Functional Specification for Proxy (part of Converged LoadBalancer)

Author(s): ramesh.parthasarathy@sun.com

Version: 1.0

1. Introduction

The proxy in sailfin, which is a part of the converged HTTP-SIP load balancer, is an entity that is responsible for intercepting all the inbound HTTP and SIP requests, and forwarding it to an appropriate back end instance for further processing. Essentially the proxy functionality is predominantly, request interception and forwarding. In the larger sailfin architecture, it sits between the socket connector (grizzly, ...) and the converged load-balancer, and provides the load balancer with all the information that is required to make a routing decision. The proxy does not have any intelligence to determine the target instance to which the requests needs to be forwarded, this logic comes from the core load balancer. The proxy's tasks is limited to intercepting HTTP/SIP requests and parsing them (if necessary) to extract the header information, the headers are then used by the load balancer to determine the target instance that should be serving this request. The parsing process (if performed) is optimized by just identifying the byte boundaries of headers and this avoids creating string objects for all headers required by the CLB. The algorithm/routing-logic (sticky, consistent hashing.....) used by the load balancer is unknown to the proxy and is out of scope of its requirements/design principles. The implementation of the proxy will be stateless and it might depend on some standard header(s). Although, it might also use (set) some custom/proprietary header in the HTTP/SIP request in the internal path (proxy-sailfininstance-proxy communication) to convey certain decisions. For e.g in the HTTP and SIP path custom headers will be used to indicate that the request has already been fronted by a load balancer. Also some auth headers are needed to propagate the client certificate chain from front end to backend. These proprietary headers would be removed before the message is sent to the public network. It will act as a 'transparent proxy' in the HTTP path, which means it does not modify the request or response beyond what is required for proxy authentication and identification. However, in the SIP path, some SIP awareness is essential to be able to route the messages.

The proxy, as an entity is pluggable and can be enabled or disabled from the request processing chain of the HTTP/SIP container (listener). Enabling the proxy would provide proxying/load-balancing capabilities to the sailfin instance. In the disabled state, the HTTP/SIP containers would perform their normal request processing tasks, as if there was no proxy present, and would not incur any performance penalty. The life-cycle of the proxy is managed by the container(s) (web/sip) and is started/stopped along with the container. The ability to manage the life-cycle of the proxy through an external entity (other than the container(s)) will not be supported and is out of scope of this document.

It will support inbound HTTP requests (from external clients) over TCP and TLS (HTTPs) and SIP requests over TCP, UDP and TLS (SIPs). It will also perform certificate validation and ssl-handshake (offloading) if server authentication is required in the secure path. Handling HTTP/SIP requests include functions like determining the end of a request, parsing the headers etc.., for which the proxy will/might have to depend on certain APIs provided by the container. It will also be extensible so that other application protocols and transports can be supported in future. The inbound functionality in proxy will be based on Grizzly 1.0 port unification mechanism and as an effect the proxy/clb functionality cannot not be supported with the Coyote Connector in sailfin.

The implementation of the proxy outbound functionality in sailfin (SCAS) will be based on the grizzly connector (1.5.x). However, in the inbound path some artifacts of the proxy will be in based on grizzly 1.0 (for HTTP), depending on what is being used in sailfin.

It will not perform any caching of http/sip messages. It is also unaware of any SIP UDP retransmissions that the external client might send, it is the responsibility of the sailfin instances to handle retransmissions that might occur as a result of UDP packet time-outs.

The following documents would provide a better understanding of the purpose of this module and its role in the sailfin project..

This document covers the basic use cases of the http and sip proxy functionality which the converged load balancer depends upon. However it includes operational details only for the http implementation, the working details of the sip side will be provided in the converged load balancer specification.

.1.1 Terminologies:

For the purpose of this document, the term “proxy” always refers to the entity that is being implemented through this functional specification. It should not be misunderstood for a “SIP proxy” (which is a SIP application), which would be explicitly stated if required.

Any reference to “converged loadBalancer (CLB)” or the “core loadBalancer” refers to the entity that is responsible for the routing decision, more description of how this entity performs its functions will be described in the Converged LoadBalancer functional specification. For the purpose of the “proxy” the core loadBalancer is a black-box.

“FrontEnd” of a sailfin instance refers to the module(s) that perform the proxy and load balancing functions. Proxy and the core load balancer are part of the fronted.

“Backend” of a sailfin instance refers to the module(s) that would actually be processing the request and generating the response. These include the HTTP and SIP containers and the applications deployed on them.

1.2 Features/Requirements

Supported :

1. Receive HTTP requests over TCP/TLS, parse and extract header information from it.
2. Receive SIP requests over TCP/UDP/TLS, parse and extract header information from it. (implementation details covered in CLB specification[3])
3. Forward the HTTP requests over TCP to the selected instance after the routing decision is made.
4. Forward the SIP requests over TCP to the selected instance after the routing decision is made.(implementation details covered in CLB specification[3])
5. Optimize the request forwarding path if the request has to be routed to the same instance. Applicable when the sailfin instance is both a front end and a back end, and the front end has to forward it to the back end in the same instance.
6. Export interfaces for allowing the container to perform certain configurations.
7. Should allow the container(s) to control its life-cycle by exposing an API.
8. Provide activity and event logging (using logger API in server.log). Does not have a separate log file.
9. Asynchronous client channels (proxy-instance) using grizzly client API.
10. Should allowinterceptors/functional blocks to be plugged into interception chain.

Not Supported :

10. Does not support HTTP caching.
11. Will not support filtering.
12. No plans of supporting ACLs or other forms of checks in the inbound path.
13. It will not perform any Network Address Translations (NAT) functions.

<List all requirements and features you are implementing. List those which may be normally expected to be implemented but are not.>

2. Design Overview

The proxy will be implemented using the Grizzly NIO client framework in the HTTP path. The SIP implementation will be done as part of Converged Load balancer[3]. The implementation will cater to the following basic SIP/HTTP use cases. Other use cases that have not been covered here (if any) , will be satisfied on a need basis.

.2.1 Use Cases - Runtime

Some points that are valid for all SIP use cases

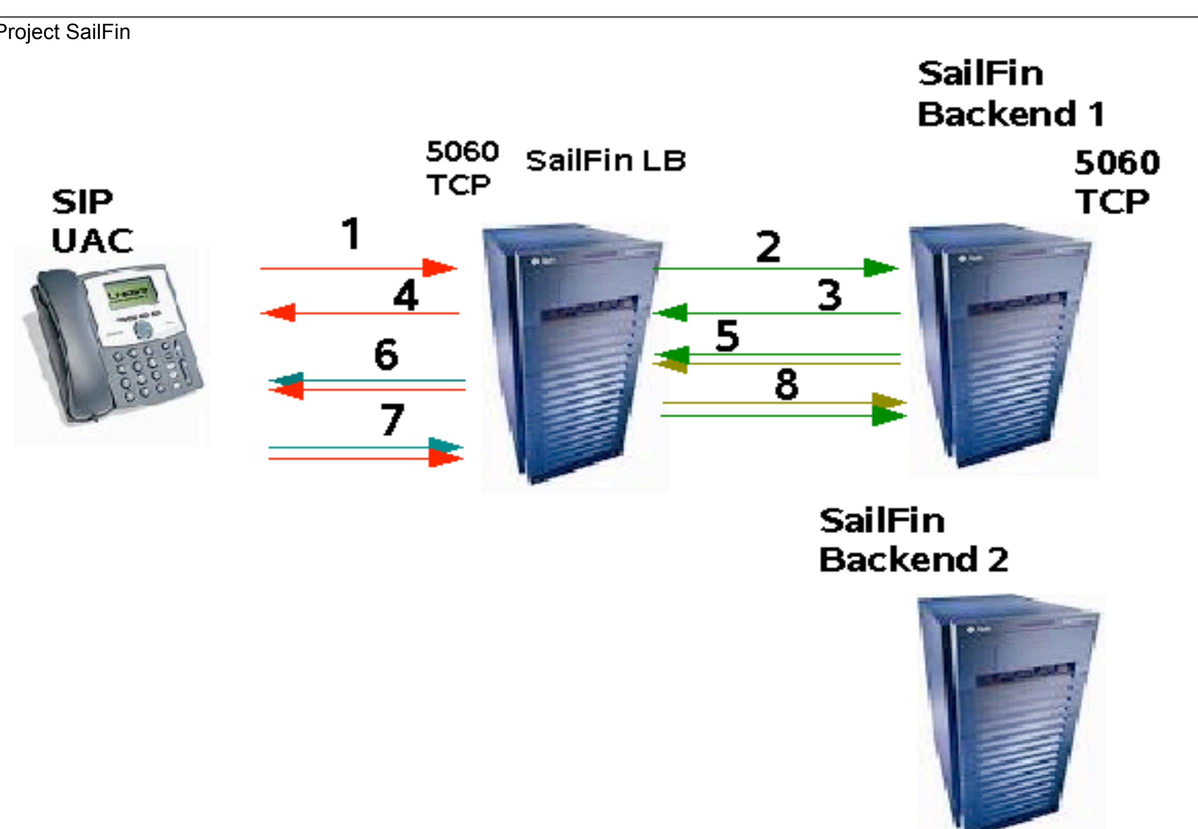
1. *The proxy functionality in SIP will be implemented as part of the load balancer in the SIP stack. This is a deliverable from Ericcson.*
2. *The proxy is not an individual component in the SIP stack.*
3. *The changes which are required for implementing the proxy'ing behavior will be independent of the network layer in the SIP stack*
4. *The use cases have been mentioned in this FS for completeness of the converged functionality.*

The scenario that has been considered here is a SIP INVITE request from a UAC.

The following are the possible exchanges (at the socket layer) for such a request.

2.1.1 SIP UAC sends a TCP INVITE request to the proxy:

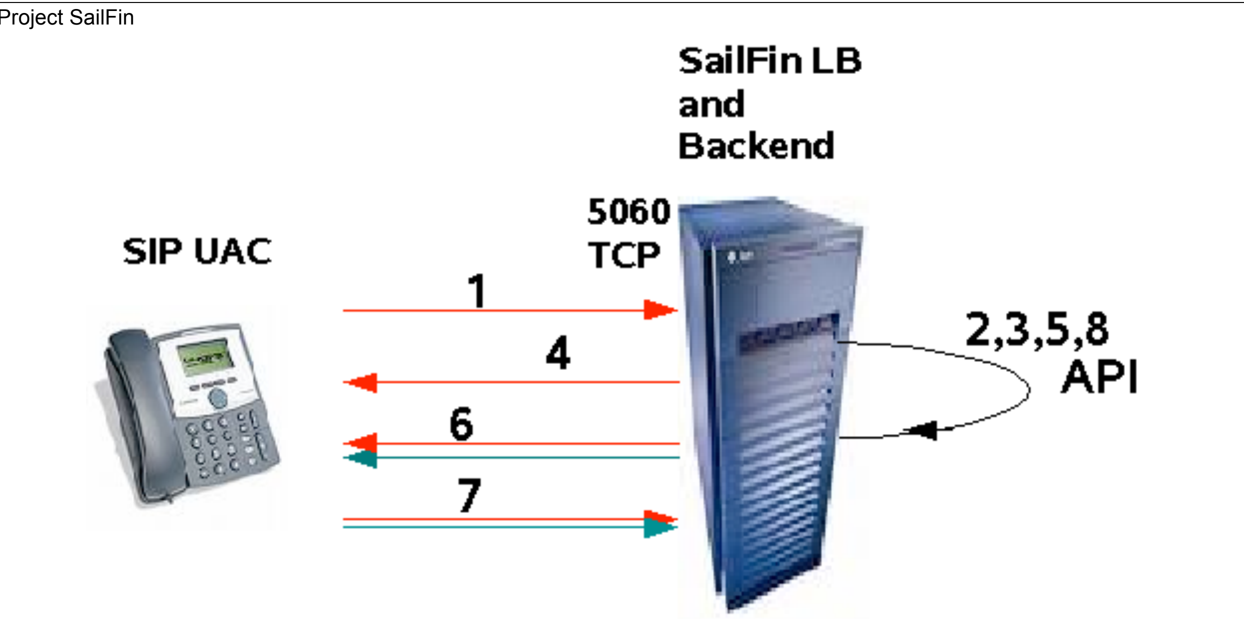
2.1.1.1 Proxy forwards it using TCP



1,4,6,7: Client-proxy channel(s)
2,3,5,8 : Proxy-instance channel(s)
unique channels are indicated using different color schemes

Refer Converged Load Balancer functional specification [3], Appendix – SIP Use cases

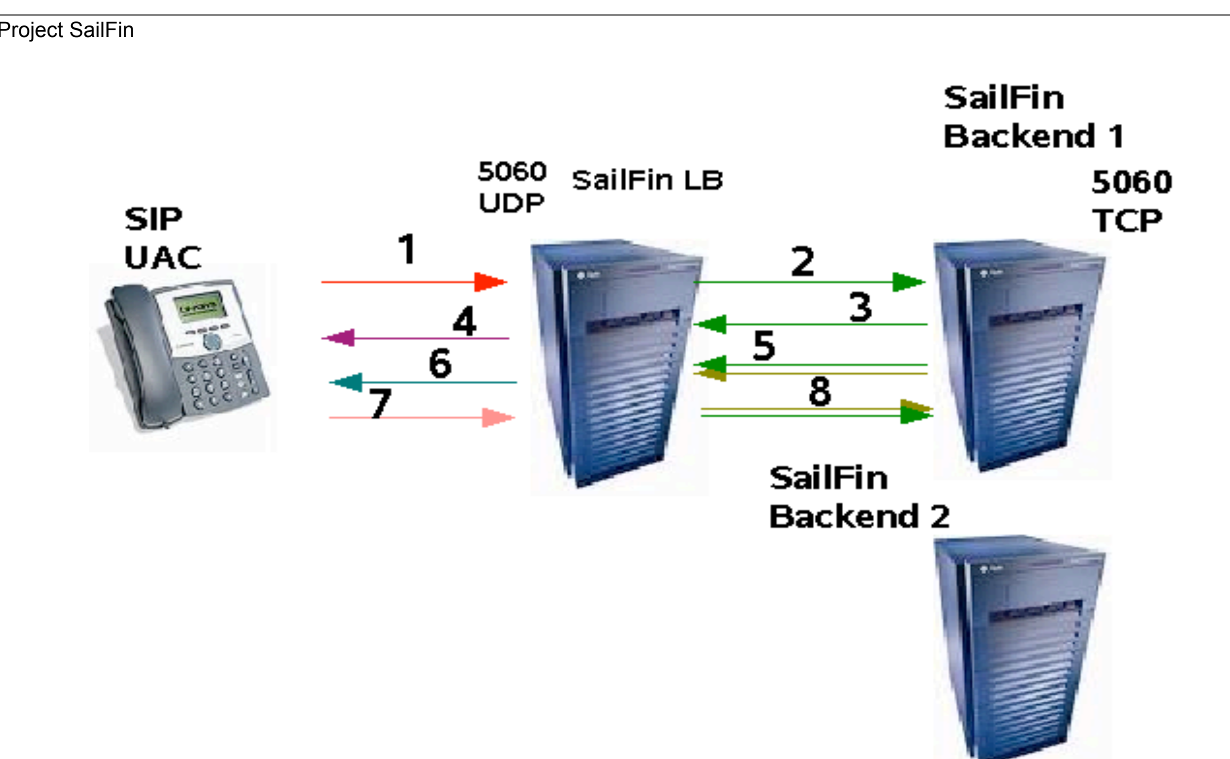
2.1.1.2 Proxy forwards it to the local instance.



Refer Converged Load Balancer functional specification [3], Appendix – SIP Use cases

2.1.2 SIP UAC sends a TCP INVITE request to the proxy:

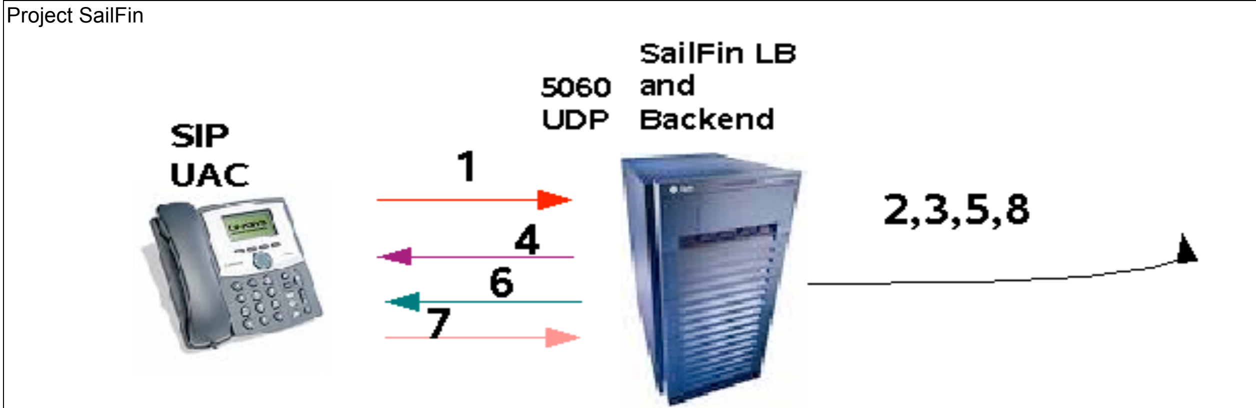
2.1.2.1 Proxy forwards it to the using TCP: Here the protocol used between the instance and the proxy is always TCP though the request was received through UDP.



1,4,6,7: Client-proxy UDP request/response(s)
2,3,5,8 : Proxy-instance TCP channel(s)
unique channels are indicated using different color schemes

Refer Converged Load Balancer functional specification [3], Appendix – SIP Use cases

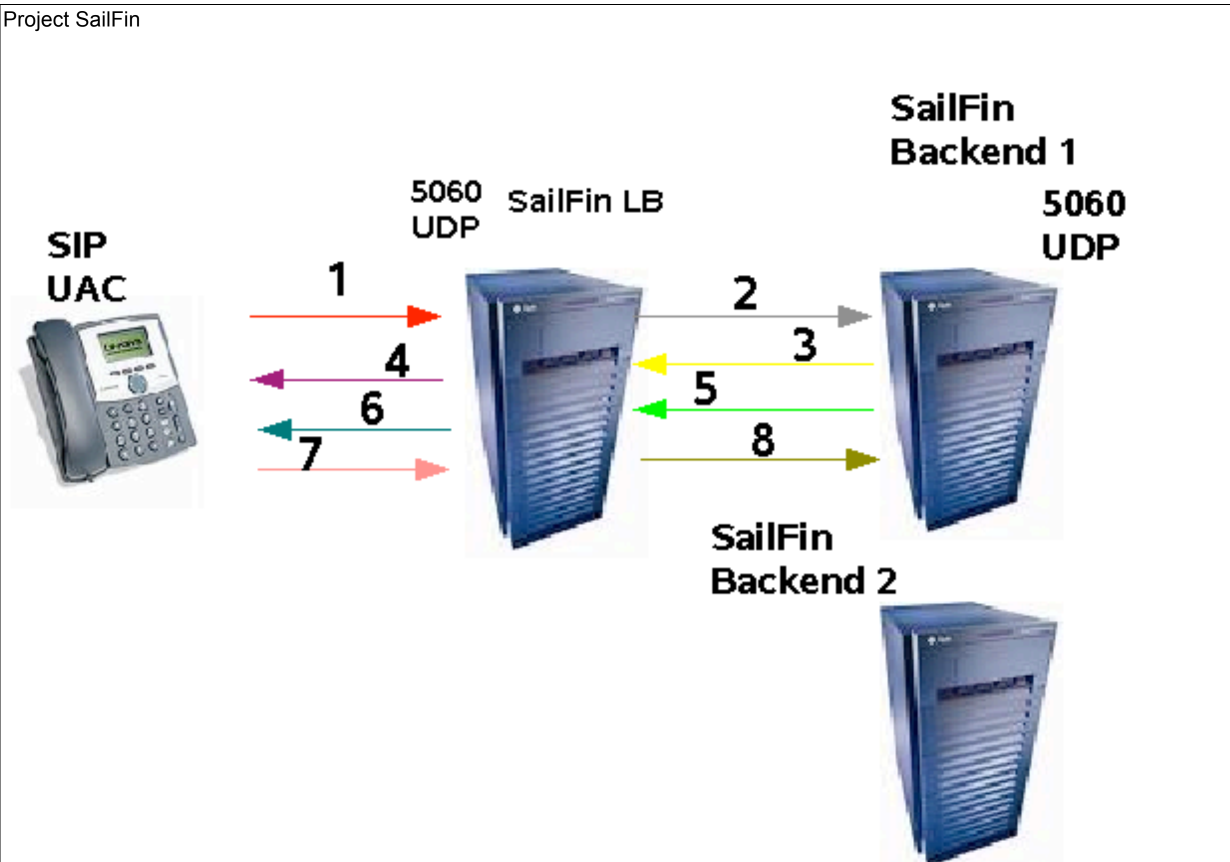
2.1.2.2 Proxy forwards it to local instance.



Refer Converged Load Balancer functional specification for how this is handled

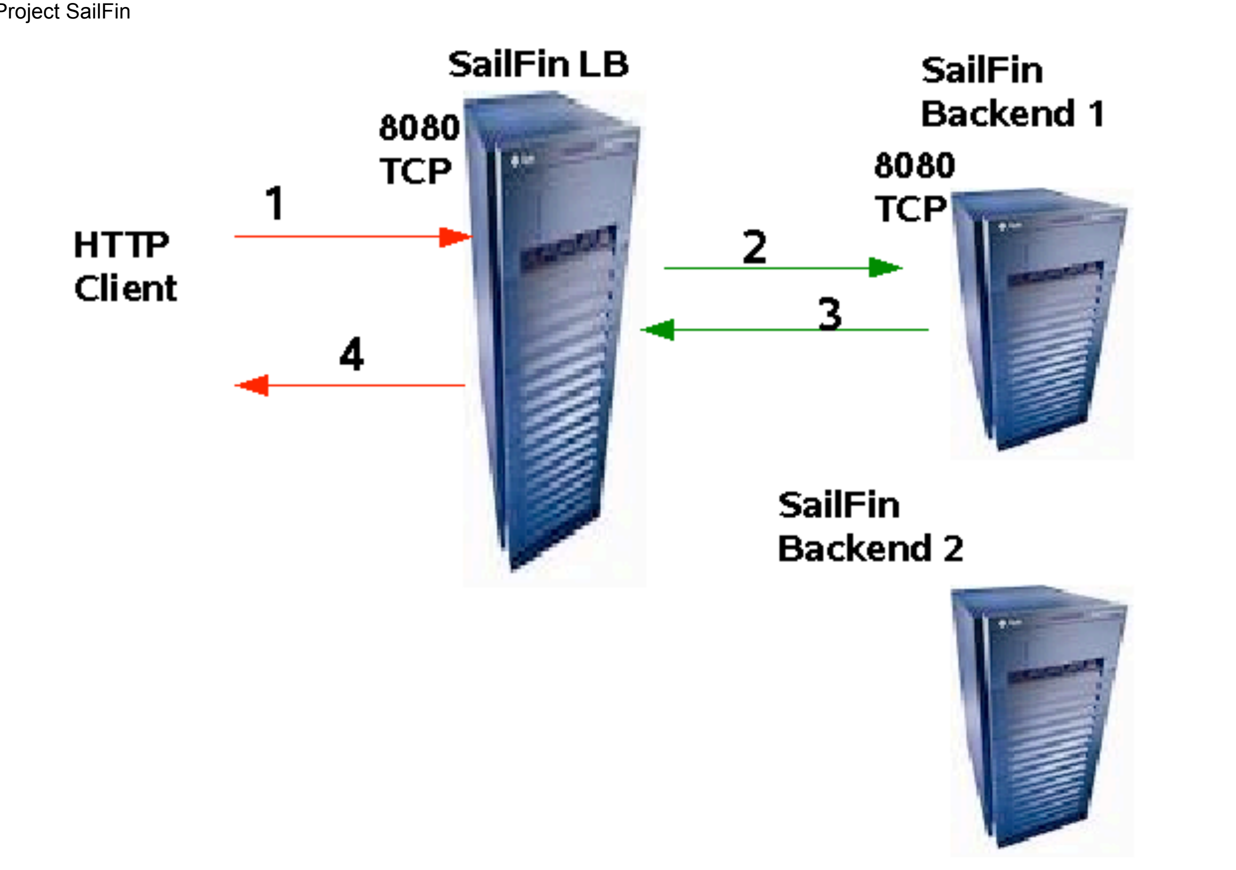
2.1.2.3 Proxy forwards it to using UDP. The option of using UDP for internal communication (proxy-instance) ***will not be supported.*** *It has been described here only for the sake of completeness of the use cases.*

Not Supported



2.1.2 HTTP client sends a HTTP request (over TCP)

2.1.2.1 Proxy forwards it to a remote instance



1. HTTP client sends TCP request to the proxy, this accepted channel is maintained (kept alive) until a response is received from the backend serving instance (or until a time-out).
2. The instance to which it needs to forward to is determined, this is done by the core converged load balancer, using the configured policy. The proxy does the header parsing and makes all the headers available to the core load balancer, which enables it to take a routing decision. The proxy forwards the request to the instance1 by opening a TCP channel to it. The proxy might also add some information to indicate that this message was a proxy   by the LB. This would be using a private header field. Please refere to Table 4.2.1 for details
3. The remote backend instance responds back on the same channel created in Step 2.
4. The proxy picks the client channel (in Step 1) and sends the response back.

2.1.2.2 Proxy forwards it to a local instance

The steps are same as in 3.1 but only that 2,3 are performed through API calls rather than socket channels.

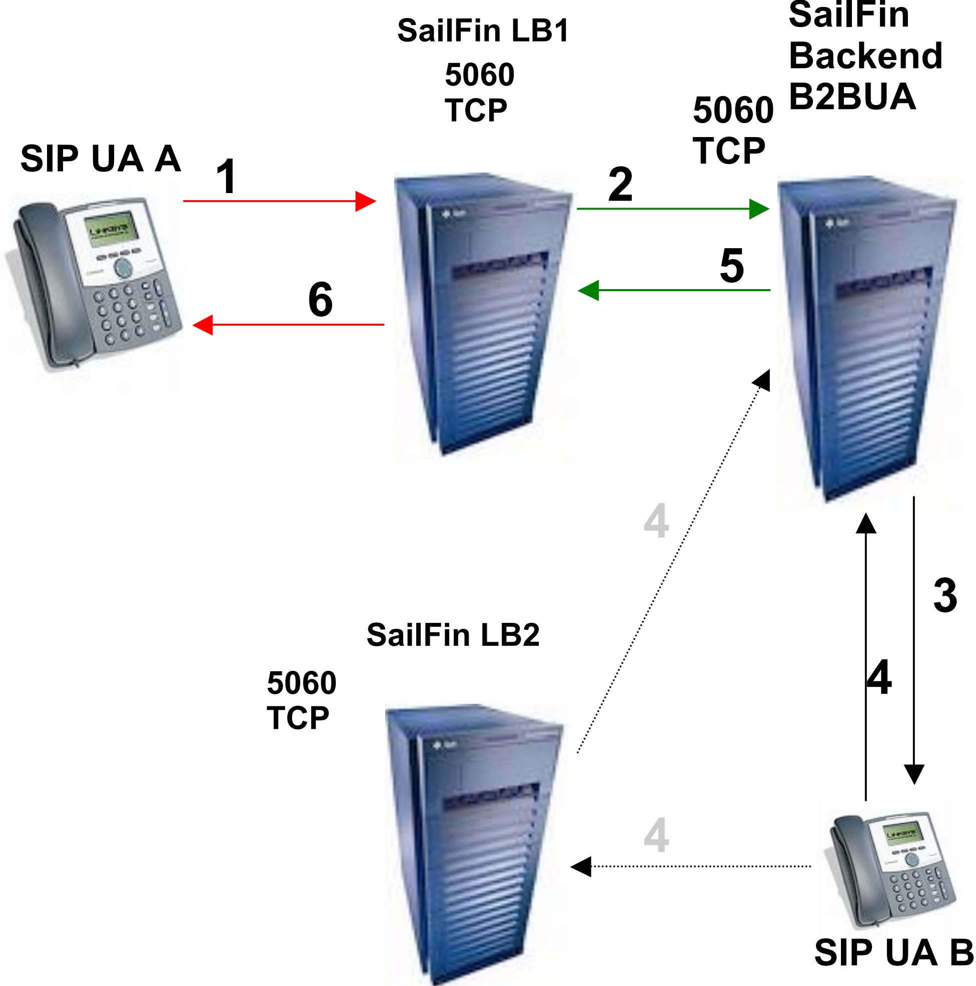
2.1.3 SIP UAC sends a SIPs request.

Refer Converged Load Balancer functional specification [3], Appendix – SIP Use cases

2.1.4 Back-to-Back User Agent Application

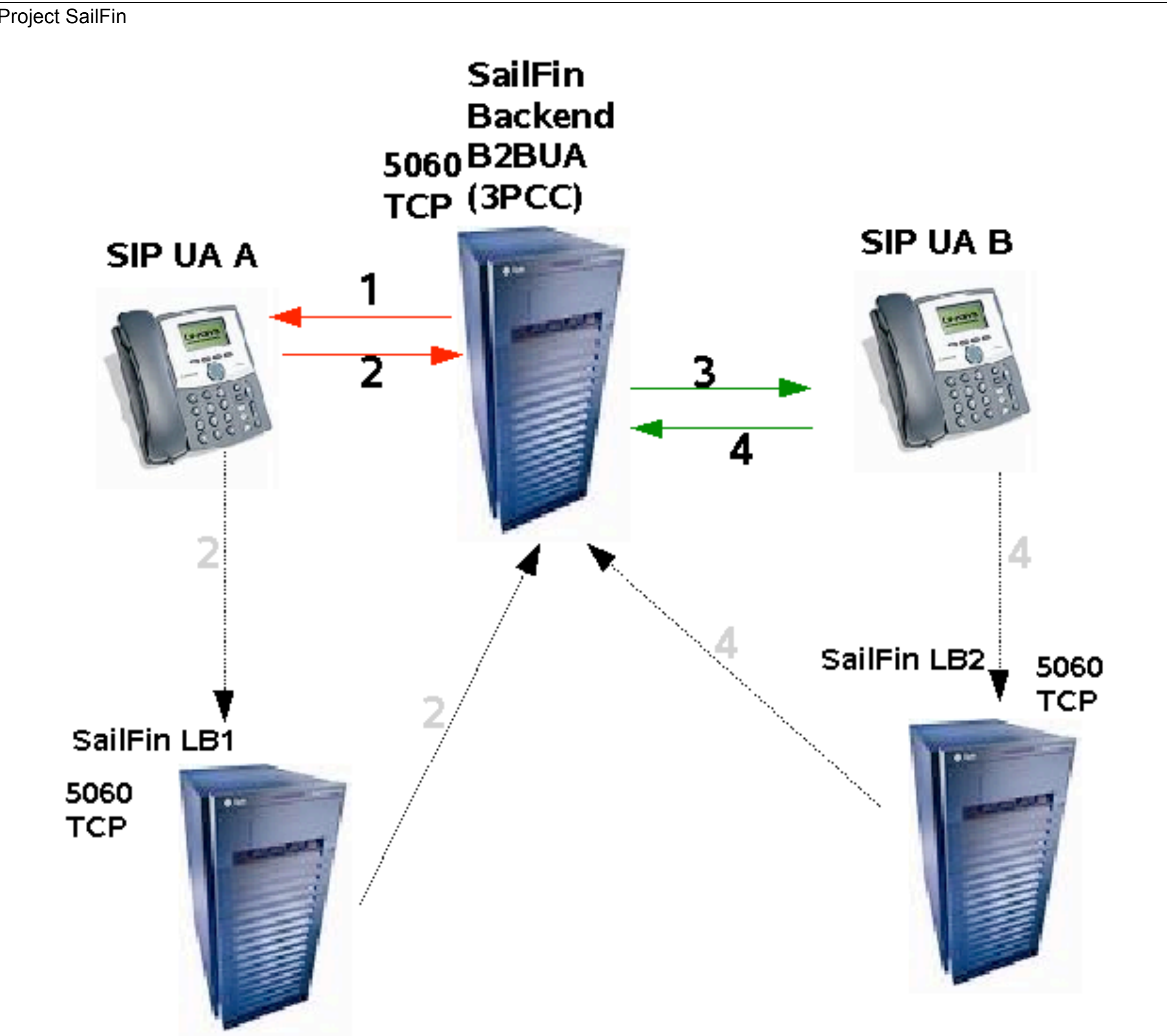
The SIP standard briefly defines a B2BUA as a logical entity that receives requests as a User Agent Server (UAS) and in order to respond to them, it acts as a User Agent Client (UAC) and generates requests. Additionally it maintains dialog state and must participate in all of the requests sent on the dialogs it has established. The standard defines it as a concatenation of a UAC and UAS and therefore doesn't provide additional definition for this entity.

2.1.4.1 Basic B2BUA (invite).



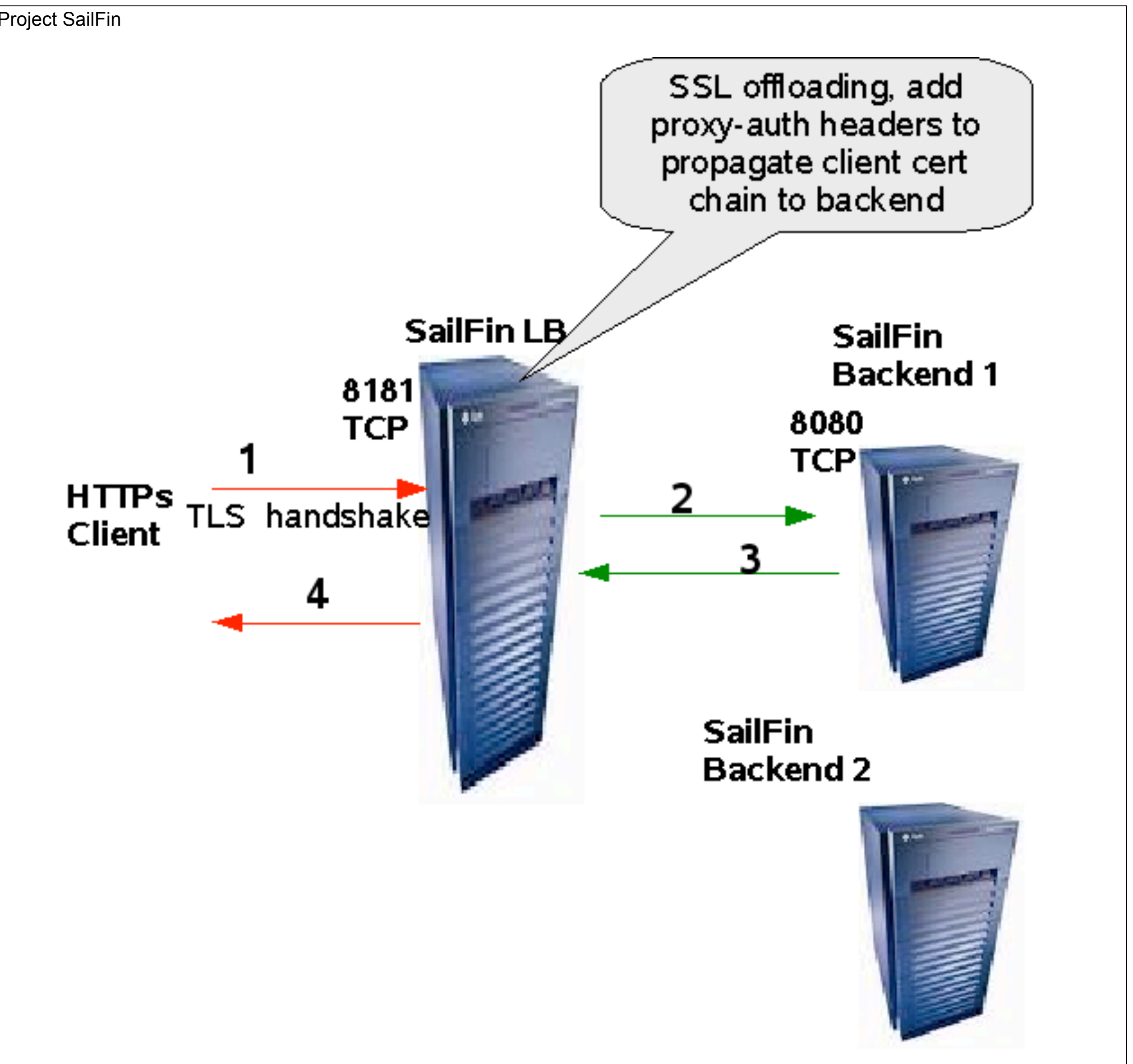
2.1.4.2 Third Party Call Control (3PCC):

A B2BUA may act as what the standard defines as Third Party Call Control (3PCC) and connect a call between 2 User-Agents.



Refer Converged Load Balancer functional specification [3], Appendix – SIP Use cases

2.1.5 HTTPs client sends a HTTPs request.



1. HTTPs client sends TLS (secure transport) request to the proxy. The TLS handshake is performed and the certificate extracted. this accepted channel is maintained (kept alive) until a response is received from the backend serving instance (or until a time-out).

5. The instance to which it needs to forward to is determined, this is done by the core converged load balancer, using the configured policy. The proxy does the header parsing and makes all the headers available to the core load balancer, which enables it to take a routing decision. The proxy forwards the request to the instance1 by opening a TCP channel to it. The proxy also adds some information to indicate that this message was a proxy by the LB. This would be using a private header field. Security information is propagated to the backend through the proxy-auth headers (Proxy-auth-cert, Proxy-keysize,

Proxy-ip). The headers are being used currently by native http load balancer to propagated certificate information to the backend, the same will be used by the proxy.

6. The remote backend instance responds back on the same channel created in Step 2.
7. The proxy picks the client channel (in Step 1) and sends the response back.

.*2.2* Use Cases – Startup/Configuration

1. SailFin instance started as both FrontEnd and Backed : Here the CLB and the proxy are enabled and the proxy goes through all the runtime use cases specified above.
2. SailFin instance is started only as a Backend: Here the CLB and the proxy are disabled and the instance does not perform any load balancing or proxying. The use cases are not valid.

2.3 Design

The current implementation of the connector-container stack has been designed with the intent of local termination (in the HTTP path) and thus lacks the proxying characteristics. By local termination we mean that a request (HTTP) that is received by the connector, is always passed on to the container with the intent that the processing of the request will take place in the container and the response would be returned through the connector. But, proxy throws up a new requirement, i.e a request that is received by a connector has to be forwarded to another instance, which would be serving the request and generating a response. In this case, the response might take a different path to the client rather than the same path through which the request flow happened.

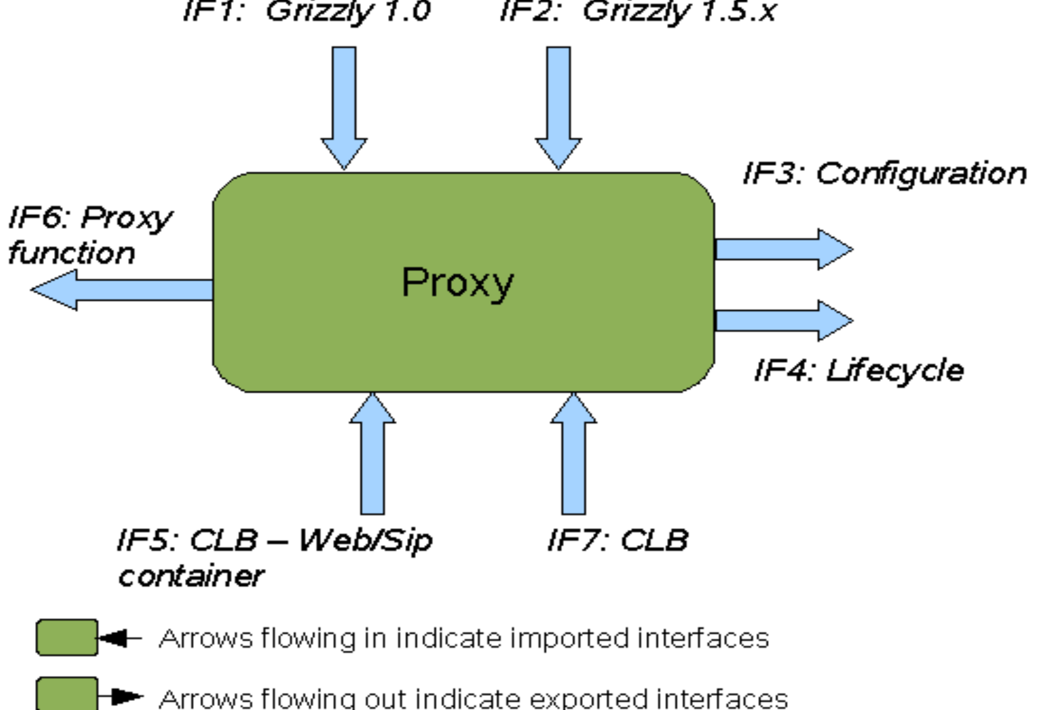
The proxy is implemented as a layer between the connector and the container, and in some way is also a logical extension of the connector's responsibility. The proxy functionality can be segregated based on how it affects the requests in the inbound path and the interfaces it provides to the converged load balancer to dispatch the requests to a remote serving instance.

In the request processing part (inbound), when a request is received from a client, the proxy has to parse (or partially parse) the HTTP/SIP request, so that the headers are available to the CLB to take a routing decision. Depending on the interception model of the CLB the request parsing may or may not be done by the proxy. For e.g if the CLB is implemented as a Catalina Valve (see Appendix) then the request parsing happens in the default connector-container path and there is no additional function that the proxy has to perform. But for performance reasons if we implement the CLB interception closer to the connector, then the request has to be parsed and assembled into

a data structure that is used by the CLB (see Runtime use case 3) . This proxy-CLB interface is dependent on the CLB implementation.. In the HTTP path (in sailfin) the grizzly 1.0 port unification mechanism will be utilized to perform the request interception and the parsing. . However in the SIP path, the parsing happens in the network manager layer. Also any logic that assumes that a response would be received after invoking the container API (local termination) has to be changed because when the request has been forwarded to another instance the response may not flow back on the same path. The request/response objects might have to be modified to supply adequate information to the proxy.

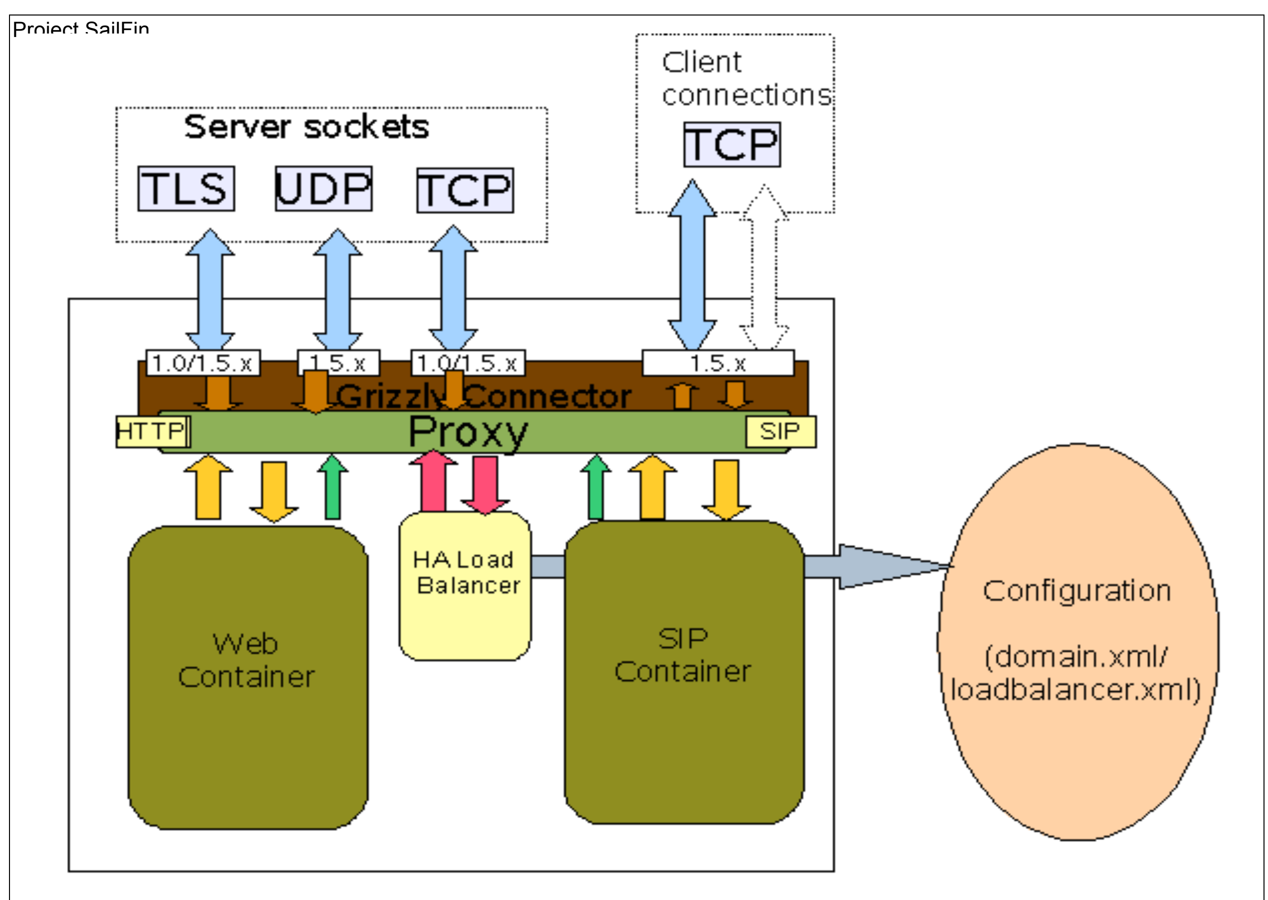
The implementation of the outbound (which forwards the request to another instance) would be using Grizzly 1.5.x NIO client APIs. The proxy also exposes a set of interfaces for configuration and Lifecycle management.

The following figure provides a view of the interfaces consumed/invoked by the proxy

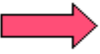


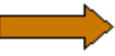
Interactions with other sub-systems

The following figure shows how the proxy is dependent/interacts with other modules and subsystems in sailfin.



Admin Interface

 HA CLB Interface

 Grizzly (1.0/1.5.x) APIs

JDK NIO APIs

Proxy Lifecycle

Web/Sip Container Interface

Proxy Lifecycle : The proxy is pluggable and can be enabled or disabled in the sailfin instance. Configuration information is conveyed by the container when it initializes the connector (grizzly). This mechanism will be different for HTTP and SIP containers because of the difference in interface with the Grizzly APIs (1.0 and 1.5). The proxy exposes an API with which the container

can determine if the proxy filter/handler has to be enabled in the connector. Also, the proxy functionality is configurative for an instance of sailfin and not per-container, i.e. the proxy cannot be enabled for the HTTP container (or a listener) alone or a SIP container only. In some deployment scenarios the proxy will also be aware if the HA CLB has been enabled/disabled in the request chain.

Administration Interface : The administration interface is used by the HA CLB to read the configuration file. The CLB uses the loadbalancer.xml to store its configuration, whereas the proxy uses the domain.xml.

Private Interface : The private request dispatching interface for the proxy would be through the CLB interface. The proxy ensures that the complete header parsing has been completed before invoking the container method. On the return path the HA CLB will set an “Endpoint” Data-structure as a thread local object which the proxy will use to identify the instance to forward the request to. The absence of such a header would mean that the request processing happened in the local instance and the response has to be returned.

2.4 Runtime

2.4.1 Use cases 1,2,5 described above.

Refer Converged Load Balancer specification for details [2]

2. Use case 3 described above.
The HTTP request is is sent by a http client to port 8080 in sailfin. The http listener in sailfin is activated by the Grizzly 1.0 connector by default. Proxy functionality uses the Port unification mechanism available in grizzly. The PU in grizzly allows the proxy to intercept the requests at the bytes level and take a load balancing decision at the earliest possible oppurtunity. Grizzly 1.0 PU pipieline invokes the LoadBalancerProxyFinder class. The job of the finder class would be to determine if the incoming stream of bytes is an http request and if so, whether the request needs to be processed by the local container or a remote sailfin instance. Also, to satisfy requirement 10, there has to be a mechanism to invoke a set of functional blocks before the routing decision is made for the received request. The proxy exposes an interface – HttpLayer , and all the interceptors that are configured in the dispatcher.xml that implement this interface will be invoked by the proxy in the same order in which they are configured. The onus of reading the dispatcher.xml and collecting the interception points and setting them on the proxy class lies with the startup glue code that exists in the Sip LayerHandler and SipServiceListener. So, the activity of collecting the interceptors and setting them on the proxy class has to be done much prior to the proxy initialisation. Please refer to converged load balancer specification for further details. The converged load balancer is configured as one of the interceptors in the dispacther.xml. The load balancing decision is made as part of the interceptor chain invocation. The HttpLayer interface (shown in Interfaces section) exposes one method which has to be implemented by the intercepting classes, and this method will be invoked by the finder on all the interceptors. The interface uses the grizzly Requet and Response objects because they were found to be lighter than the catalina request and response apis. Also, the interceptors require a higher level api to invoke methods like getHeader , setHeader and getParameters on the request objects. The Request object holds a lazy parsed input buffer for holding

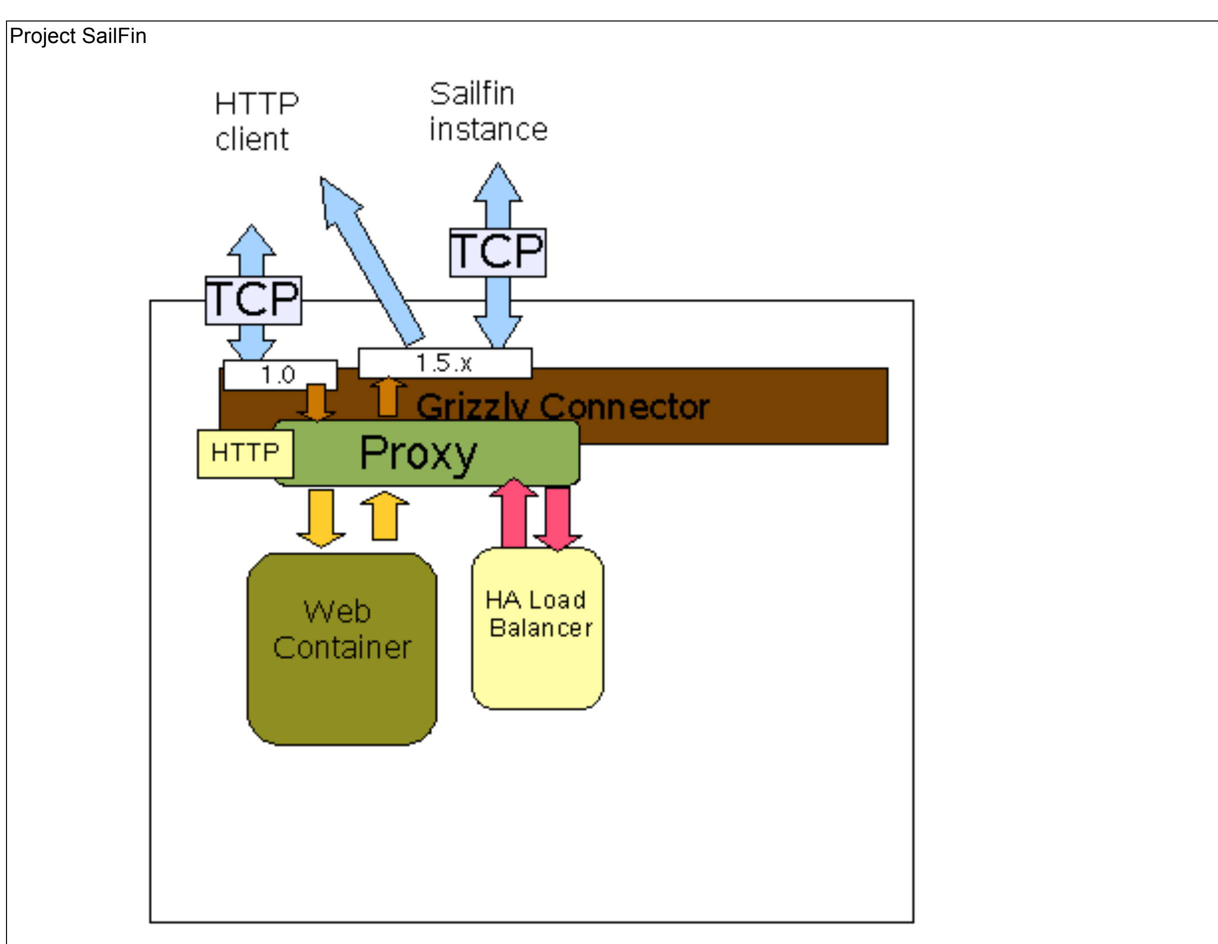
the request bytes , the header values are assembled into string objects only when they are requested by the interceptors. The invoke method returns a true if the next layer has to be invoked , and false if the request processing has to be terminated at this layer. The interceptors can invoke any getter methods on the Request object, but at this point in time its expected that they would only do a setStatus on a Response object. Though other APIs may work, they are not supported at this point in time. The load balancing decision by the converged load balancer interceptor (shown in figure below) is propagated back to the proxy though the setConvergedLoadBalancerEndpoint method in the request object. To use and proxy/clb specific api on the request object , the object has to be cast to the org.jvnet.glassfish.comms.clb.proxy.http.util.HttpRequest object. The Endpoint is a standard interface exported by the proxy (desribed in Interfaces section)

Once all the configured interceptors have been invoked, the finder examines the request object and retrieves the Endpoint object from the Request and checks if the request has to be sent to the local container (if the saifin instance is both a front end and backend, section 2.2 part 1) or routed to a remote instance. If its for a local instance the finder simple returns the protocol as http and lets the request to be processed as it were any other http request to the instance. If it has to be routed to a remote instance then the finder returns the protocol as “lb/http” which causes the PU pipeline to invoke the LoadBalancerProxyHandler. The finder also resets other data structures like ProtocolInfo so that it could hand over the already processed bytes to the handler. The finder also propagates the parsed request and the endpoint to the handler, so that the handler has sufficient information to forward the request. The handler is a lightweight object and it just invokes the proxy API to forward the request. The interfaces that are used by the proxy API to create client channels are Grizzly 1.x (x > 5) interfaces. So there is a handoff from 1.0 to 1.5,x APIs here. The handler also propagates the SelectionKey to the proxy API so that more data can be read from the channel and the response written back to it. Eventually when the backend instance responds back with data it is passed through to the client channel if its still open .

The figure below describe this use case in terms of the interaction between various sub-systems in the sailfin instance. It also describes the interfaces and their nature involved in this use case.

The flow chart describes the request flow through the various grizzly artifacts (depicted in green). The CovergedLoadBalancer has been depicted in yellow and proxy is only aware of the interface to the CLB.

The table 4.2.1 and 4.2.2 show the requests.



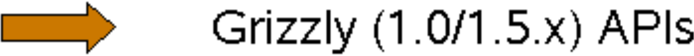
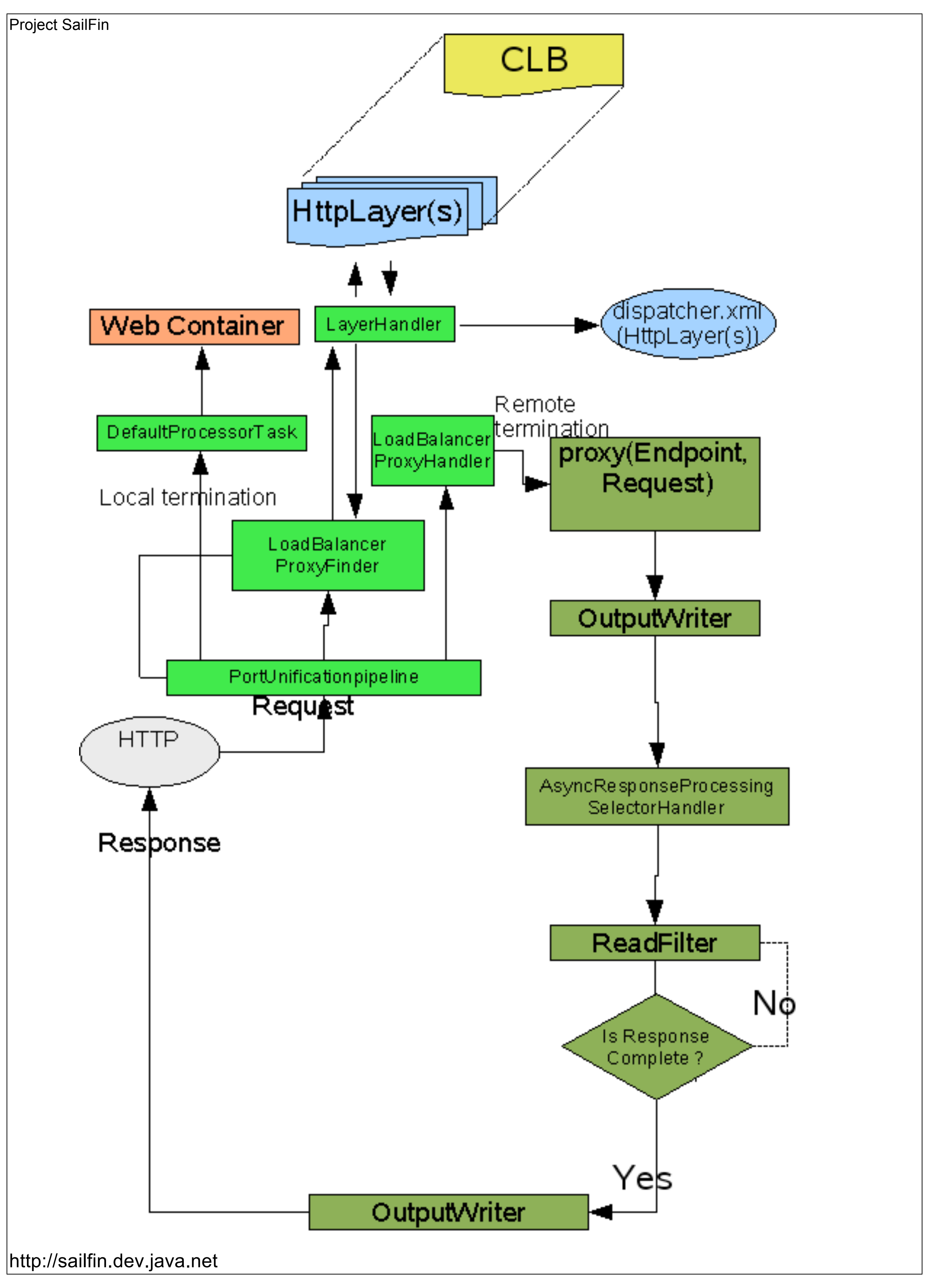


Figure 4.2.1



Project SailFin
CLB
HttpLayer(s)
Web Container
LayerHandler
dispatcher.xml
(HttpLayer(s))
Remote
termination
DefaultProcessorTask
LoadBalancer
ProxyHandler
proxy(Endpoint,
Request)
Local termination
LoadBalancer
ProxyFinder
OutputWriter
PortUnificationpipeline
Request
HTTP
AsyncResponseProcessing
SelectorHandler
Response
ReadFilter
No
Is Response
Complete ?
Yes
OutputWriter
http://sailfin.dev.java.net

Figure 4.2.2

Request at frontend	GET /SimpleWebApp/SimpleServlet HTTP/1.1 connection:keep-alive user-agent:Java/1.5.0_09 host:eas-v240-20.india.sun.com:9090 accept:text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2 content-type:application/x-www-form-urlencoded
Request at backend	GET /SimpleWebApp/SimpleServlet HTTP/1.1 connection:keep-alive user-agent:Java/1.5.0_09 host:eas-v240-20.india.sun.com:9090 accept:text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2 content-type:application/x-www-form-urlencoded felb:129.158.228.224:9090

Table 4.2.1

Secure Request at backend	GET /myindex.html HTTP/1.1 Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */* Accept-Language: en-us Accept-Encoding: gzip, deflate User-Agent: Java/1.5.0_09 Host: eas-v240-20:2323 Proxy-keysize: 128 Proxy-auth-cert: MIIDBTCCAq+gAwIBAgIBBTANBgkqhkiG9w0BAQQFADB9MQswCQYDVQQG EwJpbjEMMAoGA1UECBMDa2FyMQwwCgYDVQQ HEwNibmcxDDAKBgNVBAoTA3N1bjEMMAoGA1UECxMDandzMREwDwYDVQ QDEwhuYWdlbmRyYTEjMCEGCSqGSIb3DQEJAR YUbmFnZW5lZHJhLmprQHN1bi5jb20wHhcNMDQxMjE3MDYxNjUwWhcNMDUx MjE3MDYxNjUwWjBKMRAwDgYDVQQLEwdXZ WJUaWVyMQ8wDQYDVQQDEwZzYW5qYXkxJTAjBgkqhkiG9w0BCQEWFnNhb mpheS5peWVuZ2FyQHN1bi5jb20wgZ8wDQYJ KoZIhvcNAQEBBQADgY0AMIGJAoGBAMeuinauMVc9hE+FWHxtBKxqV4Mpo59 OV0F8DZeAbgNMNoX6JtJRCy+4s22mldW 2UDCpr14Ap8pkYo5TcFynh81K2TtsCuqitY1fOCUoVJObUgTOPoOLi5VJqKoUw5C T6s+TShQly6s3BRamr9eDbGrHpa u4MeTF8cqHgJZM5e7fAgMBAAGjggEHMIIBAzAJBgNVHRMEAjAAMCwGCWC GSAGG+EIBDQQfFh1PcGVuU1NMIEdlbmVyY XRlZCBDZXJ0aWZpY2F0ZTAdBgNVHQ4EFgQUY16JdeH4PIpiVx46hg/4V0I8iQc wgagGA1UdIwSBoDCBnYAUqi21noO6 8mbTlhgmKWDm/8Wqk/qhgYGkfzB9MQswCQYDVQQGEwJpbjEMMAoGA1UEC BMDa2FyMQwwCgYDVQQHEwNibmcxDDAKBgN VBAoTA3N1bjEMMAoGA1UECxMDandzMREwDwYDVQQDEwhuYWdlbmRyYT EjMCEGCSqGSIb3DQEJARYUbmFnZW5lZHJhLm prQHN1bi5jb22CAQAwDQYJKoZIhvcNAQEEBQADQQA1Sr2+NUmG/GRyf7lpvW J5r6gRNWqXPGeM2maox1Ce/e6lXSiEj VBjxawieYnJudCHPG4fo5b7yNUc+NX5RFJG Via: 1.1 proxy-server1 Connection: keep-alive felb:129.158.228.224:9090

Table 4.2.2

.

.

.2.5 Interfaces

Imported Programmatic Interfaces

IF7: CLB API	*[3]*	*CLB would pass the select endpoint as a datastructure on ThreadLocal. This is used by proxy to dispatch the request, in case a remote instance is selected. If local instance is selected, this programmatic artifact is not passed.*
IF8: Proxy-Heade r		*Protocol header added to a passthrough / proxied request. It would help instance identify that the request has already been proxied and needs to processed locally. CLB is not invoked .*
IF5 - proxy API		*The proxy method exposed to the CLB.*

Imported Programmatic Interfaces

IF1 : Grizzly 1.0	*[9]*	*API used by the Proxy for handling HTTP requests*
IF2 : Grizzly 1.5	*[10]*	*API used by the Proxy for handling outbound requests*

Imported Protocols

IF13 : HTTP 1.0/1.1	*[13]*	*HTTP Protocol*
IF14 : SIP 1.0	*[5]*	*SIP Protocol*
IF3 : TCP Protocol	*[7]*	*Grizzly supports TCP as transport protocol*
IF4 : UDP	*[8]*	*Grizzly supports UDP as the transport layer protocol.*

Exported interfaces

HttpLayer

```
package org.jvnet.glassfish.comms.clb.proxy.api;
import com.sun.grizzly.tcp.Request;
import com.sun.grizzly.tcp.Response;
import java.io.IOException;
public interface HttpLayer {
    public boolean invoke(Request request, Response response) throws Exception;
}
```

Endpoint

```
package org.jvnet.glassfish.comms.clb.proxy.api;
import java.net.InetAddress;
import java.net.SocketAddress;
/**
 * This is an interface for describing the structure and contact details of
 * a remote instance.
 */
public interface Endpoint {
   public SocketAddress getSocketAddress();
   public void setHost(String host);
   public InetAddress getIp();
   public void setPort(int port);
   public int getPort();
   public void setSecure(boolean secure);
   public boolean isSecure();
   public boolean isLocal();
   public void setLocal(boolean local);
}
```

3. Performance

We would like the performance cost due to the load balancing functionality in an instance to be as minimal as possible. The load balancing functionality (frontend of the sailfin instance) can be split into two logical parts , one being the proxy and other being the core LB that performs routing decisions. It is essential to measure the performance of the "proxy" component in the load balancer module separately. This would help us gauge the performance penalty incurred by the proxy alone and establish the tuning/sizing requirements of a system.

4. Management

1. Interfaces

5. Packaging, Files, and Location :

The proxy classes will be available along with the CLB jar file (clb.jar) that can be included in the sailfin classpath directly.

6. Quality

All the use cases mentioned in Section 2 must be tested. The SIP use cases have to be tested with different types of SIP requests like INVITE, CANCEL, BYE, REGISTER, ACK and OPTIONS. HTTP use cases have to be tested with 1.0 and 1.1 requests. It would be good to have tests that simulate failure after every step in the sequence diagrams described in the use cases. Explicit tests are required to simulate client time-outs. Failover scenarios, where a back end instance goes down, this would test scenarios where the backend's connection(s) with the proxy are terminated abnormally and would be a good test of how the proxy handles this situation. Few tests that exercise all possible configurations that can be applied on the proxy.

All the tests should be performed using the topologies that have been described in the Converged Load Balancer FS or the SailFin product FS.

7. Documentation Requirements

The proxy component is transparent to users/developers and is activated implicitly if load balancing is required by the sailfin instance. The proxy configuration that might be used for tuning/sizing has to be documented.

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

8. Open Issues

1. The proxy element and attributes in domain.xml needs to be finalized.

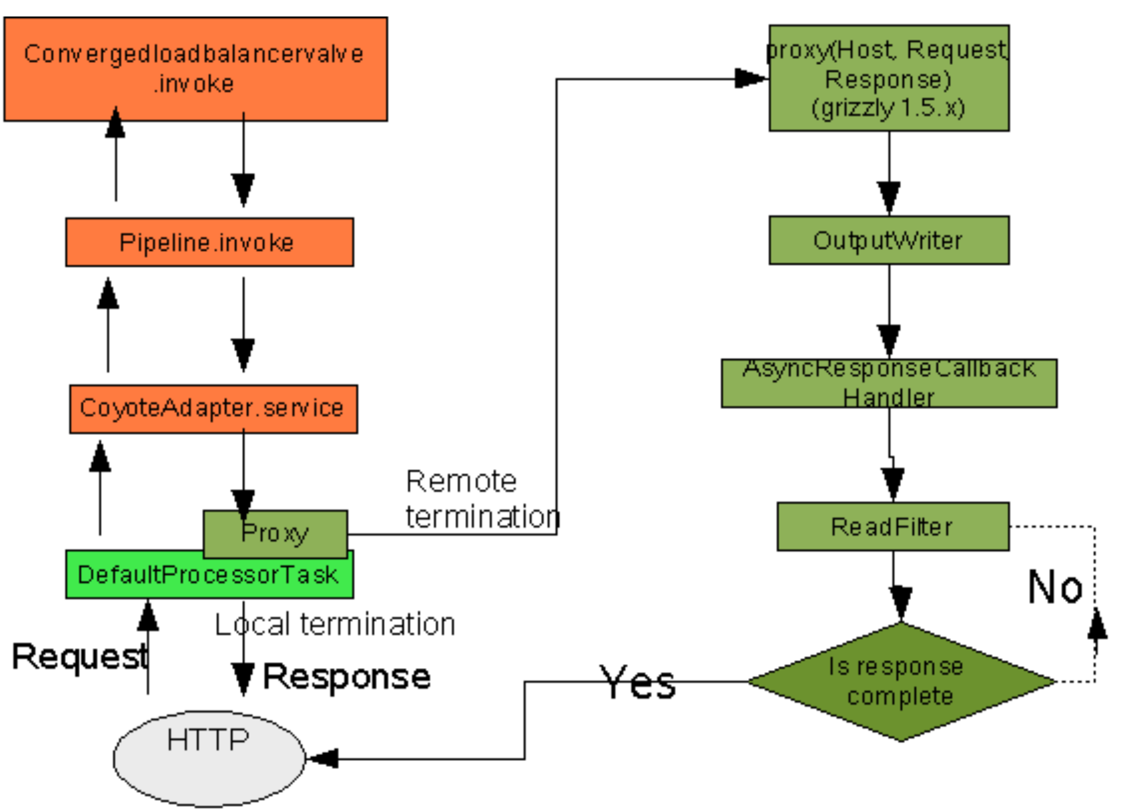
.Reference Documents

Reference Document	Location (URL)
[1] SailFin Architectural Overview Document	
[2] SailFin Requirement Document	
[3] Converged Load Balancer specification.	http://wiki.glassfish.java.net/Wiki.jsp?page=FunctionalSpecsOnePagers Converged Load Balancer
[4] SIP Servlet Container	http://jcp.org/en/jsr/detail?id=289
[5] Session Initiation Protocol	http://www.ietf.org/rfc/rfc3261.txt
[6] Domain DTD	https://sailfin.dev.java.net/documents/sun-domain_1_4.dtd
[7] TCP Protocol	http://www.faqs.org/rfcs/rfc793.html
[8] UDP Protocol	http://www.faqs.org/rfcs/rfc768.html
[9] Grizzly 1.0 API	
[10] Grizzly 1.5 API	https://grizzly.dev.java.net/nonav/apidocs/index.html
[12] Administration FS	
[13] HTTP 1.0/1.1	http://www.w3.org/Protocols/rfc2616/rfc2616.html

.Appendix

Alternatives that were considered for implementing CLB and proxy: (Runtime)

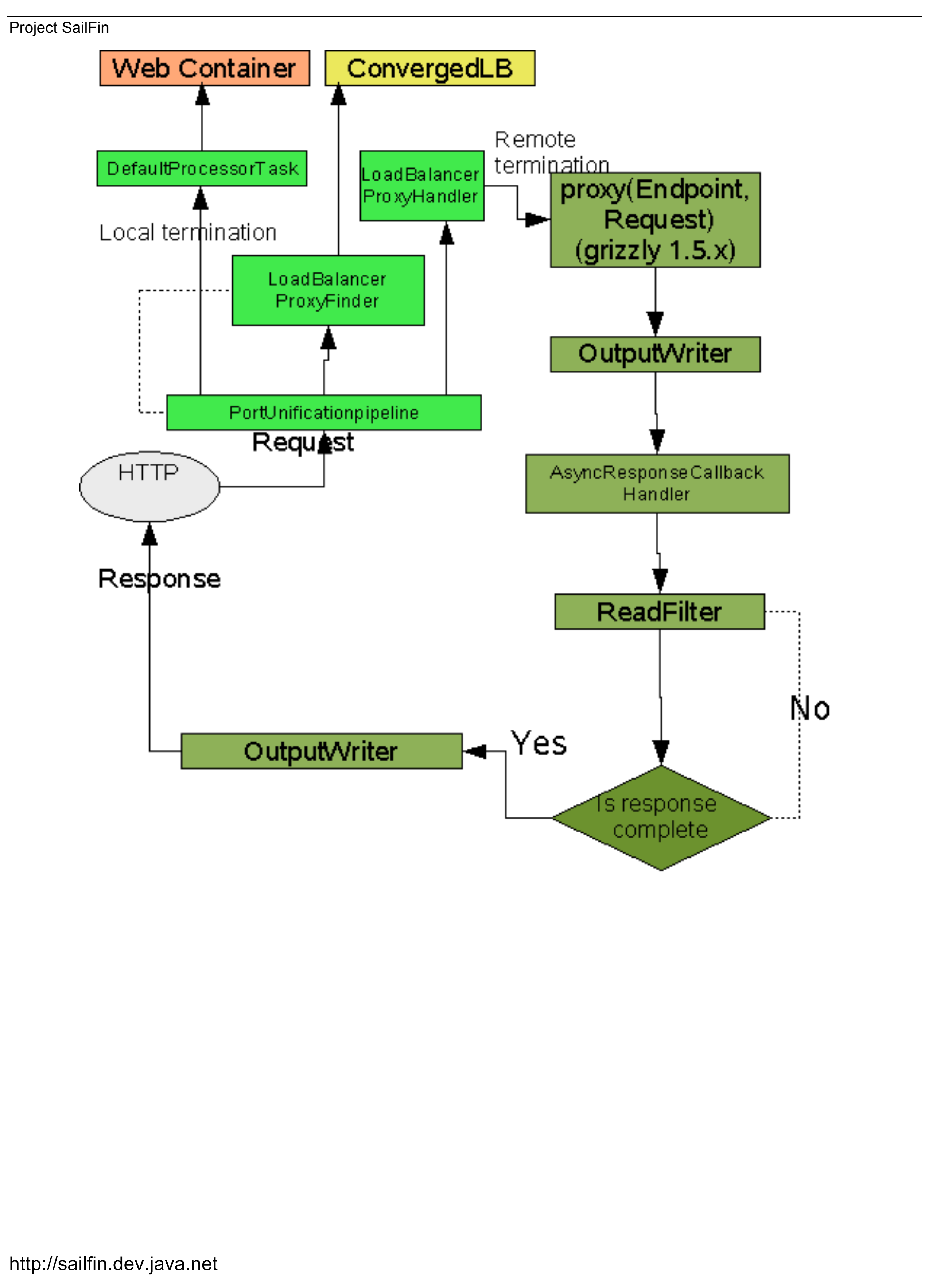
1. When interception is through Valve



The above implementation shows the working when the converged load balancer is implemented as a valve and intercepts the request at the container. In this case, the proxy is not present in the inbound path becausethe headers and request line are available to the CLB through the catalina interfaces - Request and Response. The CLB takes the routing decision and propagates it to the proxy through the Endpoint (thread local) interface. The proxy intercepts the outgoing request and then examines the Endpoint interface to determine the host of the remote instance and forwards the request to it.

2. When the interception is at the connector, but this approach does not allow users to plug-in any functionality

The HTTP request is received at 8080, and reaches the LoadBalancerProxyHandler (grizzly 1.0) that is registered by the proxy in the Grizzly 1.0 portunification pipeline. The http inbound path exploits the grizzly 1.0 port unification method. Port unification scheme allows users to plug-in a protocol and a handler to process the requests for that protocol. The LoadBalancerProxyFinder is where the byte parsing and CLB invocation happen, and the finder determines whether this is a local request or a remote request. If this is a local request the finder allows the request to propagate to the container, whereas for a remote request the LoadBalancerProxyHandler is invoked. The finder also propagates the parsed request and the endpoint to the handler, so that the handler has sufficient information to forward the request. The handler is a lightweight object and it just invokes the proxy API to forward the request. The interfaces that are used by the proxy API to create client channels are Grizzly 1.5 interfaces. So there is a handoff from 1.0 to 1.5,x APIs here. The handler also propagates the SelectionKey to the proxy API so that more data can be read from the channel and the response written back to it. Eventually when the backend instance responds back with data it is passed through to the client channel if its still open .The figure below describe this use case in terms of the interaction between various sub-systems in the sailfin instance. It also describes the interfaces and their nature involved in this use case.



Project SailFin
Web Container
ConvergedLB
DefaultProcessorTask
LoadBalancer
ProxyHandler
Remote
termination
proxy(Endpoint,
Request)
(grizzly 1.5.x)
Local termination
LoadBalancer
ProxyFinder
OutputWriter
PortUnificationpipeline
Request
HTTP
AsyncResponseCallback
Handler
Response
ReadFilter
No
Yes
OutputWriter
Is response
complete
http://sailfin.dev.java.net