>
<!ELEMENT session-case-type  (INTERNAL|EXTERNAL|ORIGINATING|TERMINATING|
                              TERMINATING_UNREGISTERED) >

<!ELEMENT cookie (exist|notexist)>
<!ATTLIST cookie
      name NMTOKEN #REQUIRED
      return NMTOKEN #REQUIRED>

<!ELEMENT equal (#PCDATA)>
<!ELEMENT exist EMPTY>
<!ELEMENT notexist EMPTY>
<!ELEMENT match (#PCDATA)>
```

2.4.1.3.2 Location of the Data Centric Rule File

The DCR file location is the config directory of the server instance.

2.4.2 CLB Runtime

The runtime phase comes into play when the CLB receives the SIP and or HTTP client requests. For the case of SIP, CLB can also receive SIP responses. The CLB has a load balancing algorithm that is used to decide to which instance the request has to be forwarded. The CLB supports the following runtime functionality These are:

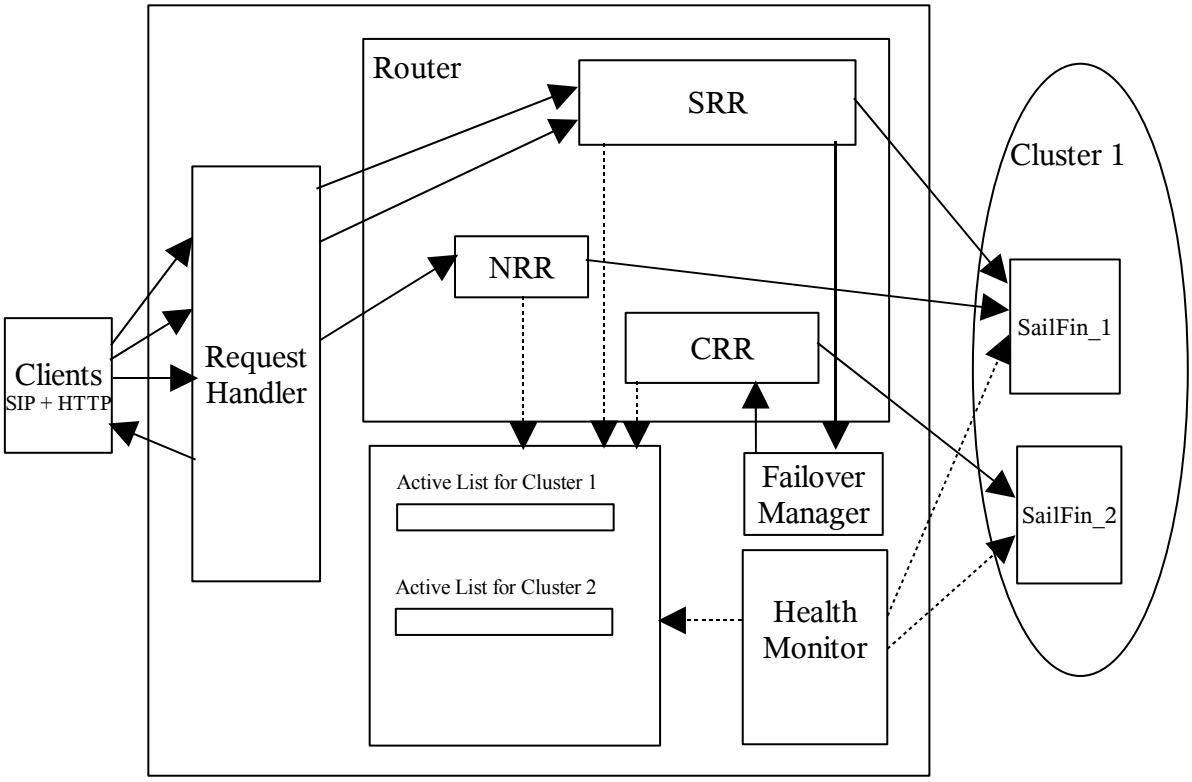
1. Request matching for HTTP/SIP requests
2. Forwarding of new requests
3. Forwarding of subsequent requests
4. Monitors the health of the server instances
5. Performing dynamic reconfiguration.
6. Logging of Messages.
7. Handling server/instance quiescing – online upgrade

The following sections, where necessary for highlighting the difference between handling of HTTP and SIP request would categorize them.

Figure 6 below presents a composite, for both SIP and HTTP request processing, functional view of the CLB Runtime phase. Please note that the internal subsystems of the CLB illustrated in this figure are purely with the intent of explaining the functional aspects of the CLB Runtime and not so much to prescribe the design. For HTTP and SIP requests the illustrated subsystems would be work differently at lower levels of their respective request processing stack.

2.4.2.1 Deciding if CLB should do the processing

When the ***"converged-loadbalancer"*** element is passed in the *availability-service,*[8] , it would imply that the server is being configured to run with CLB. While specifying this it would pass an attribute "***config-file"***; which specifies the location of CLB configuration file converged-loadbalancer.xml.



Conventions and Abbreviations

NRR – New Request Router SRR – Sticky Request Router CRR – Cluster Request

Requests and Responses

Interactions between components

The Converged Load Balancer internally has two components :-

1 HTTP component

This caters to load balancing HTTP / HTTPS requests based on the feature set mentioned in Suppotrted Features section.

2 SIP component

This caters to load balancing the SIP / SIPS requests based on the feature set mentioned in Supported Features section.

2.4.2.2 Request matching

2.4.2.2.1 HTTP Requests

The following activities happen in this step:

The CLB Request Handler gets request from the Grizzly connector *[6]* and it tries to match the request URL with context roots of the deployed applications.

1 If a match is not found, then it detects this as a non-CLB request. In this case the Request Handler returns control back to the connector.

2 If a match is found, the Request Handler then ascertains if the request is a new request or a sticky request. The load balancer determines a request is sticky by looking for routing information stored in either the cookie or the URL. If the information is found, then it is identified as a sticky request, else as a new request. If it is a new request, then the Request Handler delivers this request to the New Request Router. If it is a sticky request, then the Request Handler delivers this request to the Sticky Request Router.

2.4.2.2.2 SIP Requests

For SIP requests there is no way to identify for which application they relate to. In this case following activities happen in this step:

1 Request Handler ascertains if the request is a new request or a sticky request. The load balancer determines a request is sticky by looking for routing information stored in either SIP header ***"Route"*** or in the SIP ***URI*** of the request-line. If the information is found, then it's identified as a sticky request; else it is identified as a new request. If it is a new request, then the Request Handler delivers this request to the New Request Router. If it is a sticky request, then the Request Handler delivers this request to the Sticky Request Router

2.4.2.2.3 SIP Responses

For SIP responses the following activities happen in this step:

1 Request Handler always forward responses to the Sticky Request Router

2.4.2.3 Forwarding a New Request

2.4.2.3.1 HTTP Request

Depending on configuration the New Request Router either uses consistent hash or Round Robin to select an instance to serve the request.

Round Robin

The New Request Router delivers the request to the next healthy instance from the list of all instances belonging to all clusters that are serviced by the LB for a given application.

Consistent Hash

The New Request Router, extracts a hash key from the request based on the rules specified in the Data Centric Rules file. The hash key is used to look up an instance using a consistent hash function, which maps the hash key to one of the instances from all clusters that are serviced by the LB. The New Request Router delivers the request to the selected instance.

Determination of an healthy instance is made by excluding all instances that have been marked as unhealthy. Instances that are disabled are also considered "unhealthy" and hence are unavailable for selection. While actually forwarding the request, if the Router detects that the instance has turned unhealthy, the Router picks the next healthy instance.

2.4.2.3.2 SIP Request

For SIP requests received by LB, there is no way for it to determine to which application deployed for the cluster(s) it belongs to. The New Request Router, extracts a hash key from the request based on the rules specified in the Data Centric Rules file. The hash key is used to look up an instance using a consistent hash function, which maps the hash key to one of the instances belonging to all clusters that are serviced by the LB. The New Request Router delivers the request to the selected instance.

Determination of the healthy instances is made by excluding all instances that have been marked as unhealthy. Instances that are marked "disabled" are also considered unhealthy and hence are unavailable for selection.

2.4.2.4 Forwarding a Sticky Request/Response

The mechanism used to determine sticky attribute of a received request differs between HTTP and SIP requests. The following section details the approach taken for each of these application protocols.

In general the following activities occur during this step :

1. The Sticky Request Router, retrieves the sticky information. From the sticky information, it first determines the instance to which the request was previously forwarded. The sticky information can either be the hash key used for the initial request ***"BEKey"*** or the identity of the serving instance ***"BERoute"***.
2. It checks if the instance to which the request is sticky is marked as healthy. If the instance is healthy, it forwards the request to the specific instance.
3. If the instance is unhealthy, then Sticky Request Router delegates the request to the Failover Manager.

2.4.2.4.1 Strategy for Detecting HTTP Session Stickiness

Following two mechanism are used to determine the stickiness in this case :

- HTTP Cookie

In this approach, HTTP request headers are checked for presence of a Cookie added by the serving SailFin instance. This HTTP Cookie, ***"BERoute"*** or ***"BEKey"*** (depending on whether round robin or consistent hash policy is used) contains an encoded value that LB uses to determine the instance which has serviced the established session.

- HTTP URL

  This approach works even if the browser does not support cookies. Here the sticky information, HTTP Cookie, presence is checked in the HTTP request URL

2.4.2.4.2 Strategy for Detecting SIP Session Stickiness

Following mechanisms are used to determine the stickiness in this case :

SIP requests

- SIP Header based (Route)

  In this approach; SIP request header, ***Route,*** is checked for presence of a tracking parameter added by the serving SailFin instance. This parameter is identified with the name "***BEKey"*** and the its value is used by LB to determine the instance to forward the request to.

- SIP URI

  In this case, the SIP URI of the request line is checked for presence of "***BEKey"*** parameter added by the serving SailFin instance. LB uses the value encoded in this to determine the instance to forward the request to.

SIP responses

- SIP Via header

  For SIP responses the topmost ***Via***, is checked for presence of a tracking parameter added by the serving SailFin instance. This parameter is identified with the name "***BERoute"*** and the its value is used by LB to determine the instance to forward the request to.

2.4.2.5 Failing over sticky requests

As the dynamic membership of cluster changes either due to controlled failure or uncontrolled failure of participating instances, LB would need to fail over requests to established sessions to another instance in the cluster. Online upgrade and erratic loss of instance are examples of controlled and uncontrolled failures.

Following sections details how fail-over of sticky HTTP and SIP requests are handled.

2.4.2.5.1 Fail over of HTTP Request

The following occurs in this step:

1. The Failover Manager identifies the cluster to which the failed instance belongs. For each cluster, there is a Cluster Request Router that handles the routing of

failed over sticky requests for that cluster. The Failover Manager delegates the request to the Cluster Request Router.

2 The Cluster Request Router selects the next healthy instance from its list of instances belonging to the same cluster and forwards the request to that instance. For selecting the next healthy instance, the Cluster Request Router either applies the Round Robin or the Consistent hash policy, depending on configuration. In case the Consistent Hash policy is applied the hash key is retrieved from the ***"BEKey"*** cookie.

3 If there is no healthy instance found in the cluster, the Failover Manager would finally delegate the request to New Request Router to select a healthy instance. The New Request Router would select the next healthy instance from its list of instances belonging to all the participating clusters over which the application is deployed. If there exist more than one cluster, this would result in fail-over of the request to another cluster. However, since sessions are not replicated between clusters, any existing sessions are lost.

2.4.2.5.2 Failover of SIP Request

1 The following occurs in this step :

2 The Failover Manager identifies the cluster to which the instance belongs. For each cluster there is a Cluster Request Router that handles the routing of failed over sticky SIP requests for that cluster. The Failover Manager delegates requests Cluster Request Router.

3 The Cluster Request Router selects the next healthy instance from its list of instances belonging to the same cluster and forwards the request to that instance. While selecting the next healthy instance, the router would re-apply *consistent hashing policy on the SIP request*, using the value in ***"BEKey"***, over the new set of instances available.

4 If there is no healthy instance found in the cluster, the Failover Manager would finally delegate the request to New Request Router to select a healthy instance. If there exist more than one cluster in the deployment of converged applications, this would result in failover of the request to another cluster. However, since sessions are not replicated between clusters, any existing sessions are lost.

2.4.2.6 Monitoring the Health of Instances in a Cluster

CLB uses the Shoal / GMS APIs *[13]* to monitor the health of instances in a cluster. Following two approaches are used :

- Spectator to Remote Application Deployment Cluster

  In this case, CLB would use the cluster name to attach itself as a Spectator to events occurring in the GMS group identified by the cluster name. These events relate to loss and joining of instances. CLB would instantiate the peer GMSService in the local VM via call to GMSFactory.startGMSModule(remoteClusterName).

- Join as Core member, for same instance deployment of CLB and Application

In this case, CLB is part of a self load balancing cluster; and it would use the GMS client API's to source the GMS events occurring within it's cluster that would signal the loss or joining of instances. In this case GMSLifeCycle would take care of instantiating the GMSService and CLB would simply get the reference to the service started in the local VM via GMSFactory.getGMSModule(clusterName).

Events that CLB needs to source from GMS -

1. Failure

   When an instance is not accessible. The turnaround time to its detect should be least over a given system, hardware configuration. CLB would internally mark the instance as unhealthy and would no longer be used for active load balancing; till it joins back the cluster.

2. Join

   When an instance joins the cluster. This should signal to CLB that an instance is available to service the requests. There are two possibilities for such an event -

   - GMS dispatches the event upon server start-up; rather than early in the start-up cycle.
   - CLB upon receiving the signal, can send a dummy HTTP request to server such that a successful response would imply instance ready to service the request. Consequently CLB would internally mark the instance as healthy and available for load balancing.

2.4.2.7 Instance Quiescing – Online Upgrade

Instance quiescing is activated by setting the “lb-enabled” attribute of the server to false and setting an appropriate value to the “disable-timeout” *[8]*. The time out represents the elapse time after which the CLB will disable that particular instance. In this case, the CLB uses the following policy for quiescing:

1 If an instance is disabled, and the time out has not expired, then any sticky requests will be still delivered to the instances. However, any new requests will not be sent to the instance and will be diverted to another healthy instance in the cluster.

2 Once the time out has expired, the instance is disabled. The CLB will not send any requests to this instance, instead CLB will fail-over sticky requests to another healthy instance in the same cluster.

An instance that has been quiesced to it's disable timeout period, would not be able to take part in load balancing, till it is "enabled" again by marking the "lb-enabled" attribute of the server to "true". Session replication functional specification *[4]* would articulate details on how it uses this feature to carry out the rolling upgrade of the servers in a SailFin cluster.

2.5 Supporting Container Changes

Marking a request as sticky would be taken care by the Web and SIP container for HTTP and SIP request respectively.

2.5.1 Web Container

If the incoming HTTP request results in session establishment, then the container would added the sticky information, depending on whether the Round Robin or Consistent Hash policy is applied:

Round Robin

"***BERoute",*** either as an HTTP cookie or in the request URL

In the case of request failover, the container would re-stamp the request with ***"BERoute"***, with the value indicating it as the serving instance.

Details pertaining to this functionality are described in [4].

Consistent Hash

"***BEKey",*** either as an HTTP cookie or in the request URL.

2.5.2 SIP Container

If an incoming SIP request results in SIP session establishment where the container is the callee, the SIP container will stamp the ***Contact*** header in the response with ***"BEKey"*** parameter. In this way it is ensured that the ***"BEKey"*** is included in the request-line of subsequent requests sent by the caller.

If an incoming SIP request is proxied by the SIP container will stamp the ***Record-Route*** header in the response with "***BEKey"*** parameter. In this way it is ensured that the ***"BEKey"*** is stored in the endpoints and that is included in the ***Route*** header of subsequent requests sent by the caller.

SIP Header Example: Record-Route: <sip:sailfin-server1.biloxi.com;BEKey=encoded-value;lr>

SIP URI Example: sip:alice@atlana.com;BEKey=encoded-value

In outgoing requests the topmost **Via** is stamped with the ***"BERoute"*** parameter. This ensures that the response is returned to the instance where the transaction exists. In case that instance has gone off-line the response is dropped (the transaction is lost anyway). If the outgoing request is initiating a dialog (e.g an initial INVITE), the ***Contact*** header in the response is stamped with ***"BEKey"*** parameter. In this way it is ensured that the ***"BEKey"*** is included in the request-line of subsequent requests sent by the caller.

Details pertaining to this functionality are described in [5] and [16].

2.5.3 BERoute Sticky Information

The route information saved in ***"BERoute"*** is derived using the host/port information that is part of each listener associated with an instance. The following is the process by which the "***BERoute"*** cookie is generated:

1 Take the host/port information associated with every listener.

2 Use a one-way hash function to obfuscate it.

3 Convert the obfuscated information to a 32 bit integration.

4 Encode it into 4 bytes using base-64 encoding.

2.5.4 BEKey Sticky Information

The **"BEKey"** contains the hash key that was used to route the first incoming request of a dialog, or in case the first request was sent from the container, from that message. The hash key is extracted from the message using a function called Data Centric Rules. This function parses the message and extracts the hash key from the message according to configurable rules set up in an XML-file.

2.6 Interfaces

This section documents the interfaces for the LB. When reading the interface table, the "*Specified in Document*" column refers to document links specified under Reference Documents.

2.6.1 Exported Programmatic Interfaces

Interface Name	Specified in Document	Comments
IF7: CLB API	[6]	The API provided by the connector over which CLB is invoked passing the request and response artifacts.
IF8: proxy-* (Proxy Headers)	This Document	Protocol headers added to a passthrough / proxied request.

2.6.2 Exported Configuration files Interface

Interface Name	Specified in Document	Comments
IF9: converged-loadbalancer_0_1.dtd	[14]	DTD; it contains the CLB runtime configuration information.
IF5 : data-centric-rule_1_0.dtd	This Document	Data centric rules

2.6.3 Exported Sticky / Routing Interfaces

Interface Name	Specified in Document	Comments
IF10: BERoute	This Document	Sticky information used by the LB to save routing information.
IF15: BEKey	This Document	Used by CLB to store the consistent Hashkey value. This is used by CLB as part of consistent hashing policy to route the converged application requests to same backend based. The value is obtained by applying consistent hashing policy based on Data Centric rules.

2.6.4 Imported SIP Header Interfaces

Interface Name	Specified in Document	Comments
IF11: Route	[7]	SIP Request header used by CLB to determine the route information parameter – BEKey
IF16: Via	[7]	SIP Request header used by CLB to determine the route information parameter - BERoute

2.6.5 Imported Programmatic Interfaces

Interface Name	Specified in Document	Comments
IF1 : Grizzly 1.0	[11]	API used by the Proxy for handling HTTP requests
IF2 : Grizzly 1.5	[12]	API used by the Proxy for handling SIP requests
IF6 : GMS	[13]	Shoal APIs used by LB to monitor the health of instances in a cluster.

2.6.6 Imported Configuration files Interface

Interface Name	Specified in Document	Comments
IF12 : sun-domain_1_4.dtd	[8]	Domain.xml specifies the domain/server configuration

2.6.7 Imported Protocols

Interface Name	Specified in What Document?	Comments
IF13 : HTTP 1.0/1.1	[15]	HTTP Protocol
IF14 : SIP 1.0	[7]	SIP Protocol
IF3 : TCP Protocol	[9]	Grizzly supports TCP as transport protocol
IF4 : UDP	[10]	Grizzly supports UDP as the transport layer protocol.

3 Quality and Availability

<How do you handle availability concerns? Does this feature introduce any new failure modes? List testing and failure scenarios that quality team needs to worry about.>

3.1 Availability

In general CLB facilitates increased availability of the deployed applications in a cluster. This is achieved by load balancing and failing over requests upon dynamic cluster membership changes.

SailFin cluster can act as either a self load balancing cluster or be a dedicated LB cluster fronting an application serving cluster. In either case more the number of participating instances greater is the Converged LB availability.

SailFin cluster would provide support for SNMP traps, which can be sourced by a fronting network entity – IP sprayer; to detect the membership changes in the load balancing layer of the deployment topology.

3.2 Quality

The operational scenarios mentioned in section 2.4 need to be covered for testing.

4 Performance

<How do you want performance team to measure this sub-system? Any micro benchmarks necessary?Any goals? Anticipated scalability limits or goals?>

4.1 Vertical Scalability

SailFin instance configured to act as CLB, would limit the vertical scalability of the instance with its various subsystems providing the high availability of the overall system deployment.

4.2 Horizontal Scalability

If SailFin deployment tier acting as the CLB hits vertical scalability, it can be overcome by adding more instances in this tier to increase the horizontal scalability.

5 Management and Monitoring

<Describe how performance, management status, and diagnostic information is exposed. How does this feature handle dynamic configuration changes?>

5.1 Formal Interfaces

<How is this feature(s) configured by administrator? Does it introduce new commands or modify existing ones? Show syntax of expected administrative commands and response codes. What is the schema/syntax for new configuration in domain.xml? Show the DTD snippets later in this section. What are their default values? What are the validation rules? Think about the stability level for each of the above. Are you expecting that the proposed design will change?]

When a SailFin instance starts ups with CLB, the load balancer uses converged-loadbalancer.xml file to configure itself. Section 2.4.1 provides details pertaining to the CLB's configuration file. The DAS based administration functionality is covered by [14]

5.2 Dynamic Reconfiguration

CLB would source the Admin config change event from DAS whenever converged-loadbalancer.xml or dcr.xml gets updated. It would implement listeners for such event such that upon receiving the event, CLB would try pulling the updated converged-loadbalancer.xml / dcr.xml from the DAS and reconfigure itself using the updated configuration file.

Synchronization framework of DAS would take care of synchronizing the local CLB configuration of an instance as part of server start-up rendezvous with DAS. This would take of updating the CLB configuration on an instance, in case that instance was offline when an update took place.

CLB will poll the converged-loadbalancer.xml at an Administrator specified time interval to check if the file has been updated. If it detects that the file has been updated, it will

reload the file. The *reload-poll-interval* property of the converged-loadabalancer.xml specifies this poll interval value.

Also it would be possible to send the xml configuration file over an HTTPS request to CLB from DAS. Following activities occur in such a scenario -

1. If CLB, during instance start-up could not locate its configuration file, it would create a new file and initialize itself.
2. If CLB was already configured during instance start-up from its configuration file, it would over-write the existing file. Before doing so it would ensure that the contents are successfully parsed. This path of invocation would override, the *reload-poll-interval*, such that:-
    - If *reload-poll-interval* was set to zero, it would still cause CLB to reconfigure.
    - If *reload-poll-interval* was set to non-zero, it would cause CLB to reconfigure immediately upon receipt of this request and mask the poll for the subsequent poll check.

6 Packaging, Files, and Location

<Does this feature add new jar files or extend existing ones? Where are they located?>

Converged LB would be bundled into a SailFin HA *jar* distributable. This *jar* would be available, along with rest of SailFin jars, under the *lib* directory of the installation root.

7 Documentation Requirements

<List the required documentation to support this product feature.>

Following lists down the documentation need of this feature :

1. High Availability administration guide
   This scope out the steps required to configure the Converged Load Balancer. It should also describe the CLB configuration file
2. Domain DTD Changes
   The configuration elements catering to CLB need to be described.
3. Online Help
   The supporting GUI constructs need the contextual help.

8 Appendix

8.1 Use Cases

The following use cases illustrates various scenarios when load balancing is performed by all members in the cluster (each instance acts both as FE and BE). It is assumed that an IP sprayer is placed in front of the cluster. The IP sprayer is a simple device with no intelligence that just spreads traffic over the AS instances using some simple algorithm (e.g, round-robin).

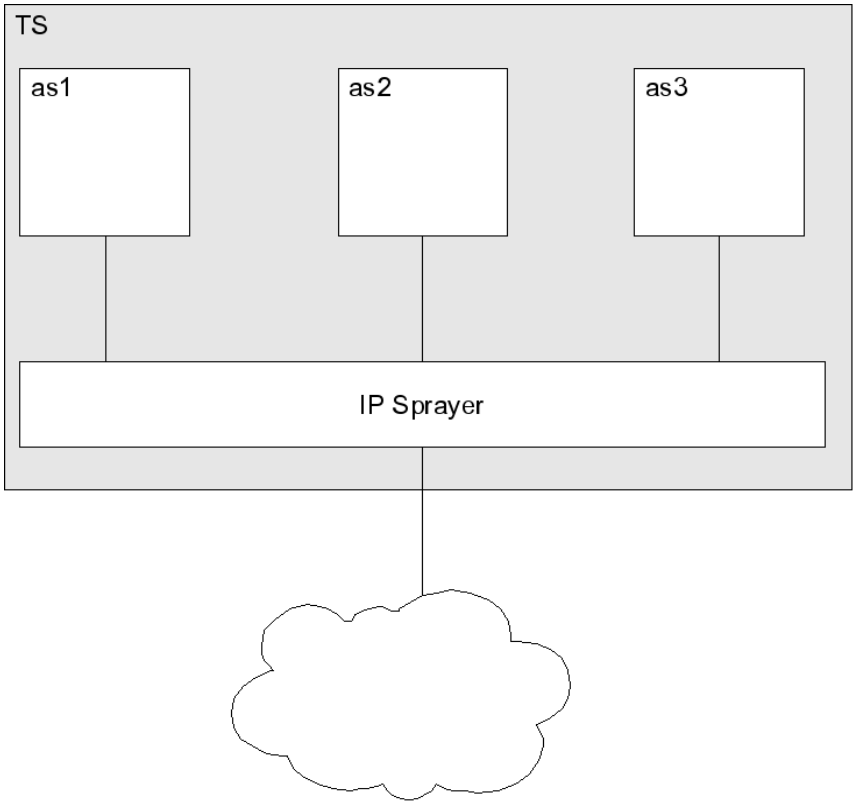


Figure 1: Example Cluster

Figure 1 Illustrates the cluster used in the example. In the example the cluster consists of three AS instances: AS1..AS2 (with the IP addresses *as1*, *as2* and *as3*, respectively). Moreover, there is an IP sprayer in front of the cluster which exposes the public interface of the TS and it has the public host name and port *ts:5060*.

8.1.1 Typical SIP Session; External Party is the Initiator

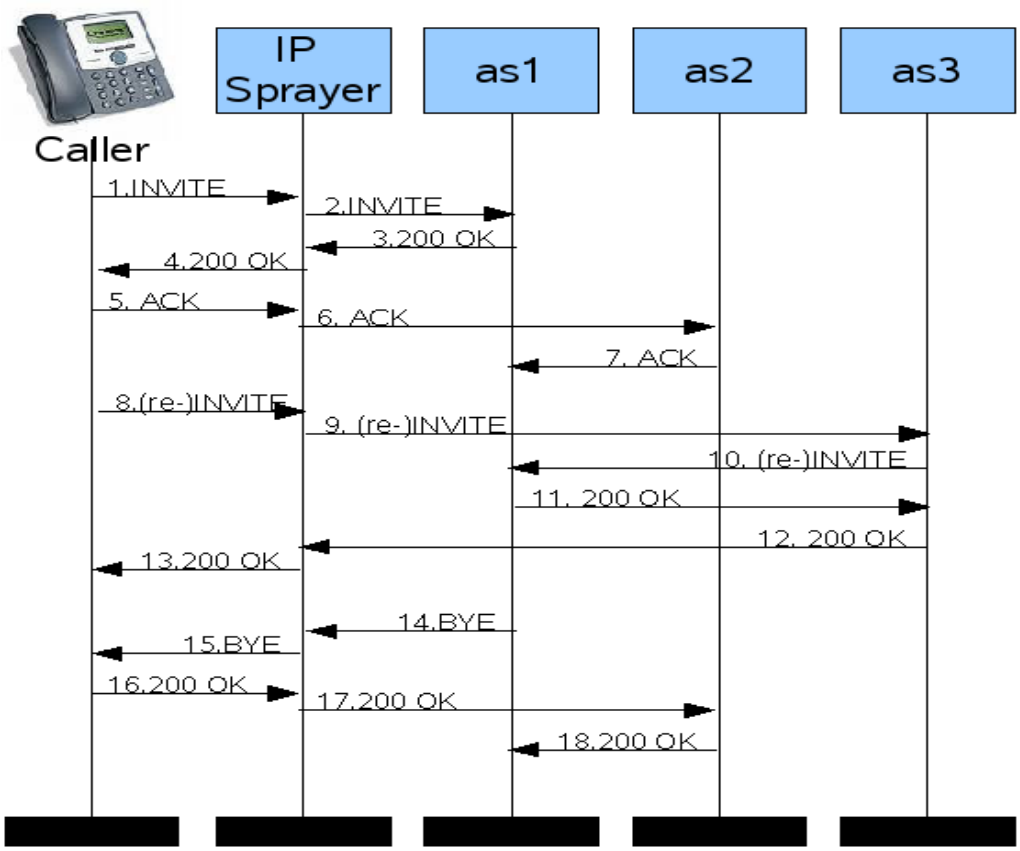


Figure 2

Figure 2 illustrates a typical SIP session (establish dialog, re-invite and close) when an external party is the initiator and an application on the TS acts as endpoint (initially UAS). In the the scenario the IP sprayer dispatches responses and subsequent requests to different AS:es in the cluster so that the FE in each AS needs to re-route or proxy.

1 *Caller* sends an INVITE request to *ts:*5060:

```
INVITE sip:app1@ts SIP/2.0
Via: SIP/2.0/TCP pc33.alpha;branch=z9h...
Contact: <sip:caller@pc33.alpha>
:
```

2 The IP sprayer selects *as1* to process the request.

3 The LBM at *as1* processes the request according to 8.2.1.1 and finds that the request should served locally. The LBM sends the request upwards in the stack. The application processes the request and issues a 200 OK response which is dispatched down through the SIP stack:

```
SIP/2.0 200 OK
Via: SIP/2.0/TCP pc33.alpha;branch=z9h...
Contact: <sip:app1@ts:5060>
:
```

On the way down the LBM processes the response according to 8.2.2.2.

```
SIP/2.0 200 OK
Via: SIP/2.0/TCP pc33.alpha;branch=z9h...
Contact: <sip:app1@ts:5060;bekey=hashkey>
:
```

Then the response is sent back to the IP sprayer via the original connection.

4 The response is sent back to *Caller*.

5 *Caller* issues an ACK and sends it to the IP sprayer with the URI specified in `Contact` as the request-URI, this including the `bekey`.

```
ACK sip:app1@ts:5060;bekey=hashkey  SIP/2.0
Via: SIP/2.0/TCP pc33.alpha;branch=z9h...
:
```

6 The IP sprayer selects *as2* to process the request.

7 The LBM at *as2* processes the request according to 8.2.1.1 and finds that it must proxy the request *as1*. The LBM of *as2* pushes the Via of the FE onto the request and adds temporary headers[1] to transfer information between the FE and BE:

```
ACK sip:app1@ts:5060;bekey=hashkey SIP/2.0
Via: SIP/2.0/TCP
hostport_of_as2;branch=z9h...;connid=connection_to_caller;felb;
Via: SIP/2.0/TCP pc33.alpha;branch=z9h...
Proxy-Remote: hostport to remote
Proxy-Bekey: hashkey
Proxy-Auth-Cert: client certificate
:
```

The LBM at *as1* processes the request according to 8.2.1.1 and finds that it shall serve the request. The LBM strips the temporary headers and sends the request upwards in the stack:

```
ACK sip:app1@ts:5060;bekey=hashkey SIP/2.0
Via: SIP/2.0/TCP
hostport_of_as2;branch=z9h...;connid=connection_to_caller;felb
Via: SIP/2.0/TCP pc33.alpha;branch=z9h...
:
```

Since it is and ACK no response is sent.

<< Media session is ongoing...>>

8 *Caller* sends re-INVITE to the IP sprayer with the URI specified in `Contact` as request-URI:

```
INVITE sip:app1@ts:5060;bekey=hashkey SIP/2.0
Via: SIP/2.0/TCP pc33.alpha;branch=z9h...
:
```

9 IP sprayer selects *as3* to process the request.

10 The LBM at *as3* processes the request according to 8.2.1.1 and finds that it must proxy the request to *as1*. The `Via` of the FE and temporary headers are pushed onto

1 Note, the Proxy-Auth-Cert is optional and will only be transferred if the incoming request was received via TLS.

the request:

```
INVITE sip:app1@ts:5060;bekey=hashkey SIP/2.0
Via: SIP/2.0/TCP
hostport_of_as3;branch=z9h..;connid=connection_to_caller;felb
Via: SIP/2.0/TCP pc33.alpha;branch=z9h...
Proxy-Remote: hostport to remote
Proxy-Bekey: hashkey
Proxy-Auth-Cert: client certificate
:
```

11 The LBM of *as1* processes the request according to 8.2.1.1 and finds that it shall serve the request. The LBM strips the temporary headers and then sends the request upwards in the stack and when the application has processed the request, *as1* sends a 200 OK response:

```
SIP/2.0 200 OK
Via: SIP/2.0/TCP
hostport_of_as3;branch=z9h...;connid=connection_to_caller;felb
Via: SIP/2.0/TCP pc33.alpha;branch=z9h...
:
```

to *as3* using the the connection of the incoming response[1]:

12 The LBM of *as3* processes the response according to 41 and finds that it acted as an FE and that the response has been received from a BE. *as3* pops the Via and sends the response back to the IP sprayer using the connection specified in `connid`:

```
SIP/2.0 200 OK
Via: SIP/2.0/TCP pc33.alpha;branch=z9h...
:
```

13 The IP sprayer forwards the response to *caller*

14 The application decides to close the call and sends BYE:

```
BYE sip:caller@pc33.alpha SIP/2.0
Via: SIP/2.0/TCP ts:5060; branch=z9h...
:
```

The LBM processes the request according to 8.2.2.1.

```
BYE sip:caller@pc33.alpha SIP/2.0
Via: SIP/2.0/TCP ts:5060; branch=z9h...;beroute=as1
:
```

15 The request is sent to *Caller*

16 *Caller* sends a 200 OK to the BYE:

```
SIP/2.0 200 OK
Via: SIP/2.0/TCP ts:5060; branch=z9h...;beroute=as1
:
```

17 The IP sprayer selects *as2* to process the response.

18 The LBM at *as2* processes the response according to 41 and finds that it must re-route the response to *as1*.
The LBM at *as1* the LBM processes the request according to 41 and finds that it shall serve the response and thus sends it up to the application which processes it and does nothing more.

---

1 Note that in case the connection is broken the sent-by value of the Via is used.

8.1.2 Typical SIP Session; An Application in TS is the Initiatior; UDP or TCP with Broken Connection

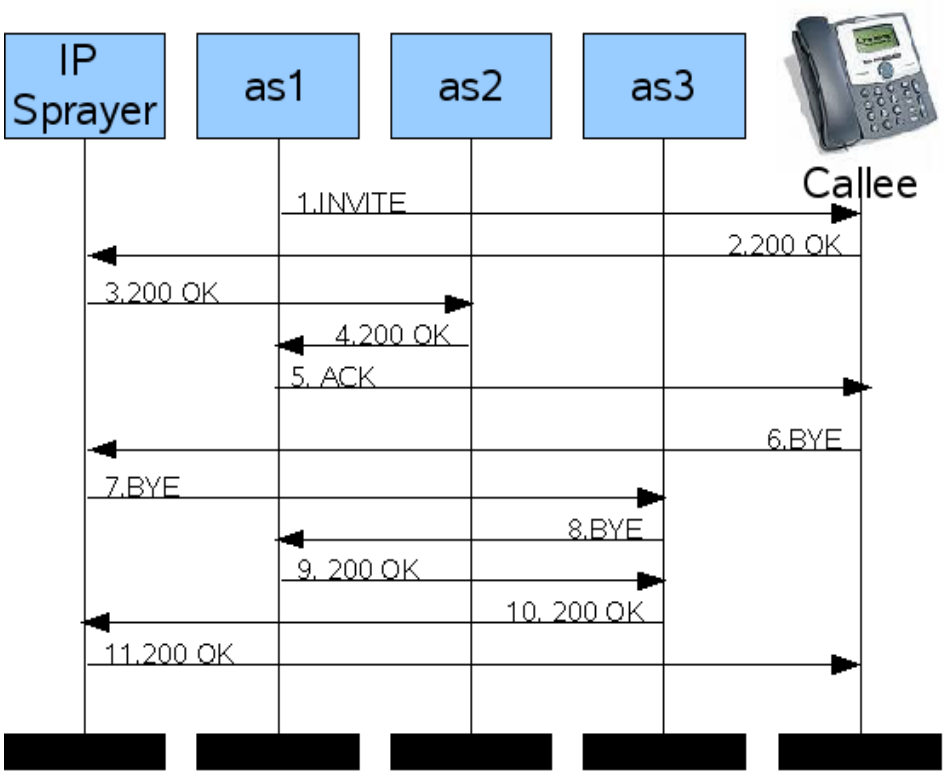


Figure 3

Figure 3 illustrates a typical SIP session (establish dialog, re-invite and close) when an application on the TS is the initiator (thus initially UAC). In the the scenario the IP sprayer dispatches responses and subsequent requests to different AS:es in the cluster so that the FE in each AS needs to re-route or proxy.

The scenario illustrates the case when communication is carried out via UDP or when the TCP connection to the *callee* is broken for some reason[1], so that *callee* must send the response via a new connection established using the information in the Contact header.

1 An application *app1* on *as1* sends an INVITE request to *callee* down the stack:

```
INVITE sip:callee@ceasar.com SIP/2.0
Via: SIP/2.0/TCP ts:5060;branch=z9h...
Contact: <sip:app1@ts>
:
```

The LBM processes the request according to 8.2.2.1 and since it is an initial request a hash key is generated and set in the `bekey` parameter added to `Contact`; `beroute` is set in `Via`:

```
INVITE sip:callee@ceasar.com SIP/2.0
Via: SIP/2.0/TCP ts:5060;branch=z9h...;beroute=as1
Contact: <sip:app1@ts;bekey=hashkey>
:
```

2 Callee sends a 200 OK:

```
SIP/2.0 200 OK
Via: SIP/2.0/TCP ts:5060;branch=z9h...;beroute=as1
Contact: <sip:callee@pc44.beta>
:
```

3 The IP sprayer selects *as2* to process the response.

4 The LBM at *as2* processes the response according to 41 and finds (from the value of `beroute`) that the response should served by *as1*; the LBM of *as2* re-routes the response to *as1* without altering anything in the response.

5 The LBM at *as2* processes the response according to 41
and finds that it shall serve the response and thus sends it up to the application.
The application processes the response and sends an ACK:

```
ACK sip:callee@pc44.beta SIP/2.0
Via: SIP/2.0/TCP ts:5060;branch=z9h...
:
```

LBM processes the request according to 8.2.2.1:

```
ACK sip:callee@pc44.beta SIP/2.0
Via: SIP/2.0/TCP ts:5060;branch=z9h...;beroute=as1
:
```

and sends it to the *callee*.

<< Media session is ongoing...>>

6 *Callee* decides to hang up and sends a BYE using URI specified in initial `Contact` header:

```
BYE sip:app1@ts:5060;bekey=hashkey SIP/2.0
Via: SIP/2.0/TCP pc44.beta;branch=z9h...
:
```

7 IP sprayer selects *as3* to process the request.

1 In the normal TCP case the response travels through the same connection as the request, and consequently the response will arrive at the serving BE directly.

8 The LBM at *as3* processes the request according to 8.2.1.1 and finds that it must proxy the request to *as1*. The `Via` to as2 is pushed onto the request:

```
BYE sip:app1@ts:5060;bekey=hashkey SIP/2.0
Via: SIP/2.0/TCP
hostport_of_as3;branch=z9h...;connid=connection_to_callee;felb
Via: SIP/2.0/TCP pc44.beta;branch=z9h...
Proxy-Remote: hostport to remote
Proxy-Bekey: hashkey
Proxy-Auth-Cert: client certificate
:
```

9 The LBM at *as1* processes the request according to 8.2.1.1 and finds that it shall serve the request and sends the request upwards in the stack. When the application has processed the request *as1* sends 200 OK response downwards in the stack. The LBM processes it according to 8.2.2.2:

```
SIP/2.0 200 OK
Via: SIP/2.0/TCP
hostport_of_as3;branch=z9h..;connid=connection_to_callee;felb
Via: SIP/2.0/TCP pc44.beta;branch=z9h...
:
```

The response is sent back to *as3* on the incoming connection.

10 LBM of *as3* processes the response according to 41 and finds that it has acted as an FE and that the response has been received from a BE. The LBM pops the Via to itself and sends the response to the IP sprayer via the original connection to the IP sprayer obtained from `connid`:

```
SIP/2.0 200 OK
Via: SIP/2.0/TCP pc44.beta;branch=z9h...
:
```

11 The IP sprayer forwards the response to caller.

8.1.3 Typical SIP Session; TS is Proxy (UDP or broken connection TCP)

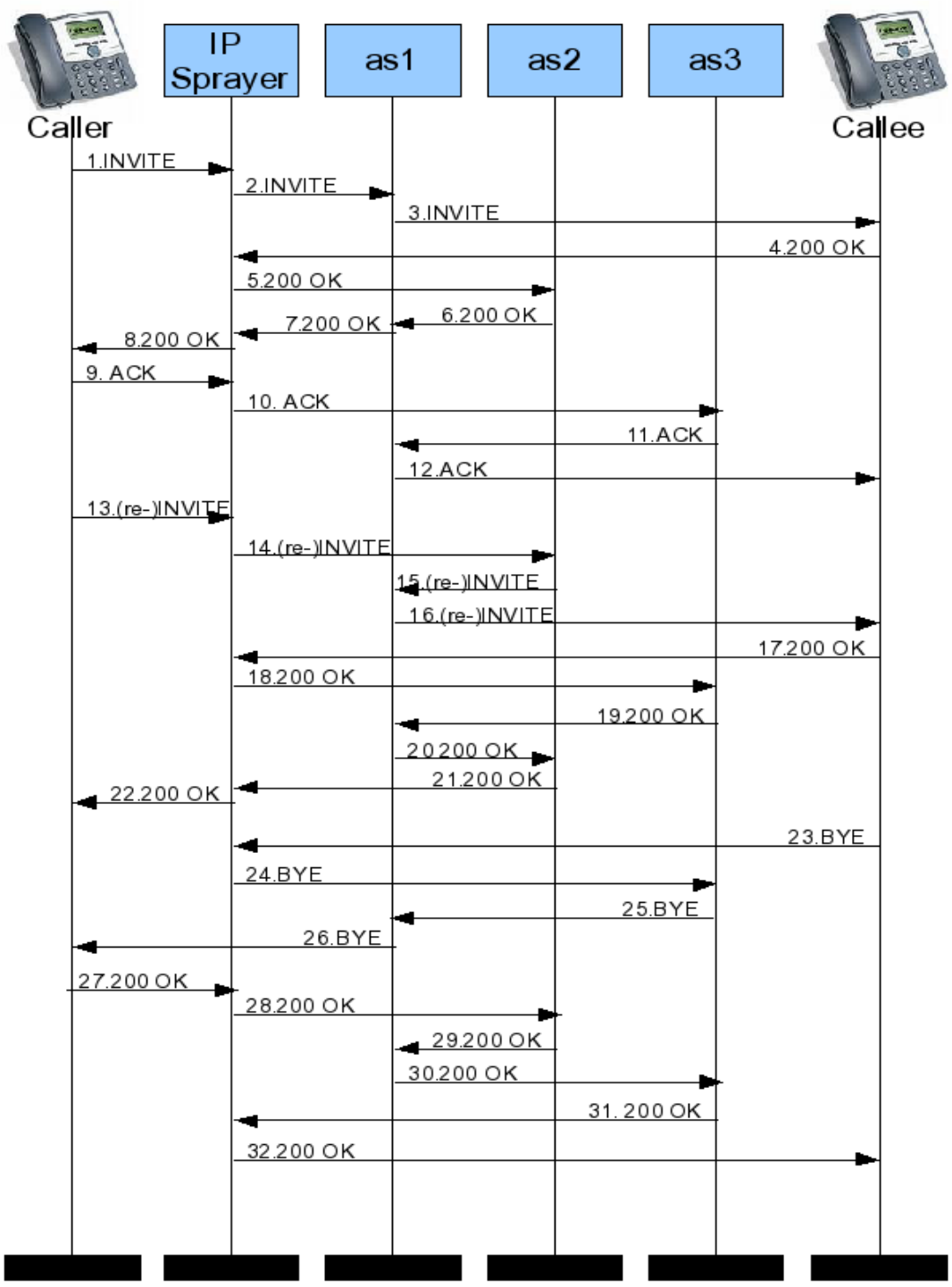


Figure 4

Figure 4 illustrates the case when an applications *app1* on the TS acts as a proxy between *caller* and *callee*. The scenario illustrates the case when communication is

carried out via UDP or when the TCP connection to the *callee* is broken for some reason[1], so that *callee* must send the response via a new connection established using the information in the Contact header.

1 *Caller* sends an INVITE request to *ts:*5060:

```
INVITE sip:callee@ceasar.com SIP/2.0
Via: SIP/2.0/TCP caller@pc33.alpha;branch=z9h...
Contact: <sip:caller@pc33.alpha>
:
```

2 The IP sprayer selects *as1* to process the request.

3 The LBM at *as1* processes the request according to 8.2.1.1 and finds that the request shall be served by *as1 a*nd sends the request upwards in the stack. The application decides to proxy the request and record-route:

```
INVITE sip:callee@ceasar.com SIP/2.0
Via: SIP/2.0/TCP app1@ts;branch=z9h...
Via: SIP/2.0/TCP pc33.alpha;branch=z9h...
Contact: <sip:caller@pc33.alpha>
Record-Route: <sip:app1@ts>
:
```

The LBM processes it according to 8.2.2.1:

```
INVITE sip:callee@ceasar.com SIP/2.0
Via: SIP/2.0/TCP
app1@ts;branch=z9h...;beroute=as1;connid=connection_to_caller
Via: SIP/2.0/TCP pc33.alpha;branch=z9h...
Contact: <sip:caller@pc33.alpha>
Record-Route: <sip:app1@ts;bekey=hashkey>
:
```

Finally the request is sent to the *callee.*

4 The *callee* stores the `Record-Route` and sends a 200 OK:

```
SIP/2.0 200 OK
Via: SIP/2.0/TCP
app1@ts;branch=z9h...;beroute=as1;connid=connection_to_ips
Via: SIP/2.0/TCP pc33.alpha;branch=z9h...
Contact: <sip:callee@pc44.beta>
Record-Route: <sip:ts;bekey=hashkey>
:
```

5 The IP sprayer selects *as2* to process the response.

6 The LBM of *as2* processes the response according to 41 and finds that that the response should served by *as1*; the LBM of *as2* re-routes the response to *as1* without altering anything in the response.

7 The LBM of *as1* processes the response according to 41 and finds that the response has arrived to the correct BE; the response is sent upwards in the stack, but before doing this the connection ID is stored temporarily in a response attribute.
The application processes the response (which implies that the top `Via` is popped) and then it is dispatched down the SIP stack again.
The LBM processes the response according to 8.2.2.2 and retrieves the connection ID of the incoming request from the response attribute. The response is sent back to

1 In the normal TCP case the response travels through the same connection as the request, and consequently the response will arrive at the serving BE directly.

the IP sprayer via the retrieved connection:

```
SIP/2.0 200 OK
Via: SIP/2.0/TCP caller@alpha;branch=z9h...
Contact: <sip:callee@pc44.beta>
Record-Route: <sip:ts;bekey=hashkey>
:
```

8 The IP sprayer sends the response back to *caller,* which will store the `Record-Route`.

9 The *caller* sends an ACK:

```
ACK sip:callee@pc44.beta SIP/2.0
Via: SIP/2.0/TCP pc33.alpha;branch=z9h...
Route: <sip:ts;bekey=hashkey>
:
```

10 The IP sprayer select *as3* to serve the request.

11 The LBM of *as3* processes the request according to 8.2.1.1 and proxies the request to *as1,* but first it pushes itself on the `Via`:

```
ACK sip: callee@pc44.beta SIP/2.0
Via: SIP/2.0/TCP
hostport_of_as3;branch=z9h...;connid=connectionto_ips;felb
Via: SIP/2.0/TCP pc33.alpha;branch=z9h...
Route: <sip:ts;bekey=hashkey>
Proxy-Remote: hostport to remote
Proxy-Bekey: hashkey
Proxy-Auth-Cert: client certificate
:
```

12 The LBM of *as1* processes the request according to 8.2.1.1 and finds that it shall serve the request. The application proxies the ACK to *callee* and sends it downwards in the stack. During this the stack pops the `Route` and pushes the `Via`:

```
ACK callee@pc44.beta SIP/2.0
Via: SIP/2.0/TCP app1@ts;branch=z9h...
Via: SIP/2.0/TCP
hostport_of_as3;branch=z9h...;connid=connection_to_ips;felb
Via: SIP/2.0/TCP pc33.alpha;branch=z9h...
:
```

The LBM processes the request according to 8.2.2.1 and extracts incoming connection from the request object and sets them in topmost `Via`:

```
ACK callee@pc44.beta SIP/2.0
Via: SIP/2.0/TCP
app1@ts;branch=z9h...;beroute=as1;connid=connection_to_as3
Via: SIP/2.0/TCP
hostport_of_as3;branch=z9h...;connid=connection_to_ips;felb
Via: SIP/2.0/TCP pc33.alpha;branch=z9h...
:
```

Finally the request is sent to the *callee*.

<< Media session is ongoing...>>

13 The *caller* send a (re-)INVITE:

```
INVITE sip:callee@pc44.beta SIP/2.0
Via: SIP/2.0/TCP pc33.alpha;branch=z9h...
Route: <sip:ts;bekey=hashkey>
:
```

14 The IP sprayer selects *as2* to serve the request.

15 The LBM of *as2* processes the request according to 8.2.1.1 and proxies the request to *as1*, but first it pushes itself on the `Via`:

```
INVITE sip: callee@pc44.beta SIP/2.0
Via: SIP/2.0/TCP
hostport_of_as2;branch=z9h...;connid=connection_to_ips;felb
Via: SIP/2.0/TCP pc33.alpha;branch=z9h...
Route: <sip:ts;beey=hashkey>
Proxy-Remote: hostport to remote
Proxy-Bekey: hashkey
Proxy-Auth-Cert: client certificate
:
```

16 The LBM on *as1* processes the request according to 8.2.1.1 and finds that it shall serve the request:

```
INVITE sip: callee@pc44.beta SIP/2.0
Via: SIP/2.0/TCP
hostport_of_as2;branch=z9h...;connid=connection_to_ips;felb
Via: SIP/2.0/TCP pc33.alpha;branch=z9h...
Route: <sip:ts;bekey=hashkey>
:
```

The application proxies and parameter are set in the request in the same manner as described in step 12 and finally the request is sent to the *callee*:

```
INVITE sip: callee@pc44.beta SIP/2.0
Via: SIP/2.0/TCP
sip:app1@ts;branch=z9h...;beroute=as1;connid=connection_to_as2
Via: SIP/2.0/TCP
hostport_of_as2;branch=z9h...;connid=connection_to_ips;felb
Via: SIP/2.0/TCP pc33.alpha;branch=z9h...
:
```

17 The *callee* sends a 200 OK:

```
SIP/2.0 200 OK
Via: SIP/2.0/TCP
sip:app1@ts;branch=z9h...;beroute=as1;connid=connection_to_as2
Via: SIP/2.0/TCP
hostport_of_as2;branch=z9h...;connid=connection_to_ips;felb
Via: SIP/2.0/TCP pc33.alpha;branch=z9h...
:
```

18 The IP sprayer selects *as3* to process the response.

19 The LBM on *as3* re-routes the response to as1 according to 41.

20 The LBM on *as1* processes the response according to 41 and saves the `connid` in a response attribute and sends the response upwards in the stack.
The application forwards the response (the stack pops the topmost `Via`) by dispatching it down the stack.
The LBM processes the response according to 8.2.2.2 and sends it back to *as2* using the connection using the value of the response attribute:

```
SIP/2.0 200 OK
Via: SIP/2.0/TCP
hostport_of_as2;branch=z9h...;connid=connection_to_ips;felb
Via: SIP/2.0/TCP pc33.alpha;branch=z9h...
:
```

21 The LBM on *as2* processes the response according to 41 and sees that it was an FE and pops `Via` and sends response to IP sprayer:

```
SIP/2.0 200 OK
Via: SIP/2.0/TCP pc33.alpha;branch=z9h...
:
```

22 The IP sprayer sends response to the *caller.*

23 The *callee* decides to hang-up and sends a BYE:

```
BYE sip: caller@alpha SIP/2.0
Via: SIP/2.0/TCP pc44.beta;branch=z9h...
Route: <sip:ts;bekey=hashkey>
:
```

24 The IP sprayer selects *as3* to process the request.

25 The LBM at *as3* processes the request according to 8.2.1.1 and proxies to *as1.*

```
BYE sip: caller@alpha SIP/2.0
Via: SIP/2.0/TCP
hostport_of_as3;branch=z9h...;connid=connection_to_ips;felb
Via: SIP/2.0/TCP pc44.beta;branch=z9h...
Route: <sip:ts;bekey=hashkey>
:
```

26 The LBM at as1 processes the request according to 8.2.1.1 and sends it upwards in the stack:

```
BYE sip: caller@alpha SIP/2.0
Via: SIP/2.0/TCP
hostport_of_as3;branch=z9h...;connid=connection_to_ips;felb
Via: SIP/2.0/TCP pc44.beta;branch=z9h...
Route: <sip:ts;bekey=hashkey>
:
```

The application proxies (`Via` is pushed) and dispatches down the stack:

```
BYE sip: caller@alpha SIP/2.0
Via: SIP/2.0/TCP sip:app1@ts;branch=z9h...
Via: SIP/2.0/TCP
hostport_of_as3;branch=z9h...;connid=connection_to_ips;felb
Via: SIP/2.0/TCP pc44.beta;branch=z9h...
:
```

and LBM sets parameters in the same manner as described in step 12 :

```
BYE sip: caller@alpha SIP/2.0
Via: SIP/2.0/TCP
sip:app1@ts;branch=z9h...;beroute=as1;connid=connection_to_as3
Via: SIP/2.0/TCP
hostport_of_as3;branch=z9h...;connid=connection_to_ips;felb
Via: SIP/2.0/TCP pc44.beta;branch=z9h...
:
```

and sends to the `caller`.

27 The caller sends a 200 OK:

```
SIP/2.0 200 OK
Via: SIP/2.0/TCP
sip:app1@ts;branch=z9h...;beroute=as1;connid=connection_as3
Via: SIP/2.0/TCP
hostport_of_as3;branch=z9h...;connid=connection_to_ips;felb
Via: SIP/2.0/TCP pc44.beta;branch=z9h...
:
```

28 The IP sprayer selects *as2* to process the response.

29 The LBM at *as2* re-routes to *as1.*

30 The LBM at as1 saves parameters as in step 20 and sends the response upwards in the stack.
The application proxies response (`Via` is popped) and LBM processes it as in step 20 and finally the response is sent back to *as2* via connection retrieved from the value of the response attribute:

```
SIP/2.0 200 OK
Via: SIP/2.0/TCP
hostport_of_as3;branch=z9h...;connid=connection_to_ips;felb
Via: SIP/2.0/TCP pc44.beta;branch=z9h...
:
```

31 The LBM at *as2* sees that it was an FE, as in step 21, pops the `Via` and sends response to the IP sprayer via the connection in `connid`.
32 **The IP sprayer sends the response to the *callee.***

8.2 Pseudo-code for LBM

8.2.1 Front-End and Back-End

8.2.1.1 Incoming Request

The LBM does the following when it receives an outgoing response (in next `(SipServletRequestImpl)`), i.e, a request has been received by NM and has been sent up through the stack.

1 **If** `Proxy-Bekey` header exists:
1.1 The hash key is extracted from `Proxy-Bekey`. The `Proxy-Bekey` header is removed.
2 **Otherwise If** `Route` exists:
2.1 The hash key is extracted from the `bekey` parameter of the `Route`.
3 **Otherwise If** request-URI contains bekey**:**
3.1 The hash key is extracted from the `bekey` parameter of the `request-URI`.
4 **Otherwise If** request method = ACK:
4.1 The hash key is extracted from the `bekey` parameter of `To`.
5 **If** hash key has not been found above:
5.1 generate the hash key from various fields in the request via a configurable algorithm.
6 Save hash key as an member variable "bekey" of SipServletRequest.
7 Use hash key to lookup *serving backend* via consistent hash.
8 **If** *serving backend* points at this AS (i.e, the request has arrived at the correct AS and processing shall continue in this AS):
8.1 Extract remote connection from `Proxy-Remote` and set on request object.
8.2 Extract client certificate from `Proxy-Auth-Cert` and set on request object.
8.3 Save the incoming connection as a request attribute, “connid”.
8.4 Send the request up to the next layer above.
9 **Otherwise** (i.e, *serving backend* points at another AS)
9.1 The current AS is the FE and must proxy to the BE specified in *serving backend*: Push Via (pointing to this FE) with the parameters*,* `connid`, containing *connection ID* of the incoming connection and `felb` indicating that this is the Via of the FE that did load balancing.
9.2 **If** protocol via which request was received != UDP:
9.2.1 Add the parameter `connid`, containing *connection ID* of the incoming connection, to the Via (pointing to this FE)*.*
9.3 Get remote connection and set in a `Proxy-Remote` header, which is added to the request.
9.4 Get client certificate and set in a `Proxy-Auth-Cert` header, which is added to the request.
9.5 Set hash key in a `Proxy-Bekey` header, which is added to the request.
9.6 Send request to BE.

8.2.1.2 Incoming Response

The LBM does the following when it receives an incoming response (in `next(SipServletResponseImpl)`), i.e, a response that has been received by NM and has been sent up through the stack.

1 **If** the topmost `Via` contains the parameter `felb` (i.e, this is the FE that has performed load balancing and the response has been received from the BE.):
1.1 Pop the `Via`
1.2 **If** `connid` parameter exists:
1.2.1 Forward the response using the connection identified by the parameter `connid`.
1.3 **Otherwise** (This will be the case when the request was received via UDP)
1.3.1 Forward the response using the resolved Via.
2 **Otherwise**
2.1 **If** topmost `Via` contains the parameter `beroute` (i.e, a dialog exists and route to BE has been established.):
2.1.1 Extract *serving backend* from `beroute`.
2.2 **Otherwise**
2.2.1 This should never happen: Error.
2.3 **If** *serving backend* is this AS (i.e, The response has arrived at the serving AS and processing shall continue on this AS):
2.3.1 **If** `connid` parameter exists:
2.3.1.1 Extract remote connection from the connection identified by the parameter `connid`.
2.3.2 **Otherwise** (This will be the case when the request was received via UDP)
2.3.2.1 Extract remote connection from the response using the resolved Via.
2.3.3 Save connection to remote party in “connid” attribute of response (to be retrieved in outgoing response if the response is forwarded by application that acts as proxy).
2.3.4 Send the response up to the next layer above.
2.4 **Otherwise** (i.e., the response came from the outside world and the IP sprayer has sent it to wrong AS.)
2.4.1 Re-route (do not proxy!) the response to that AS.

8.2.2 Back-End Only

8.2.2.1 Outgoing Request

The LBM does the following when it receives an outgoing request (in `dispatch(SipServletRequestImpl)`), i.e, a request that has been generated by higher layers and sent down through the stack.

1 **If** SipApplicationSession of request contains attribute "bekey":
2.5 Retrieve hash key from that attribute.
3 **Otherwise**
3.1 generate the hash key from various fields in the request via a configurable algorithm
3.2 Save hash key in attribute "bekey" of the SipSession.
4 Add address of current instance in parameter `beroute` to the topmost `Via`.
5 **If** the contact is set by this container AND request is initial:
5.1 Add the parameter `bekey`=*hash key* to `Contact`.
6 **Otherwise If** application has Record-Routed:
6.1 Add `connid`, containing the connection ID of the connection of the incoming request, to the topmost `Via` (so that it can be retrieved when the response shall be proxied later on).
6.2 Add `bekey` with the value of hash key (retrieved above) on the `Record-Route` added by the application(s).

8.2.2.2 Outgoing Response

The LBM does the following when it receives an outgoing response (in `dispatch(SipServletResponseImpl)`), i.e, a response that has been generated by higher layers and sent down through the stack.

1 **If** SipApplicationSession of response contains attribute "bekey":
6.3 Retrieve hash key from that attribute
7 **Otherwise**
7.1 **If** request associated with response contains attribute "bekey":
7.1.1 Retrieve hash key from that attribute
7.1.2 Add attribute "bekey" with hash key as value, to SipApplicationSession
8 **If** container has set `Contact`:
8.1 Add the hash key value in the parameter `bekey` on the `Contact`.
9 **Otherwise If** response method = INVITE AND status >= 300:
9.1 Add the hash key value in the parameter `bekey` on the `To`.
10 **If** SipSession of response contains attribute "connid":
10.1 Get attribute value and send the response on the connection identified by it.

8.3 Headers added by CLB

CLB would add the following headers to the request received, to support the changes enlisted in section 2.5 . These are added while forwarding / proxying the request to a selected server instance, which can be local or remote. At reception they are removed.

8.3.1 proxy-beroute

This is added for the HTTP request being load balanced; when the configured load balancing policy for HTTP is round-robin. CLB, upon selection of the server instance to which the request needs to be forwarded, would add this header. The value of which is the BERoute sticky information .
For the case of failover, it's value is the value of the selected server instance's BERoute.

8.3.2 proxy-bekey

This is added for HTTP and SIP (in SIP it is named Proxy-Bekey to be a well-formed SIP header) requests being serviced. It would be added for HTTP request, in case the configured load balancing policy for HTTP requests is consistent hashing. The value of this header is the BEKey sticky information.
The value of this; remains same in case of failover of the request.
In SIP it is only kept in the hop between the front-end and the back-end, i.e. the CLB at the back-end removes it

8.3.3 Proxy-Remote

This is added for the SIP request being load balanced in the hop between the front-end and the back-end, i.e, the back-end removes it. It is used to transfer information (host and port) for the remote socket, so that it can be made available for the application at the back-end

8.3.4 Proxy-Remote

This is added for the SIP request being load balanced in the hop between the front-end and the back-end, i.e, the back-end removes it. It is used to transfer the client certificate (if existing), so that it can be made available for the application at the back-end

8.4 DCR Schematic Description

8.4.1 Operators

The following logical operators are supported in DCR:

If: contains a condition which is evaluated. If the condition evaluates to a value which is not null the evaluation continues in the if-branch. If the condition evaluates to null the evaluation continues in the optional elsebranch.

and: contains a number of conditions and evaluates to true if and only if all contained conditions evaluate to a value which is not null.

or: contains a number of conditions and evaluates to true if and only if at least one contained condition evaluates to a value which is not null.

8.4.2 Conditions

The following conditions are supported by DCR:

request-uri: is a condition containing the request URI and its parameters.

header: is a condition handling headers.

session-case: determines SIP session cases.

cookie: a condition based on HTTP cookies

8.4.3 Condition Types

A condition contains a condition type. All types are not valid for all conditions:

equal: compares the value of a variable with a literal value and evaluates totrue if the variable is defined and its value equals that of the literal. Otherwise,the result is false.

exist: takes a variable name and evaluates to true if the variable is defined,and false otherwise.

notexist: takes a variable name and evaluates to true if the variable is notdefined, and false otherwise.

match: evaluates a regular expression. The return value is ‘Group 1’, which is the pattern within the first pair of ellipsis *[16]*.

All operations are case sensitive.

8.4.4 Variables

A number of variables are defined in DCR, see 8.4.5 and 8.4.6. Variables containing resolve performs ENUM lookup of TelURI’s.

Variables can be used in conditions and for return values. The syntax formatch of HTTP requests is slightly different.

The <…> below indicates strings that can be chosen freely when designing a ruleset. E.g. request.P-Asserted-Identity refers to the header 'P-Asserted-Identity'.

The HTTP variable Â"parameter.<parameter>.uri.resolve.userÂ" is resolved in anelaborate way. The variable matches a parameter value in a HTTP request, and this value may be a single name-addr or a comma-separated sequenceof such. The name-addr elements are resolved until a usable Data Centric hashkey is found. The order of resolution is the following: First all name-addrsthat contain a SIP URI are considered from left to right. In case a SIP URI contains a user=phone parameter it is resolved as a TEL URL, otherwise the user part of the URI is extracted. Resolution of a SIP URI may thus fail if it specifies a telephone number entity that cannot be resolved by ENUM, or else because there is no user part present in the SIP URI. If all SIP URIs have been considered a second attempt is made, this time considering TEL URLs from left to right. Evaluation stops as soon as a usable Data Centric key has been found. If every resolution attempt fails, resulting no Data Centric key being found; if the context is a return value then the default rule of computing the Data Centric key from the Call-ID, From tag and To tag is used. In this case the values of these would concatenated to represent the key.

For example, if the variable is "parameter.from.uri.resolve.user" and the HTTP

request is

GET ...?...&from=...&... HTTP/1.1

the outcome may be according to the table below. Note that some of the characters in the example below may in reality need to be URL-encoded ('<' would appear as %3C etc).

value of "from" parameter	Data Centric key
<sip:server.xx.yy>	none
<sip:alice@server.xx.yy> alice	alice
<tel:+1-333-555>,<sip:+1-22-22@server.xx.yy;user=phone>	from ENUM

8.4.5 SIP Variables

request.uri
request.uri.scheme
request.uri.user
request.uri.host
request.uri.port
request.method

request.uri.resolve
request.uri.resolve.user
request.uri.resolve.host

request.<header>
request.<header>.uri
request.<header>.uri.scheme
request.<header>.uri.user
request.<header>.uri.host
request.<header>.uri.port
request.<header>.uri.display-name

request.<header>.uri.resolve
request.<header>.uri.resolve.user
request.<header>.uri.resolve.host

8.4.6 HTTP Variables

request.<header>
request.<header>.uri
request.<header>.uri.user
request.<header>.uri.host
request.<header>.uri.resolve
request.<header>.uri.resolve.user
request.<header>.uri.resolve.host

parameter.<parameter>
parameter.<parameter>.uri
parameter.<parameter>.uri.user
parameter.<parameter>.uri.host
parameter.<parameter>.uri.resolve
parameter.<parameter>.uri.resolve.user
parameter.<parameter>.uri.resolve.host

match
match.resolve.user

cookie.<cookieName>

8.5 DCR Example

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<user-centric-rules>
  <sip-rules>
  <if>
    <session-case>
      <equal>ORIGINATING</equal>
      <if>
        <header name="P-Asserted-Identity"
           return="request.P-Asserted-Identity.uri.resolve.user">
          <exist/>
        </header>
        <if>
          <header name="P-Asserted-Identity"
             return="request.from.uri.resolve.user">
            <notexist/>
          </header>
          <if>
            <header name="P-Asserted-Identity"
               return="request.to.uri.resolve.user">
              <notexist/>
            </header>
          </if>
        </if>
      </if>
    </session-case>
    <else return="request.uri.resolve.user" />
  </if>
  </sip-rules>
  <http-rules>
    <or>
      <request-uri return="match.resolve.user">
           <match>/users/([^/]+)</match>
      </request-uri>
         <and>
          <request-uri>
               <match>^/css/</match>
            </request-uri>
            <or>
              <request-uri parameter="referredBy" return="parameter.requestUri.uri.resolve.user">
                   <exist/>
              </request-uri>
            <request-uri parameter="referredBy" return="parameter.from.uri.resolve.user">
                   <notexist/>
              </request-uri>
          </or>
         </and>
       </or>
  </http-rules>
</user-centric-rules>
```