Functional Specification for Sip Application Routing

Author(s): yvo.bogers@ericsson.com

1 Introduction

1.1 Revision history

Revision	Date	Author	Comments
0.1	2007-09-25	Yvo	First draft based on mail discussions with Per and Kristoffer
0.2	2007-10-01	Yvo	Processed comments Prasad, Kristoffer, Binod and Martien.
0.3			
0.4	2007-10-17	Yvo	Updated after phone conference and some deployment prototyping.
0.5	2007-10-31	Yvo	Updated after mails Roman
0.6	2007-11-12	Yvo	Distributing for review.

1.2 Terminology

The following table lists the most important terms and abbrevations used in this document.

Term	Explanation
AR	Application Router
DAR	Default Application Router
EDR	Early Draft
GUI	Graphical User Interface
Hot deployment	Refers to the process of putting a new application router into service in a running system.
In-flight request	An initial request which has been routed by the application router, but which has not yet completed the total application path yet.
JSR	Java Specification Request
SIP	Session Initiation Protocol
SPI	Service Provider Interface
TCK	Technology Compatibility Kit

2 Design Overview

2.1 Assumptions

At any point in time, a JSR289 compliant application server will contain *one* running (active) application router. This could either be the Default Application Router (DAR) or a third party Application Router (AR).

***Note**: this assumption contradicts earlier EDR consolidated feedback, which stated that*

multiple ARs could be loaded via some kind of DriverManager mechanism.

The DAR will be packaged with the SIP container, as a .jar archive similar to other application routers. It is deployed at server startup.

2.2 Use Cases

Application routing use cases can be divided into the following categories: packaging, deployment and configuration.

2.2.1 Packaging

The *packager* is the person responsible for packaging the application router software. He is the main actor involved in these use cases.

2.2.1.1 Use Case: package new AR version

An existing version of a particular AR is running on the server. Due to some bugfixes a new version of an AR needs to be deployed on a target system. The packager packages the new application router into a **.jar** archive and places it onto a location accessible by the deployer.

2.2.1.2 Use Case: package entirely new AR

An existing AR is running on the server. Due to a change in required functionality or business logic, an entirely new AR needs to be deployed on a target system.

From a packager point of view, this use case is no different from 2.2.1.1.

2.2.2 Deployment

The *deployer* is the main actor involved in these use cases. Deployment use cases are related to the process of installing new *applications* on the system or installing a new *application router*. Pluggable Application Routers could be supported by

- Update the dispatcher.xml file inside the sip-stack.jar file with a new classname for Application Router class, and restart.

- Support pluggable Application Router by implementing a Glassfish lifecycle module that would create an instance the ApplicationRouter and call ApplicationDispatcher.getInstance().setApplicationRouter(….). Both ApplicationRouter and ApplicationDispatcher are singletons.

- (**preferred**) Implement a new archive type for Application Router, and create a new deployer for the archive (through the deployment SPI). In deployment SPI terminology this is called ‘adding an Extension Module’.
  The deployer creates an instance of the ApplicationRouter and calls ApplicationDispatcher.getInstance().setApplicationRouter(….). Both ApplicationRouter and ApplicationDispatcher are singletons.

  The following classes are added to handle AR deployment.

```
com.sun.enterprise.deployment.backend.extensions.sip.ArArchiveDeployer
com.sun.enterprise.deployment.backend.extensions.sip.ArArchiveLoader
com.sun.enterprise.deployment.backend.extensions.sip.ArArchiveDescriptor
```

and the following code is updated to register the new Extension Module

```
org.jvnet.glassfish.comms.startup.lifecycle.SipContainerLifecycle.register
ExtensionDeployer()
```

The management GUI should be updated to support deployment of this new archive.

2.2.2.1 Use Case: start GlassFish server with default AR

Startup the server with only the DAR in place. The server creates the DAR instance. The DAR reads its initial configuration by reading the file pointed to by the system property `javax.servlet.sip.dar.configuration`. The configuration is cached in memory and is lost when the server is started.

TODO: which are allowed locations of the properties file? Will it be replicated? *Should* it be replicated?

2.2.2.2 Use Case: deploy functionally different AR

This use case takes place when an operator wishes to install an AR from a different vendor, or when a different solution is deployed on the server. A solution consists of a chain of applications and a new chain *may* require a new AR.

Startup the server with the existing AR in place. The server starts up and instantiates the (old) AR. Deploy a different AR into the system, using standard application server procedures. On GlassFish, this would mean dropping the AR.jar file in the autodeploy directory or using the commandline `asadmin` interface. The application server will instantiate the new AR and, from there on, use it for routing SIP traffic.

TODO: upgrade currently not supported. Server needs to be taken out of the cluster before upgrading the AR.

2.2.2.3 Use Case: undeploy AR

Take a running AR out of operation. The server will fallback to its default application router.

2.2.2.4 Use Case: upgrade AR version

Introduce a new version of AR into the system e.g. due to bugfixes. All new SIP traffic should be handled by new version. All existing traffic should be handled by new version as well. Same as 2.2.2.1.

2.2.3 Configuration

JSR289 only specifies how to configure a default application router, by defining a rather exotic file format for the dar.properties file (JSR289, appx. C). Configuration of any other AR is left up to the specific vendor.

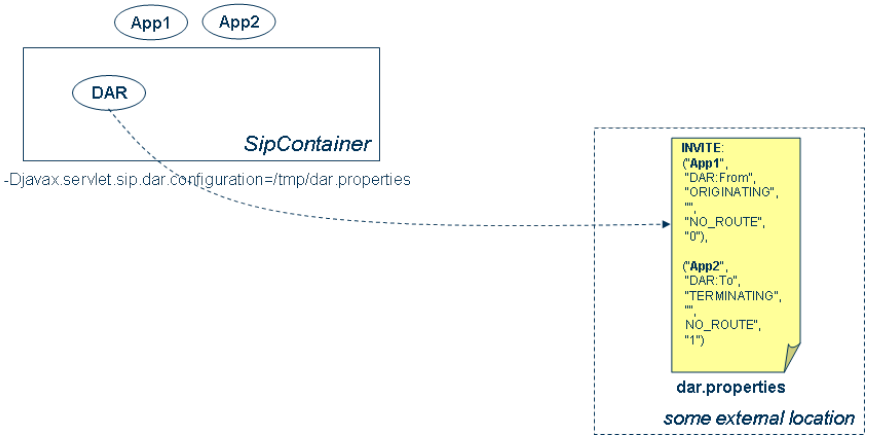
Configuration use cases are related to the process of changing something in the application router configuration. These use cases can be closely related to deployment use cases. For example, when a new application is deployed on the system, the

application router should typically be reconfigured to include this new application in the chain.

Several options exist for configuring an application router.

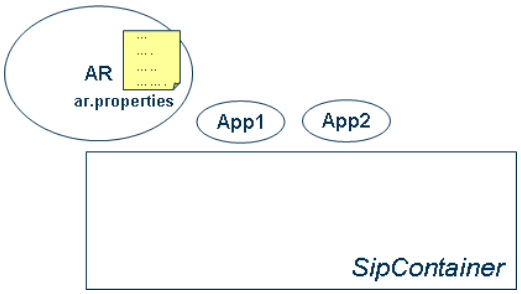
2.2.3.1 DAR properties on external location

This is what JSR289 currently specifies. Use a dar.properties file located elsewhere and point to it using a system property. Reconfiguration is possible by updating the dar.properties file and restarting the server.



2.2.3.2 AR and properties file packaged together as .jar archive

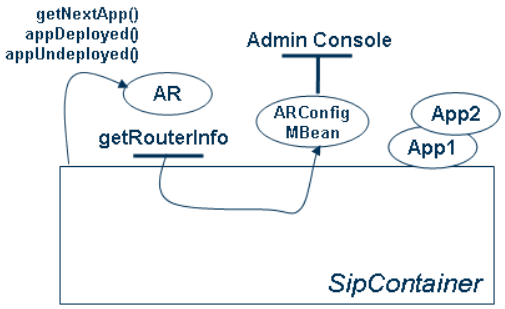
AR packaged as separate archive and properties file contained in it.



This could work, and the ar.properties file could be proprietary. Reconfiguration of the AR means unpacking the archive, updating the ar.properties and redeploying the AR archive.

2.2.3.3 AR packaged as .jar archive, configuration exposed in Admin console

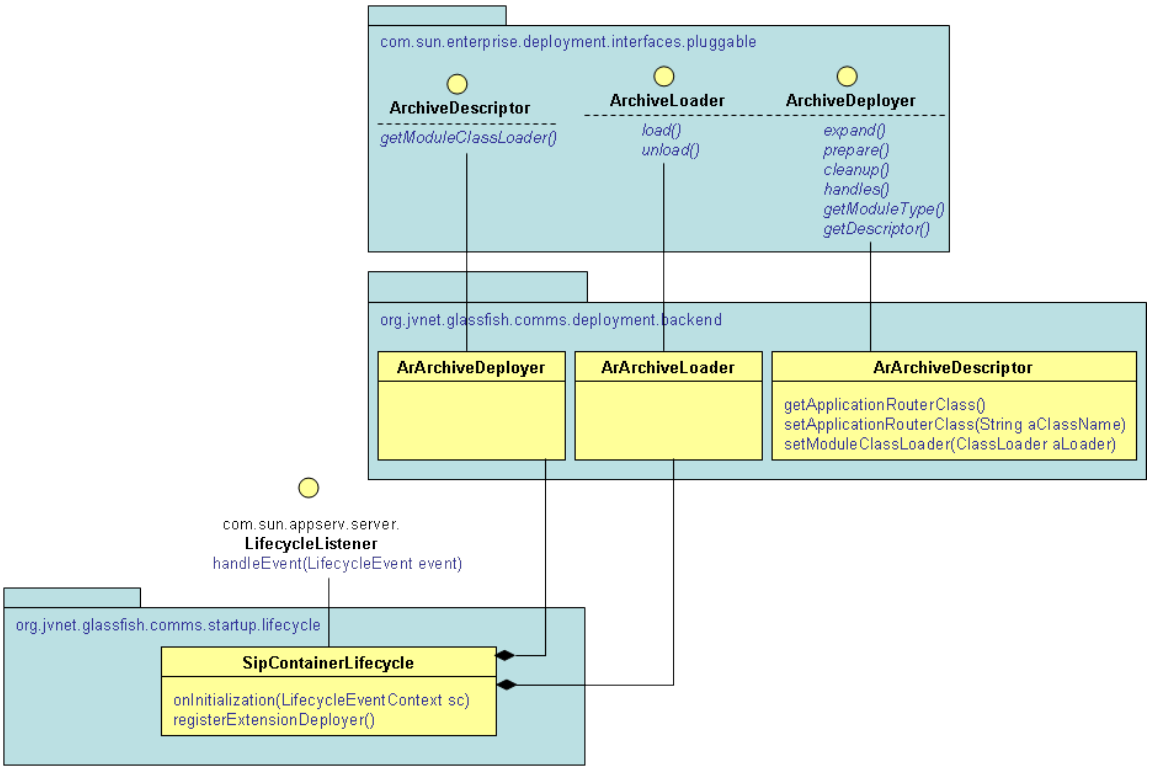
AR packaged as separate archive and AR configuration exposed via Mbean in the Admin console.



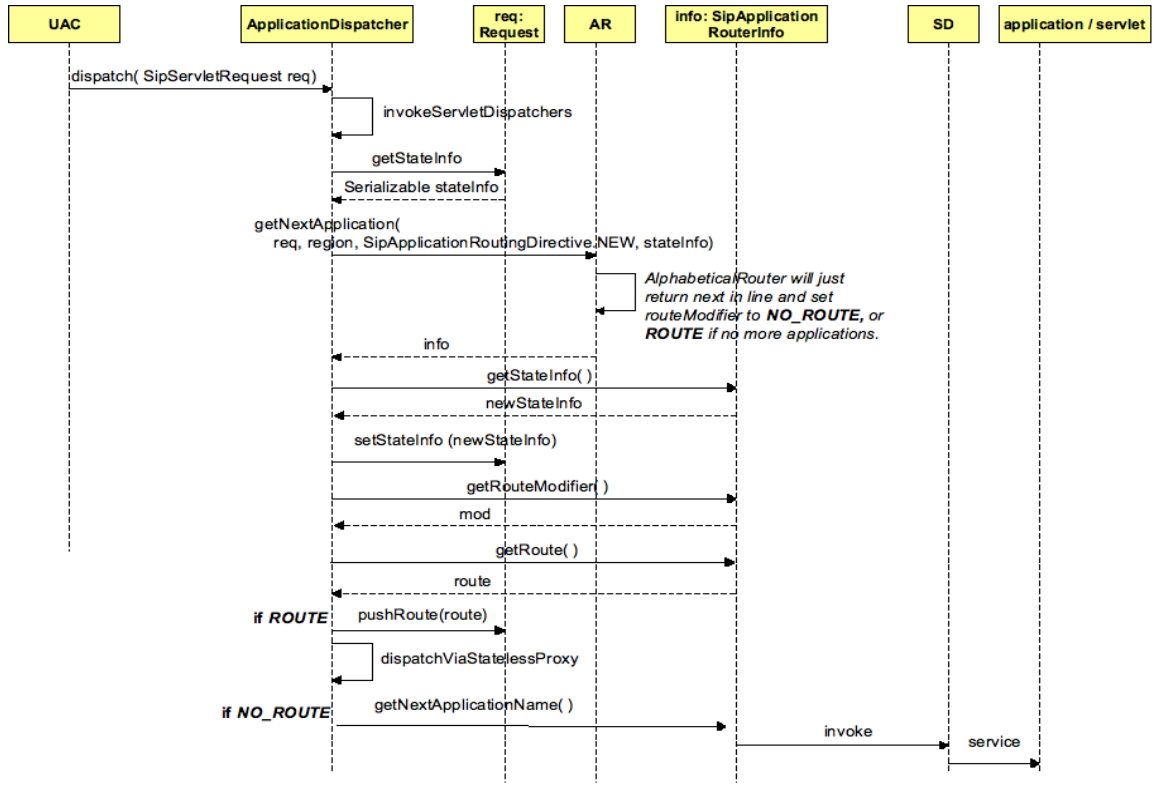
This solution offers the additional advantage of being able to visualize the AR configuration in the admin console. For example, deployed applications could be added visually to the AR configuration.

2.3 Package overview

The following picture illustrates some important interfaces and classes which are responsible for application router deployment.



The following picture shows the classes involved in the application routing process.



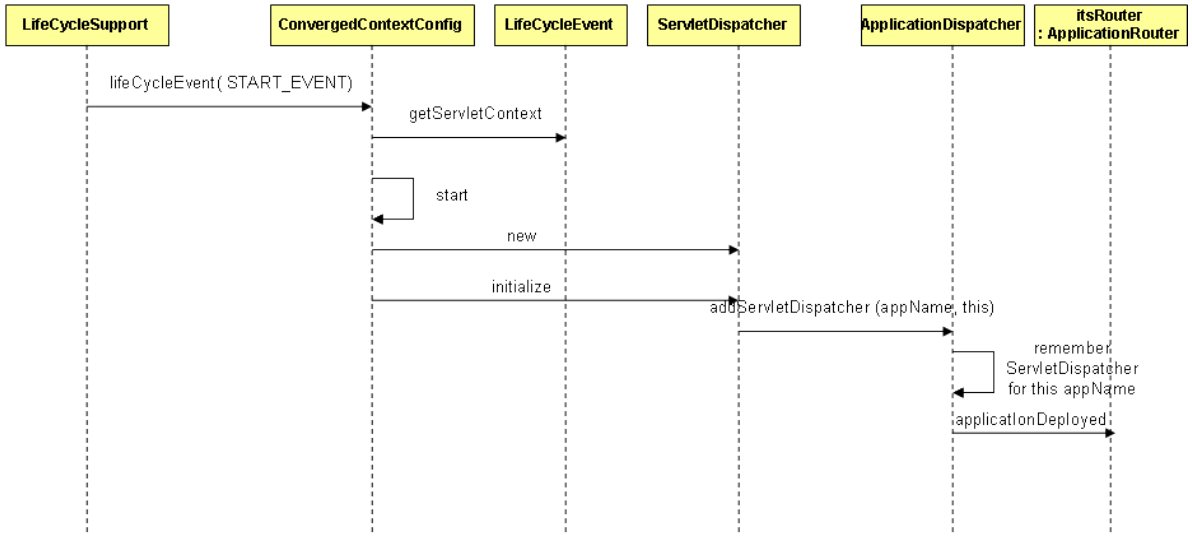
2.4 Dynamic behaviour

This chapter describes application router deployment and configuration in terms of their dynamic behaviour.

2.4.1 Application Deployment

When a new application is deployed (or undeployed, for that matter), the application server will do *two things*:

- fire a containerEvent. This event arrives at the ApplicationDispatcher. Currently this does nothing.
- fire a lifecycleEvent. This event arrives at the ConvergedContextConfig, which will install a new ServletDispatcher, resulting eventually in a call to *ApplicationRouter.applicationDeployed()*. This sequence is illustrated in the figure below.



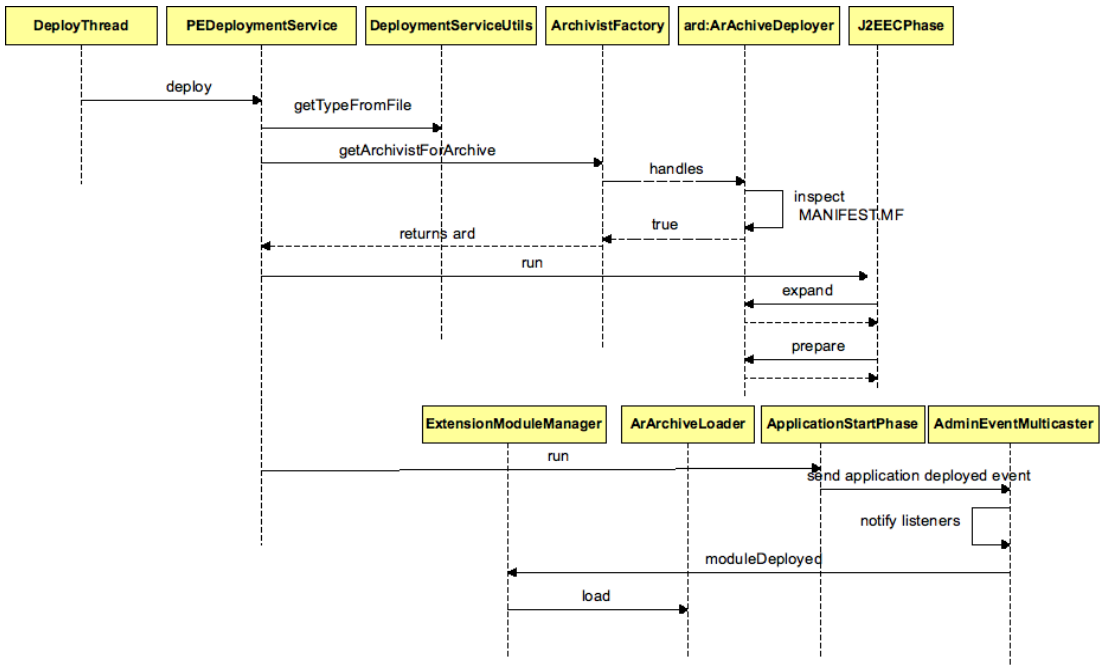
In either case the ApplicationRouter should refresh its configuration data.

TODO what to do with ongoing sessions? Subsequent requests or reponses within the dialog will not be able to follow same application path, if an application was undeployed somewhere in between.

2.4.2 Application Router deployment

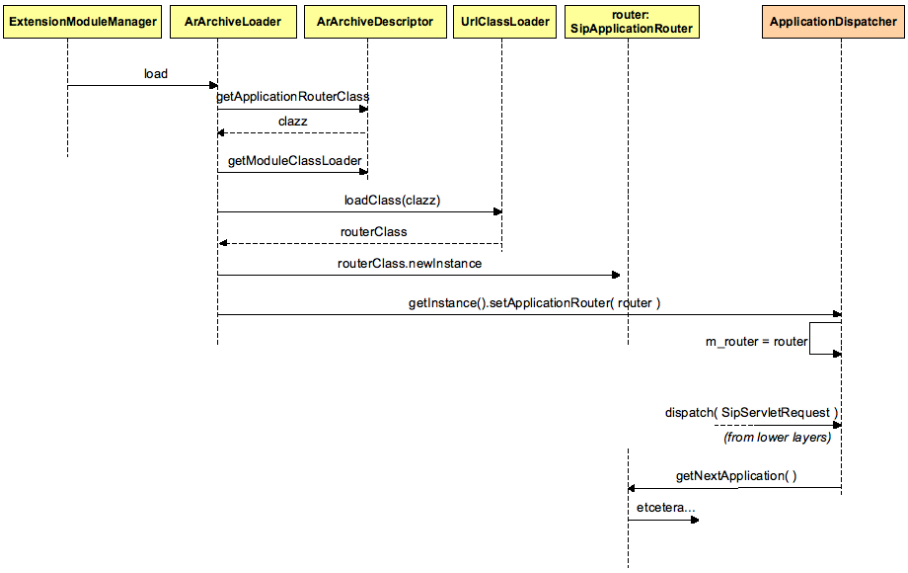
A deployment listener will listen for the application router to be deployed. A new ArArchiveDeployer and ArArchiveLoader is introduced into the GlassFish deployment framework. Whenever a .jar file is deployed, it will examine the contents of the jar and check the META-INF/Manifest.MF for the `Application-Router-Class` attribute. This attribute should list a fully qualified classname, for example:

```
Application-Router-Class: com.ericsson.servicelayer.applicationrouter.AlphabeticalRouter
```



During deployment of any application, the application server traverses a number of phases. As shown in the figure above, archive expansion and preparation happens during the J2EECPhase by calling the appropriate ArArchiveDeployer methods.

The actual loading of the ApplicationRouter class happens next, during the ApplicationStartupPhase. More details are shown in the figure below.



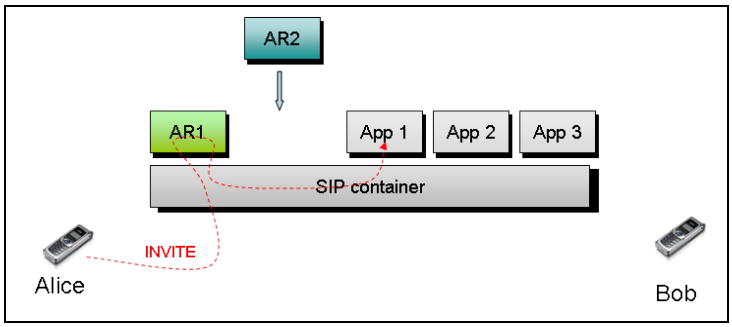
2.4.2.1 ClassLoading

The new ArArchiveDeployer will create an ArArchiveDescriptor during its prepare() method, to hold the classloader which is to be used to load the ApplicationRouter.

There may be classloading issues when an AR needs to access EJBs or SipServlets. This is not supported in the current version of the JSR289 specification.

2.4.3 Application Router rollover procedure

When deploying a new AR in a running system, an existing AR will always be present in the system (because each JSR289 compliant application server should package a DAR). The picture illustrates this situation with an example.



The SIP container was invoked for an initial request, in this case an INVITE. The request is being routed by AR1 to App1. Now a new AR2 is deployed. After App1 has dealt with the request the SIP container should call AR1 again to continue with the request. However, new initial requests arriving at the container should be routed to AR2. How will the ApplicationDispatcher know which AR to invoke? Several options exist to answer this question:

1. (**Preferred**) Don't support hot deployment. A **domain.xml** entry will instruct the SIP container which application router class to load. After deploying AR2, stop the container, update domain.xml and start the container. This may mean traffic loss.

   It was stated during last meeting that a rolling upgrade will require a server restart anyway. By hooking the AR deployment into the deployment framework, AR deployment will not differ much from the deployment of any other regular (e.g. converged) application.

2. Support hot deployment and maintain two AR references in the ApplicationDispatcher (old and new). The ApplicationDispatcher examines the stateInfo before calling ApplicationDispatcher.getNextApplication(). If stateInfo is null, this request originated from outside the container. In that case the new AR is called. Otherwise call the old AR.

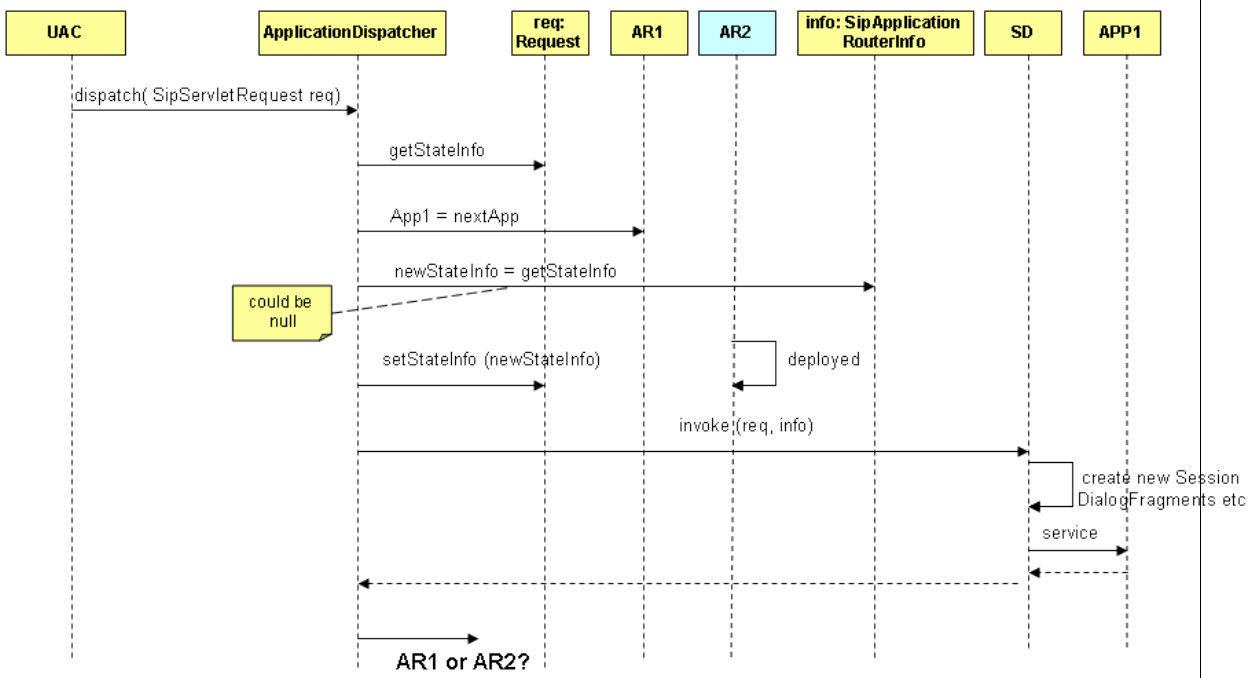
   TODO when to unload AR1? Manually?

3. Similar to option 2, but *don't call AR1 anymore*. As soon as AR2 is deployed, it will be used for all new SIP traffic. If stateInfo != null, send a *SIP 480 Temporary Unavailable* back up the application path and include a *Retry-After* header.

   A way to achieve this could be to set an 'upgradeServlet' in the config chain of the existing approuter as soon as a new Application Router is deployed. Some lifecycle module is listening for applications being deployed anyway, and translates this into a call towards the ApplicationRouter appDeployed() method.

   The ArArchiveDeployer is listening for new Application Routers being deployed. If this happens, a call towards the existing ApplicationRouter could be placed instructing it to "shutdown" or place a 'upgradeServlet' in its servlet chain. This upgradeServlet's only job would be to respond with a 480.

   TODO stateInfo inspection is not enough for this. How will the SIP application server know that a request with stateInfo != null was routed by AR1 and not AR2? Once AR2 has been invoked to handle an initial request with stateInfo==null, it will fill stateInfo with some new state. At that point, AR1 would be invoked again and we do not want that. Maybe we should include a reference to the handling applicationRouter instance in the request object, or in the SipSession ? What does that mean for memory usage and replication ?



2.4.4 Application Router Undeployment

Suppose an application router has been successfully upgraded and introduced into the system. Once all SIP traffic is being handled by the new application router, the old application router can be undeployed. The ArArchiveLoader will be called by the

deployment framework to *unload()* the application router. It will tell the ApplicationDispatcher to remove any references to the loaded application router.

The application dispatcher removes any references and falls back to the usage ot the DAR instead.

2.4.5 Configuration

Regardless of the type of Application Router (default or third-party), we should be able to tell it to refresh its configuration. It was suggested to add a method in the SipApplicationRouter interface *configurationChangedEvent* which passes the new configuration as a parameter, e.g. as an InputStream object. This suggestion did not make it to the JSR289 specification, but we could still add it as a proprietary method.

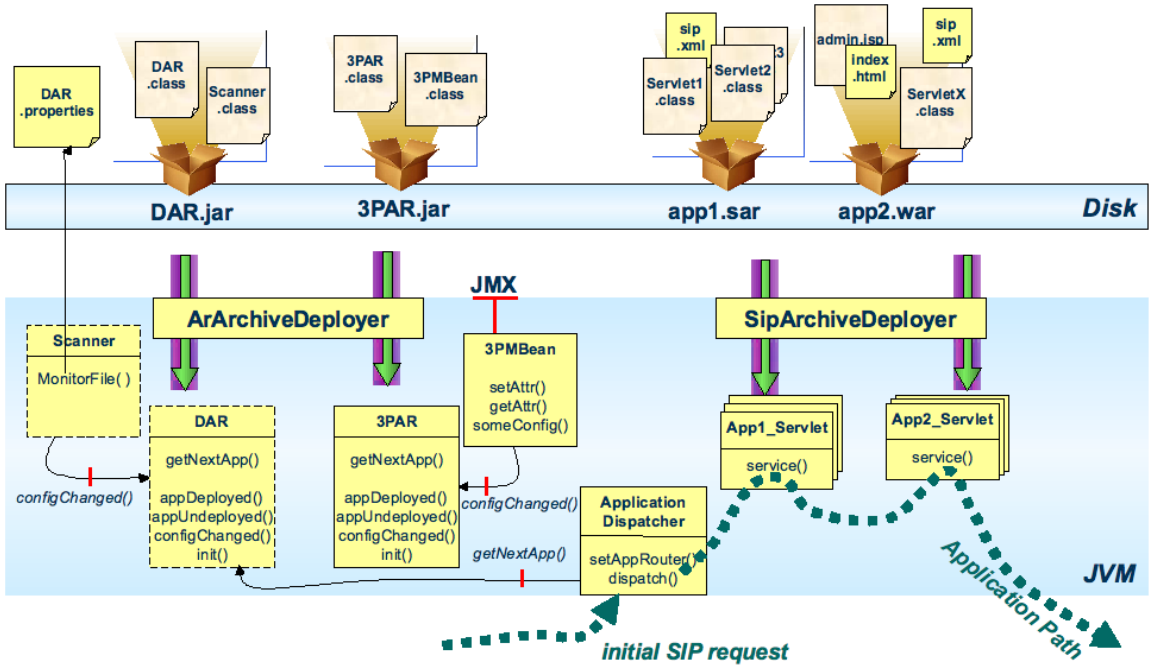
```
public interface GFApplicationRouter extends SipApplicationRouter
{
   void init(List<String> applications, InputStream initialConfig);

   void configurationChangeEvent(InputStream newConf);
}
```

How the InputStream object springs into existence is left up to the JSR289 compliant application server, i.e. it won't be standardized.

One default suggestion would be to include a configuration file in the META-INF directory inside the AR.jar . When the AR is deployed, the initial configuration is read, the InputStream constructed and passed as an argument to the GFApplicationRouter constructor. To reconfigure the AR, the config file would have to be changed, repackaged and redeployed to the server. A monitor, packaged with the AR.jar, would pickup the new config file and inform the ApplicationRouter.

Another option is to include a nofitication listener in the AR.jar which could prepare the InputStream for the ApplicationRouter. The notification listener would be listening to some proprietary AR configuration framework; this could be a simple MBean, or some other external system.



2.4.6 Proxy AR

This originally was an idea by Roman Levenshteyn:

> *I have implemented something called ProxyAR. This one does nothing on its own. Instead it reads the system property with the name of the real class implementing the real AR and creates a real AR. Then all ProxyAR invokations are delegated to the real AR invocations. This approach has some nice features:*
>
> *no need to update the standard JAR289 implementation files every time you introduce a new AR, as it is required now*
> *ability to change the AR on the fly, since only ProxyAR does exist from GF's point of view*
> *ProxyAR can be extended to delegate to different underlying ARs depending on the request or some sort of policy*
>
> *The problem so far is that the real AR class is supposed also to be placed in the GF/lib, because otherwise the classloader cannot find it. I'd like to have the ability to put the AR implementation at least into domains/domainX/lib. But for that the classloader issue should be solved somehow. Once it is possible, we can have a per domain AR.*

Great idea, just specify proxyAR in dispatcher.xml . Deployment of a new AR then boils down to setting a system property and placing the new AR.jar in de GF lib directory.

2.4.7 Food for thought...

This section is intended to get an idea about *ApplicationRouterState*. If an initial request enters the container, it will be routed by the Application Router to the first application in the configured chain. At this point, any number of situations could occur:

Application acts as UAS
Application chain is terminated, application path established so far is used to route responses and subsequent requests.

Application acts as a non-forking Proxy
The application chain continues. The Application Router is invoked again to determine the next application to invoke.

Application acts as a forking Proxy
The application chain continues. Let's say the application sets up only two forks. The Application Router getNextApplication() is called twice. How does the application router know which is the first fork and which the second? What if the first fork requires a need for a different application chain than the second.
It is most clear in this scenario that there is a need for *ApplicationRouterState*. The AR should know “where in the chain are we”.

Application acts as a B2BUA
Application chain selection starts anew. However, the Application Router is not in the same state it was in when the initial request came in.

Generally speaking, each application in a chain may traverse any number of states, choose a certain *outlet* when it has finished processing the request (or response), and produce a certain amount of *output data* for the next application in the chain.

An application router should be responsible for investigating which outlet a previously selected application has chosen, and tell the container which application to invoke next based on that information. It should also instruct the next application where to store its output data.

Example:
Application 1 is a presence application, application 2 is a call diversion application, application 3 is a call setup application, application 4 is a charging application.

Application 1 is able to produce output data based on incoming SIP request and some proprietary subscriber database.

Application 2 will divert a call, based on the presence of the called party and some proprietary subscriber data.

Application 3 will just proxy a call to any number it is instructed to proxy to.

Application 4 will write a call detail record which contains the postal address of the charged party and the duration of the call.

This example requires that the application router :

is invoked when an initial SIP request comes in;
prepares some kind of ApplicationChainContext object / EJB
points the input for the Presence application to “request.ToAddress”
points the input for the Call Diversion application to the “context.PresenceOutput” and
points the output for the Call Diversion application to “context.BNumber”
points the input for the Call Setup forking application to “context.BNumber”
points the input for the Charging application to “request.FromAddress”
returns the Presence application as the next app to invoke. stateInfo contains.... what does it contain?

We also require some terminology that all constituent services should be able to use, e.g. “calling party” , “number” , “URI”, “busy”, “location”, “time”, “greeting” , etc.

We want the constituent services to be loosely coupled, i.e. not be aware of eachothers internal workings, but it should be ok to implement the logic “if *calling party* is *in meeting*, divert to this *number*” within a component. Even though the latter suggests a coupling of presence and call diversion, it in fact decouples them.

3 Quality and Availability

-

4 Performance

The important thing is that an AR does not become a performance drain. Actually the throughput performance of the entire SIP container depends to a large extent on how fast the AR will cough up the next application to invoke.

It is therefore recommended that every AR refreshes its router info whenever a notification about change in configuration occurs. It should not have to retrieve this information from a file at call setup for example.

5 Management and Monitoring

Admin GUI extensions needed? Currently: no, since AR configuration is not standardized.

6 Formal Interfaces

Dar.properties (see JSR289 appendix C) needs to be available, and a system property needs to be added to the domain.xml as specified in JSR289:

```
javax.servlet.sip.dar.configuration
```

7 Packaging, Files, and Location

The Default Application Router code will be checked into CVS with the SipContainer. It should automatically be deployed by the container at startup. To configure the DAR, a properties file is used as defined in the JSR 289 spec.

Any other application router which is to be deployed later should be packaged as a .jar archive. The .jar archive contains the class implementing the SipApplicationRouter interface, as well as a properties-file in the META-INF directory, with some proprietary format. An example of such a application router will be stored in CVS under the sample applications directory.

Furthermore, a scanner should be included within the application router archive to monitor those files. This could perhaps be the ArArchiveLoader itself, or otherwise some dedicated monitor object. As soon as the timestamp of these config files changes due to editing, this monitor would read the new file and call configurationChanged() on the ApplicationRouter, passing in the config file.

TODO: Any configuration GUI needed for configuring the DAR properties?

TODO: TCK contains a set of applications and the DAR. What exactly does that imply for Glassfish?

8 Documentation Requirements

This FSD, javadoc, and probably some kind of programmer's guide.

9 References

1	Functional Specification for Proxy (Converged LoadBalancer)
2	JSR 289 EDR
3	JSR 289 Consolidated feedback
4	Functional Specification Description for Deployment SPI

10 Open Issues

Sailfin issues #123, #154, #158, #177

Classloading of the Application Router class. is it done correctly?

Scan this document for TODO tags ;)