Implementation of SIP Servlet User Policy

Sangho Shin
Columbia University
Department of Computer Science
ss2020@cs.columbia.edu

December 2002

Contents

NOTICE

This document is written under assumption that readers are familliar with SIP(Session Initiation Protocol) and SIP Servlet API Sepcification Version 1.0. Therefore, this document dose not describe SIP(Session Initiation Protocol) or SIP Servlet API Specification Version 1.0 presicely.
If you are not familliar with SIP or SIP Servlet API, please read RFC 3261 for SIP and SIP Servlet API Version 1.0 Draft 0.51 or 0.7, first.
And, SIP Servlet API Version 1.0 is very similar with Java Servlet API, and some classes uses the interfaces in Java Servlet API 2.2. So, I recommend that readers should read Java Servlet API Specification.

Chapter 1

Introduction

1.1 User policy

Policy is SIP transaction handling. That is policy decides how to handle SIP requests and responses. Low-level policy is transaction policy. This policy controls the modules standard handling of a transaction. High-level policy is user policy, which implements user specific features, and this user policy is implemented by SIP CGI, CPL, SIP Servlets. I implemented SIP servlet user policy in this research.

1.2 SIP servlet

A SIP servlet is a Java-based application component, managed by a container, which performs SIP signaling. SIP servlets can inspect and set message headers and bodies, and they can proxy and respond to requests and forward responses upstream.
SIP servlets have ready access a wide variety of APIs, directories, databases, CORBA, the Java Media Framework, etc. and they can reuse Java security infrastructure.

Currently, two versions of SIP Servlet API are published. The first version was published in 1999, and this is not standard version. The second one was published in May 2002, which is a standard version (version 1.0 Draft 0.51). This implementation is based on SIP Servlet API Version 1.0 Draft 0.51.
When I was implementing this version, SIP Servlet API Version 1.0 Draft 0.7 was published. In this version, some classes and functions are appended. However, they are not included yet in this implementation.

Chapter 2

Architecture

SIP servlet user policy mechanism is composed of mainly three parts, policy_core, sipservlet user policy handler, ServletEngine. Policy core is a mechanism for handling various user policies, and sipservlet user policy handler is interface between sipd(more exactly, policy_core) and ServletEngine, and ServletEngine is an engine for SIP servlets.

2.1 User policy

Policy core invokes policy functions and provides the basic state machinery for policy invocation.
When SIP message arrives at sipd, sipd checks user policy and gets policy_core start user policy handling. Policy_core calls the following 6 functions.

```
user_policy_ret (*init)(request_t *r, const policy_info_t *info, int status);
user_policy_ret (*handle_initial_request)(request_t *r);
user_policy_ret (*handle_subsequent_request)(request_t *r, message_t *m);
user_policy_ret (*handle_response)(request_t *r, message_t *m, branch_t *branch);
user_policy_ret (*timeout_expired)(request_t *r);
user_policy_ret (*cleanup)(request_t *r);
```

Every user policy handler should implement above 6 functions. Please refer to 'The CINEMA LIBSIP policy API (Jonathan Lennox)' for specific architecture.

2.2 SIP servlet user policy handler

The sipsevlet user policy handler implements the following 6 functions.

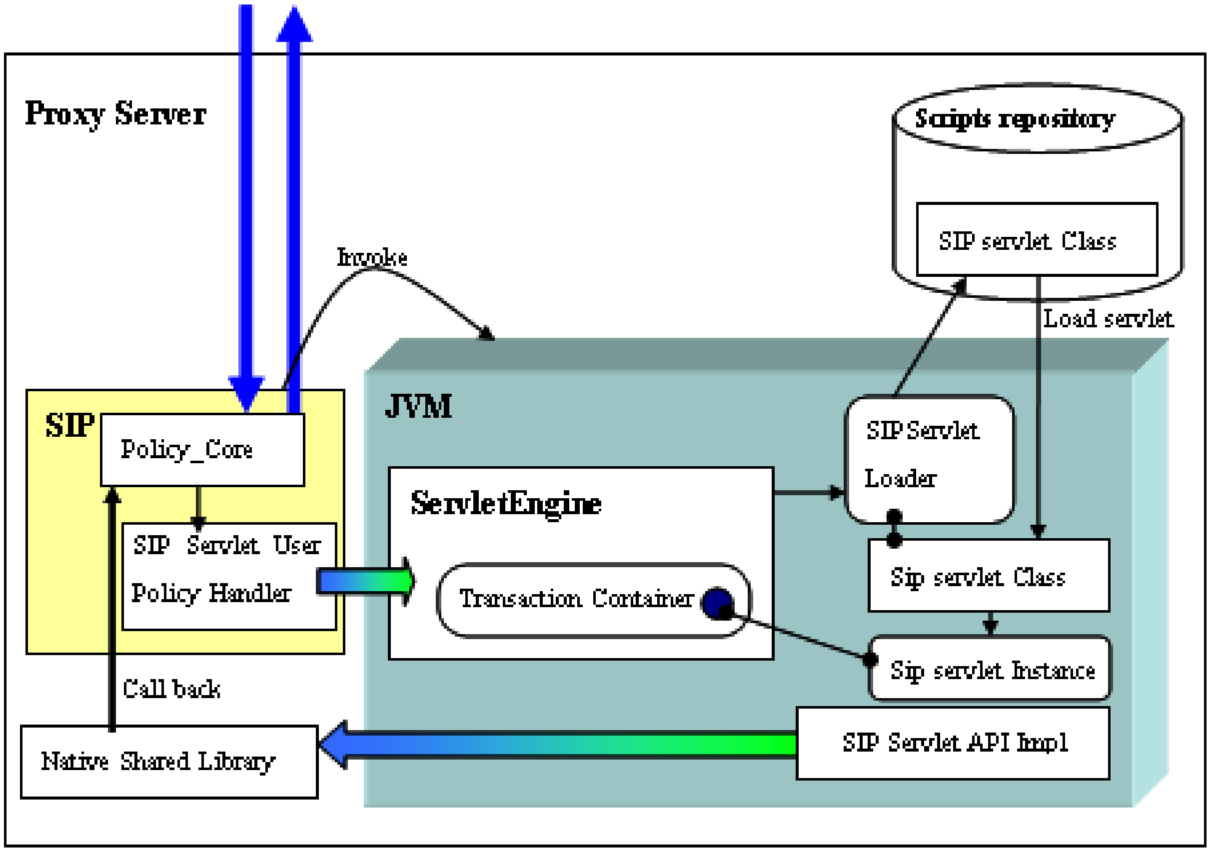


Figure 2.1: SIP servlet user policy architecture

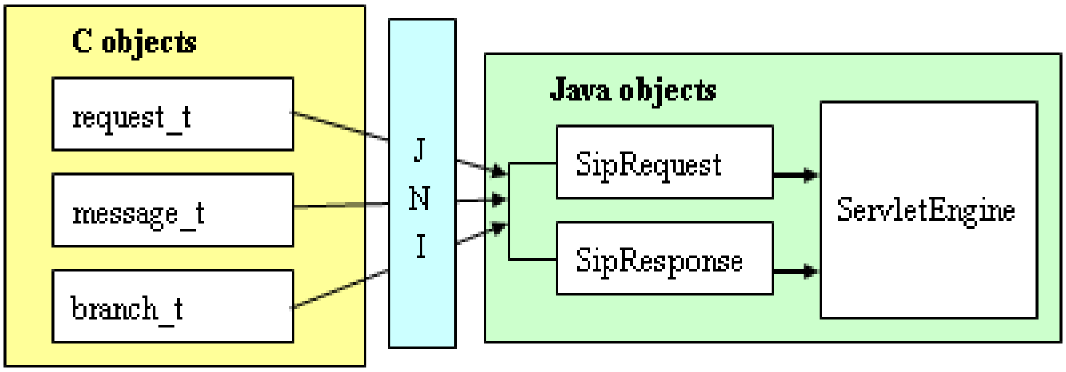


Figure 2.2: SIP message objects conversion

```
user_policy_ret sipservlet_init(request_t *r, const policy_info_t *reg, int status);
user_policy_ret sipservlet_handle_initial_request(request_t *r);
user_policy_ret sipservlet_handle_subsequent_request(request_t *r, message_t *m);
user_policy_ret sipservlet_handle_response(request_t *r, message_t *m, branch_t *b);
user_policy_ret sipservlet_timeout_expired(request_t *r);
user_policy_ret sipservlet_cleanup(request_t *r);
```

`sipservlet_init()` function is called immediately after `execute_policy()` is invoked. It constructs `user_policy_info` object. `user_policy_info` contains the callback function pointers for native shared library and the original request object pointer. This `user_policy_info` is stored in `request_t` object for later use.

```
struct sipservlet_info {
  script_t *script;
  void (*send_request_to)(request_t *r, const uri_t *uri, message_t *m,
        void *policy_info, policy_type type);
  void (*send_new_response)(request_t *r, int status, const char *reason);
  request_t *original_request;
};
```

After constructing `user_policy_info`, it constructs `SipServletResquest` object(Java) with `request_t` using JNI, and constructs servlet class path with script info, and all these objects are passed to ServletEngine by calling `init()` function of ServletEngine.
`xxx_request()`, `xxx_response()`, and `xxx_cleanup()` functions constructs `SipServletRequest` and `SipServletResponse` objects and call mapping function of ServletEngine.

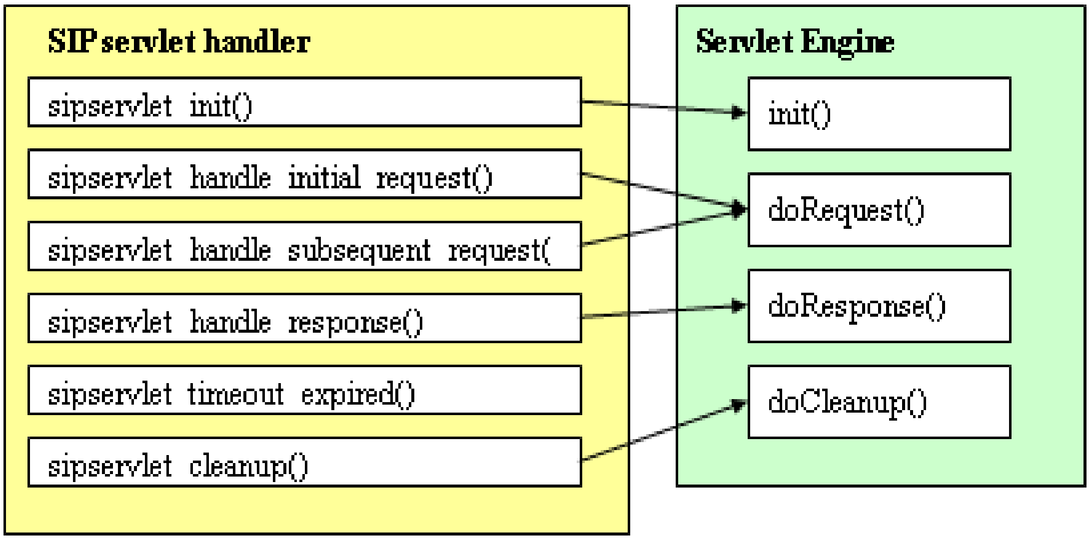


Figure 2.3: Function mapping between SIP servlet handler and Servlet Engine

2.3 Servlet Engine

Servlet Enigne loads SIP servlets, and invokes functions of SIP servlets and maintains states of SIP servlets. To maintain states of each SIP servlet, Servlet Engine maintains session container (Hashtable).
`SipSession` object has a `SipServlet` object as its listener, and the `SipSession` object is included in a `SipApplicationSession` object. According to the specification of SIP Servlet API 1.0, `SipApplicationSession` may have multiple protocol sessions such as `SipSession` and `HTTPSession`. Therefore, when multiple SIP servlets are registered, all their `SipSession` objects are included into a `SipApplicationSession` object.
Whenever new request arrives, instances of registered SipServlets are created and they have own `SipSession` object, and all the `SipSession` objects are added into one `SipApplicationSession` object. All the `SipApplicationSession` objects are added into the session container.

2.3.1 Servlet Mapping

Actually, all SIP servlts are not invoked. Servlet mapping conditions, which are defined in SIP servlet descriptor, decide whether the SIP servlet will be invoked or not.
ServletEngine reads and parse servlet mapping conditions from descriptor, and checks if the request satisfies the conditions, and the only SIP Servlets that have

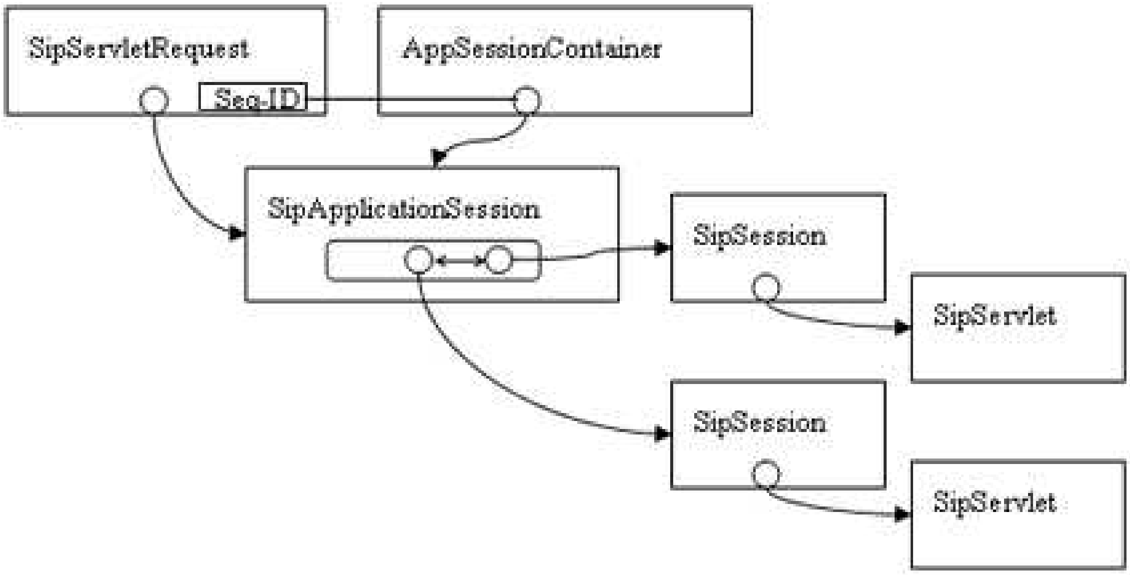


Figure 2.4: SipServlet, SipSession and SipApplicationSession

satisfying conditions are created.

2.3.2 Initialization (init())

SIP servlet handler does not give servlet class name, and it passes just servlet class path. Servlet Engine reads and parses all descriptor files from the path, and the descriptors are sorted with their priorities. Descriptors that has higher prioritiy are processed first.
ServletEngine gets class file name and checks servlet mapping conditions, and loads the class into JVM using ServletLoader. In this process, `SipSession` and `SipApplicationSession` are created. Finally, it calls `init()` function of SIP servlet.

2.3.3 Process Request (processRequest())

When this functions is called by sipd, `SipServletRequest` object is passed to ServletEngine. ServletEngine get the corresponding session iterator using the `SipServletRequest` from the session container. `service()` of SIP servlet in each `SipSession` is called with `SipServletRequest`.
All SipSession are sorted with their priorities already, so SIP servlets are processed in order of their priorities.

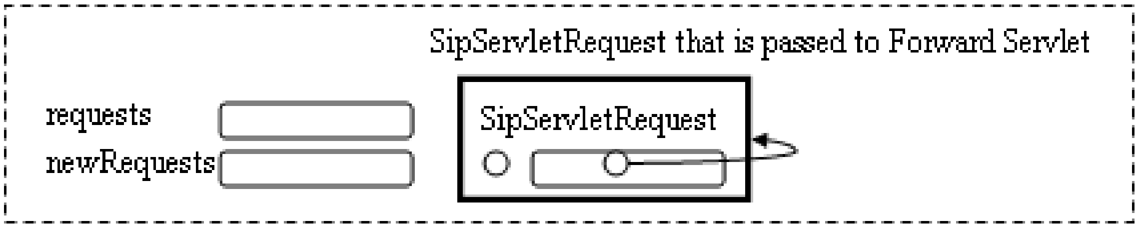


Figure 2.5: Original SipServltRequest

Originally, only one `SipServletRequest` objects exits. However, when it is forked, multiple `SipServletRequest` can be created.
Generally, if `SipServletRequest.send()` is called, the request is sent by native funtion. However, in this implementation, if there are next SIP servlets to be processed, the request is not sent. Instead, `SipServletRequest` object is cloned and added into `request` Vector, and it is passed to the next SIP servlets by ServletEngine. So, if `send()` is called twice, two `SipServletRequest` objects are created and added in the Vector, and later `service()` are called two times with the `SipServletRequests` when processing the next SIP servlets.
Originally, `processResonse()` is called by sipd with incoming response. However, `processResonse()` can be called by ServletEngine when a SIP servlet sends response and its previous SIP servlet exits. In this case, `SipServletResponse` object is created and registered in `SipServletRequest`. After processing request, ServletEngine checks if there is `SipServletResponse` object in `SipServletRequest` object. When ServletEngine finds the `SipServletResponse` object, it calls `processResponse()` with the `SipServletResponse`.

Let's look at the inside of ServletEngine using Forward and OCS SIP servlets example. In this example, Forward SIP servlet is invoked first and the original request is forked into two destinations. Then, OCS SIP servlet is invoked and one destination is screened by OCS SIP servlet.
Originally, only one SipServletRequest exist(Figure 2.5).
Before the SipServletRequest is passed to Forward SIP servlet, it is added into `requests` Vector(Figure 2.6).
When Forward SIP servlet is invoked, the original request is forked into two destinations, so two new `SipServletRequest` objects are created(Figure 2.7).
After processing Forward SIP servlet, the new `SipServletReqeusets` are moved into `newRequest` Vector(Figure 2.8).
Before OCS SIP servlet is invoked, the new `SipServletReqeusets` are moved into `requests` Vector(Figure 2.9). When a request is screened by a OCS SIP servlet, the `SipServletRequest` creates new `SipServletResponse` object, and ServletEngine calls `processResonse()` with the `SipServletResponse`(Figure 2.10).

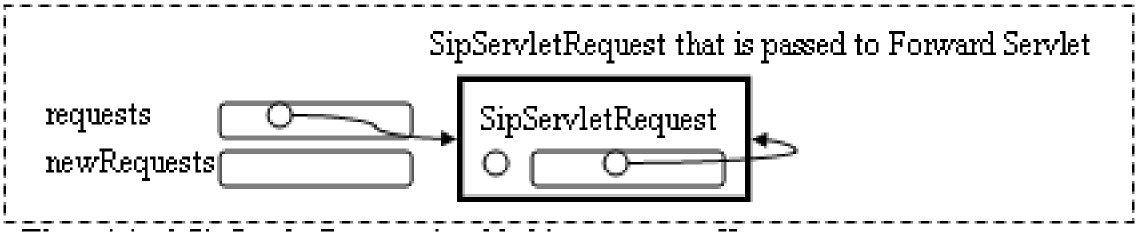


Figure 2.6: Before SipServletRequest is passed to Forwared SIP servlet

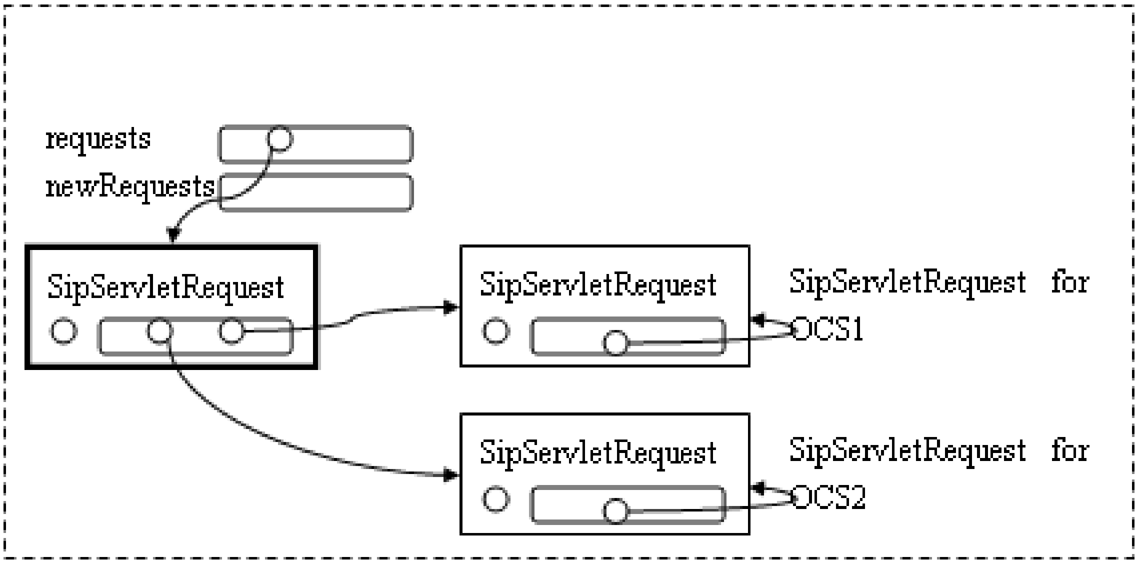


Figure 2.7: Rgiht after request is forked into two destination

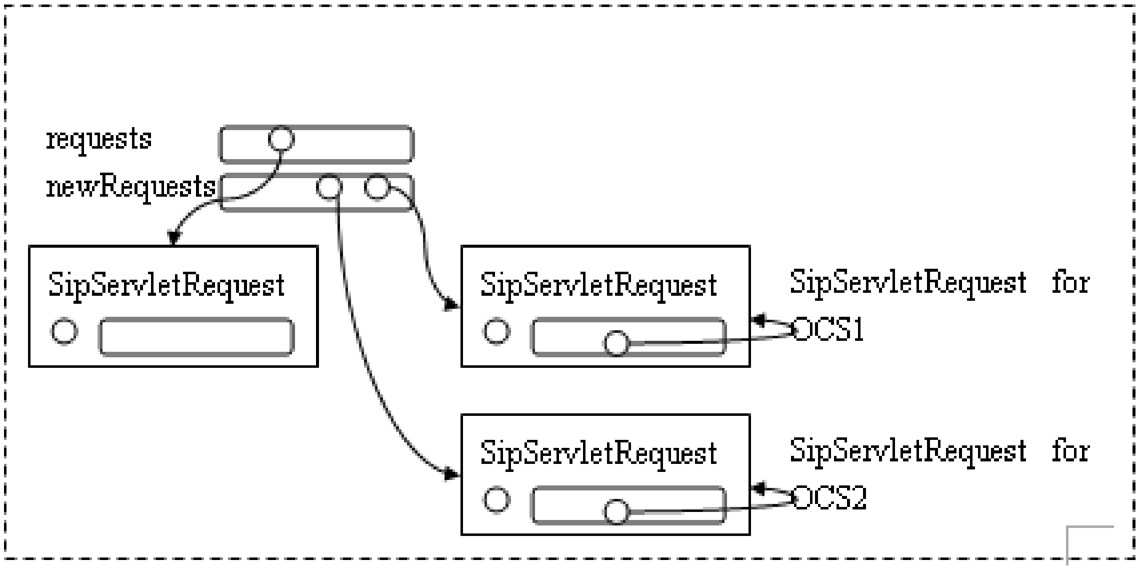


Figure 2.8: After Forward SIP servlet is invoked

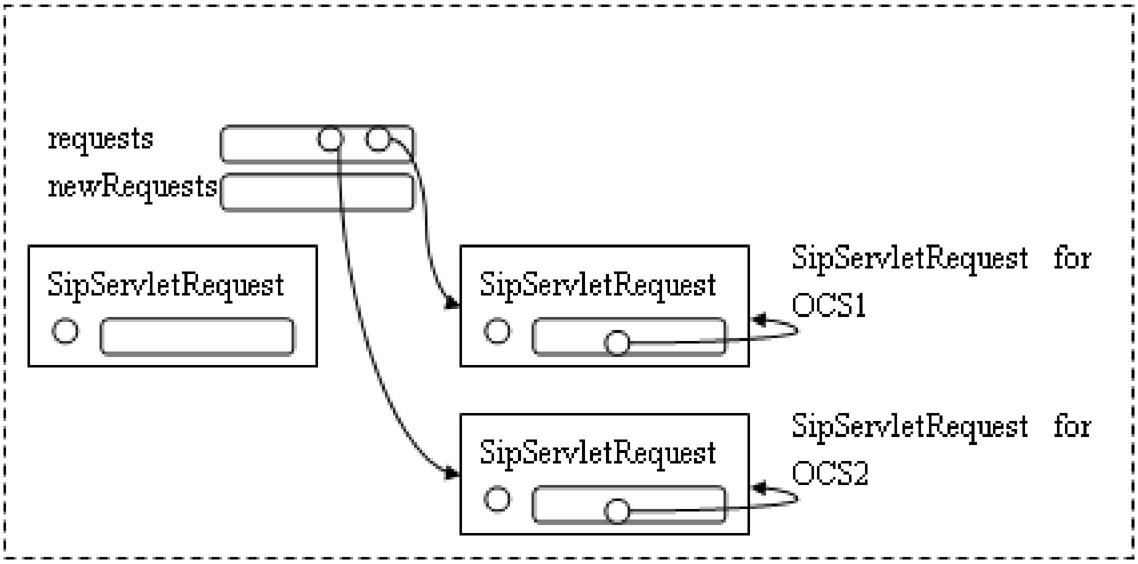


Figure 2.9: Before OCS SIP servlet is invoked

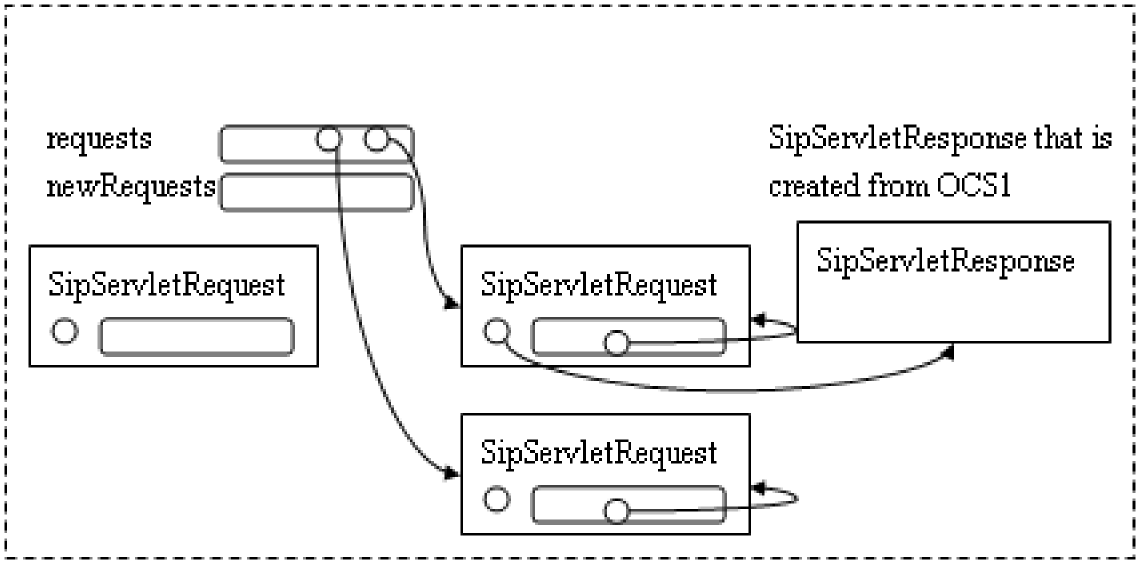


Figure 2.10: When one request is screened by a OCS SIP servlet

2.3.4 Process Response (processResponse())

The process of incoming response is simpler than the request process because the response is never forked by SIP servlets. Like `processRequest()`, `processResponse()` is called by sipd basically. However, as I mentioned before, it is called by ServletEngine sometimes.
Response follows the reverse application path. So, the last(the lowest priority) SIP servlet is invoked first. That's why `ListIterator` is used for session iterator.

2.3.5 Cleanup

`doCleanup()` function is called from sipd by user policy cleanup process. ServletEngine finds `SipApplicationSession` object and removes it from the session container, and the SIP servlets are removed by calling `destroy()` of the SIP servlet.
However, if the session timeout value in descriptor is specified, after the seesion time out, ServletEngine triggers this cleanup process.

2.4 Servlet Loader

Servlet Loader loads SIP servlet class into JVM from its servlet path. Servlet Loader class extends ClassLoader class. In JDK1.1, the ClassLoader class was abstract class, but in JDK1.2, it is not abstract class any more. Please refer to my short report, 'Implementation of SIP Servlet Loader' for more specific implementation.

2.4.1 Class reloading

SIP Servlet can be updated at anytime by user. Furthermore, various SIP servlets can be built and registered as user policies by many users, so the SIP servlet names can be conflicted. Therefore, ServletEngine should be able to reload new byte code for the servlet from repository without restarting Java Virtual Machine when SIP servlet classes are updated or SIP servlet of different user is uploaded.
ClassLoader maintains hashtable for caching the loaded class, and it always returns the cached class once a class is loaded into JVM. So, we should call findClass() method to load a SIP servlet class instead of loadClass() in the ServletEngine.
However, we have a problem still. Once a class is defined, the class that has the same name with the class cannot be defined again until the class is unloaded from JVM. Therefore, we should make the class unloaded from JVM to redefine the updated class. However, We cannot explicitly unload the class from JVM, but we should delete all the references to the class and make the class garbage-collected. Furthermore, ServletLoader instance should be released for this. Doing this, the class is unloaded from JVM. In my ServletEngine and Servlet Loader, there are lots references to the class when a servlet class is loaded.
So, we should set all the variables that refer to the class to `null`, including ServletLoader instance, and reload the class from repository.
And, we cannot always reload all classes from repository. We should check whether the class is updated or not, and only when the class is updated, we should reload the updated classes.
Hence, in `init()` funtion, ServletEngine checks the servlet class is updated or not, and if the servlet class is updated, current ServletLoader instance is released and new ServletLoader instance is constructed and updated class is redefined.

2.4.2 Multiple class loading

Generally, SIP servlet is composed of a class file. However, for specific reason, another class can be required(for example, to use `Timer` class, `TimerTask` class needs to be defined).
When another class file is defined, the file name is inserted in descriptor so that ServletEngine can read it. Although the file is in depository, JVM cannot find it when it is used in main class. So, the class file should be loaded into JVM so that JVM can find it, when main class is defined.

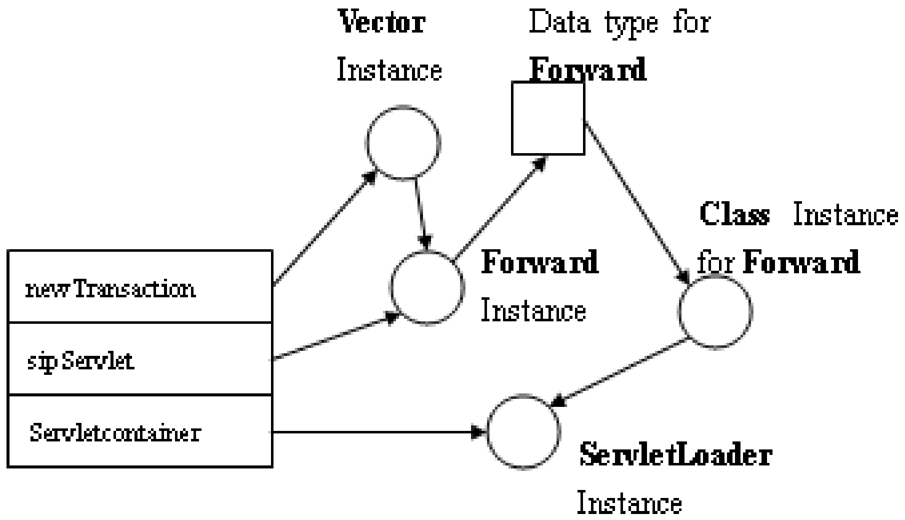


Figure 2.11: References of Forward SIP servlet class

2.5 Shared Library

```
void policy_send_new_response(request_t *r, int status, const char *reason);
void policy_send_modified_new_response(request_t *r, int status,
                                       const char *reason,
                                       headers_t *new_headers,
                                       headers_t *del_headers,
                                       body_t *body);
void policy_forward_response(request_t *r, message_t *m);
void policy_forward_user_response(request_t *r, message_t *m);
void policy_proxy_original_request_to(request_t *r, const uri_t *uri,
                                      message_t *m, void *policy_info,
                                      policy_type type);
void policy_proxy_subsequent_request_to(request_t *r, branch_t *b,
                                        message_t *m);
void policy_cancel_branch(request_t *r, branch_t *b);
void policy_set_timeout(request_t *r, double delay, policy_type type);
void policy_done(request_t *r, policy_type type);
```

For final real handling of request or response, policy_core supports the above functions, and all the final handling should be done by above the functions. So, `send()` function of `SipServletRequest` class and `SipServletResponse` class shoud be implemented with the above C native functions, and the functions are called using JNI callback mechanism. To call the abovce functions in native shared library, we should pass the function pointer to shared library.

Of course, we can include all the functions in the shared library, but the size of the shared library becomes very large. So, the sipservlet interface converts the function pointers to long type values and stores them to `user_policy_info` object, and the function pointers are converted to Java long type, and passed to shared library through JVM.
Native shared library is composed of two libraries, libSipServletResponseWrapper.so for `SipServletResponse` class, and libSipRequestWrapper.so for `SipServletRequest` class.
These libraries support `send()` function of `SipServletResponse` class and `SipServletReqeust` class. They construct `request_t` object and extract `user_policy_info` from the `request_t` object. In the `user_policy_info`, original `request_t` object pointer and lots callback function pointers are contained. These native functions call the corresponding callback function(in policy_core) with the original `request_t` object.
For mechanism of sipd, the original `request_t` object should be passed as an argument of callback functions. Otherwise, sipd cannot identify that response or request is sent, and it will ignore the response to the response or request.

2.6 Handling XML

SIP servlet descriptor has XML format. So, to parse the XML information, ServletEngine needs to handle XML file.
ParseXML.java is a module for reading and parsing XML file. To handle XML file, this module uses JAXP(Java API for XML Processing). Especially, SAX(Simple API for XML API) in JAXP is easy and simple to use. So, sax.jar and jaxp-api.jar files are required and included in `/cinema/sipservlet/api`. Please refer to Java API for XML Processing part in 'The Java Web Service Tutorial'.

2.7 Contacts Information

In old SIP servlet API, users could get contacts information using `ContactDatabase` class. However, the class is removed from new API. New API specification just says that users can get contacts information using 'non SIP method'. Therefore, ServletEngine provides a method for getting contacts inforamtion.
Sipd extracts contacts information from registration DB and stores it. So, sipservlet interface transfers the contact information to ServletEngine by packaging it with LinkedList object using JNI. Then, ServletEngine sets the contacts information to `SipServletConfig` object. Users can get contacts by just calling `getContacts()` in `SipServletConfig` class.

Chapter 3

CINEMA Web Interface for SIP Servlet

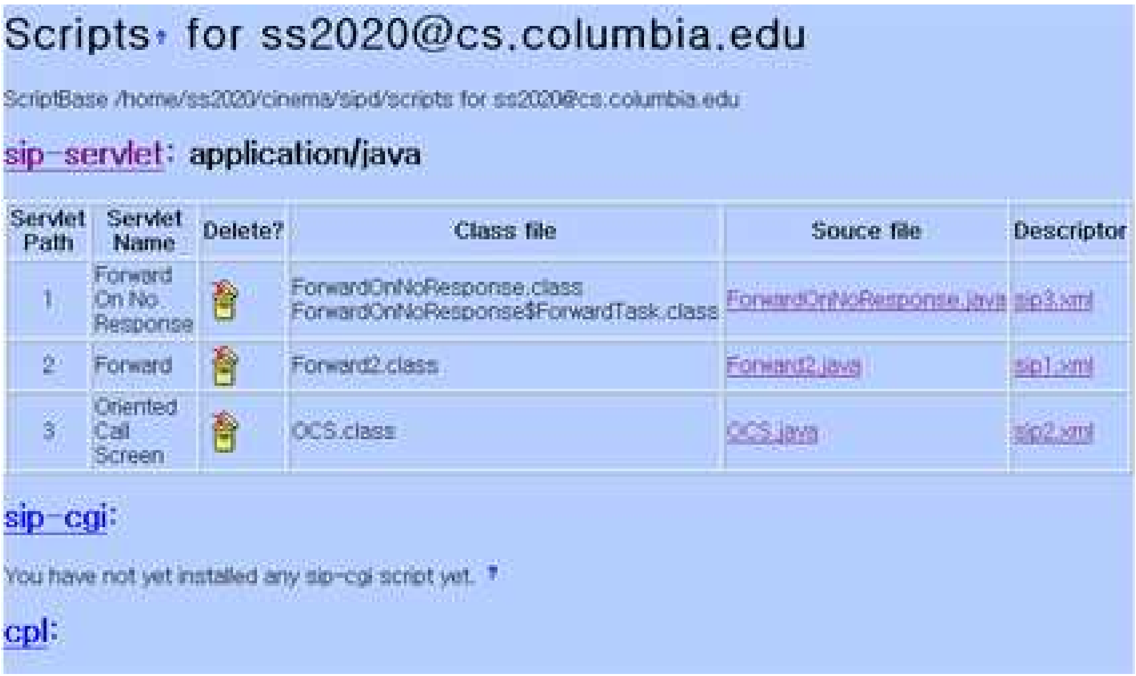
Users can upload SIP servlet using CINEMA web interface like SIP CGI and CPL. However, uploading SIP servlet is more complex than uploading SIP CGI and CPL.
All the scripts are written in Tcl language. For Tcl for CGI, please refer to 'Writing CGI scripts in Tcl)'.

3.1 SIP Servlet Display

If SIP servlet is registered, all SIP servlet information(servlet name, class file name, source file name, descriptor file name) is displayed in Scripts main page. When multiple SIP servlets are registered, SIP servlets are displayed in order of priorities, so the display order itself is the servlet path.
When users click a source file name, the source is displayed, and when users click a descriptor file name, the descriptor is displayed. By clicking `sip-servlet`, users can upload new SIP servlet application.

3.2 Registring New SIP Servlet

Registering new SIP servlet is composed of two steps. The first step is uploading source or class file, and the second step is writing descriptor.



Scripts for ss2020@cs.columbia.edu

ScriptBase /home/ss2020/cinema/sipd/scripts for ss2020@cs.columbia.edu

sip-servlet: application/java

Servlet Path	Servlet Name	Delete?	Class file	Souce file	Descriptor
1	Forward On No Response		ForwardOnNoResponse.class ForwardOnNoResponse$ForwardTask.class	ForwardOnNoResponse.java	sip3.xml
2	Forward		Forward2.class	Forward2.java	sip1.xml
3	Oriented Call Screen		OCS.class	OCS.java	sip2.xml

sip-cgi:

You have not yet installed any sip-cgi script yet.

cpl:

Figure 3.1: CINEMA Web Interface for SIP Servlet User Policy

3.2.1 Uploading SIP servlet

Users can upload not only SIP servlet class file but also SIP servlet source file. Of course, users can write or paste SIP servlet source file into TextBox. After user uploads SIP servlet source file or writes source in TextBox, web server compiles the source file, and displays error messages if there are compile errors. Otherwise, the compiled class file is registered as SIP servlet application.
Users can use inner class in SIP servlet, and upload multiple inner classes. However users can upload just one public class file. If another public class file is uploaded, the previous public class will be removed.
When SIP servlet source file is edited in TextBox, the file name should be decided automatically, and the file name should be the same as the public class name. So, CGI extracts public class name(and inner class name, if any) and sets it as the source file name. If CGI can't extract class name from the source, error message will be displayed.
When SIP servlet source is compiled without error, "Create SIP servle descriptor" button appears. If users click the button, users can jump to descriptor edit screen.

3.2.2 Editing descriptor

Users cannot edit or upload descriptor XML file directly. Users can just fill appropriate values, and then CGI generates descriptor XML file automatically. The reason is that syntactical and semantical error checking of XML file is not easy.
Whenever users click any `Change` button, the descriptor file is updated automatically. Descriptor file name is also decided automatically. Default file name would be "sip1.xml" and if the file name conflicts, the number is increased until it does not confilct.

Servlet Display Name The name to be displayed in Script main page.

Servlet Name The name for identifying SIP servlet application.

Class Name This is filled with the name extracted from source by CGI. I strongly recommend not to edit this. If there is a inner class, the inner class file name is also displayed.

Priority Real number from 0 to 1 is accepted, and 0 is the highest priority and 1 is the lowest priority. When the smaller value than 0 is inputted, it is changed into 0, and the greater value than 1 is inputted, it is changed into 1 by CGI. The higher priority SIP servlet is invoked first.

Session Timeout After `session timeout`, the SIP session is closed and the SIP application session is cleaned up. If this value is not specified, that is this value is 0, SIP session cleanup is up to sipd.

Servlet Parameters Both parameter name and value should be provided. This parameters are read and used in SIP servlet.

Servlet Mapping Conditions This specifies servlet mapping conditions. First, users select mathing method. "Match all of the following" works like Conjunctive normal form, and "Match any of the following" works like Disjunctive normal form. So, in CNF 'or' can be selected as connector, and in DNF 'and' can be selected as connector. If anything is not selected, the condition is connected with 'and' in CNF and 'or' in DNF.

1. Object : When a condition is inputted, CGI checks if the combination of objects is valid. If there are errors, the error messagss are displayed. For example, `request.method.uri.scheme` is not valid because `request.method` cannot have subobject. This check follows SIP Request Object Model. Please read chapter 11, Mapping Request to Servlets in 'SIP Servlet API Specification' for more information.

2. Not : 'not' or blank can be accepted.
3. Operator : There are four operators. An operator should be selected. 'equal' operator supports regular expression.
4. Value : This should not be blank.
5. Conn : 'or' can be selected in CNF, and 'and' can be selected in DNF. Blank is also accepted.

The order of mapping conditions can be changed by clicking 'Up' or 'Dn' buttons.
Users can check antomatically generated SIP sevlet descriptor XML file at any time by clicking "Display changed descriptor file" button.

3.3 Editing SIP servlet

When users click the source file name, the source is displayed. When users click "Edit source" button, users can edit the source in the text box or upload new source.
When users click a descriptor file name, the descriptor XML file is displayed. When users click "Edit source" button, users can edit the descriptor file.

3.4 Deleting SIP servlet

When users click the trash icon in scripts man page, the descriptor file, class files and the source files are deleted. To delete all SIP servlet applications, users have to click all trash icons of each SIP servlet application.

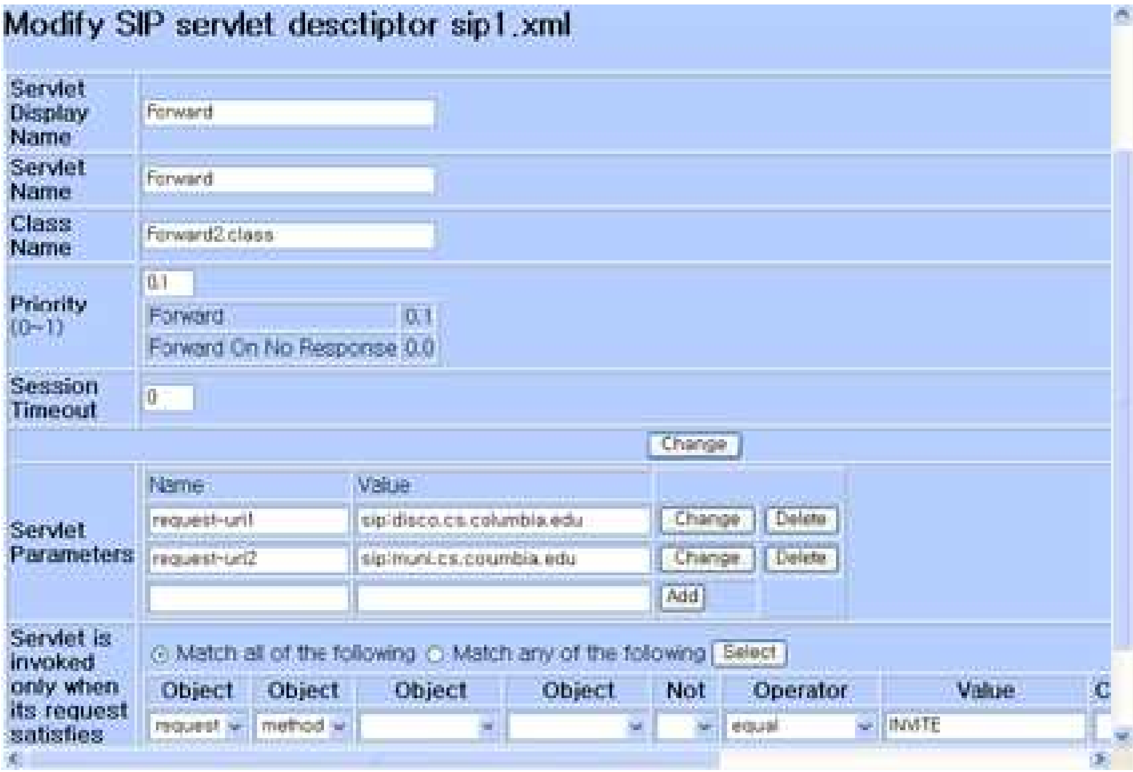


Figure 3.2: Editing SIP servlet descriptor

Chapter 4

Sample SIP Servlets

4.1 Forward SIP Servlet

Forward SIP servlet reads two request URIs from parameters, and sends original request to the two request URIs.

```
/*
 * Forward.java
 *
 * When request arrives, it forwards the request to the two requested uris
 * that are specified in servlet parameters.
 *
 * Sangho Shin (ss2020@cs.columbia.edu)
 *
 */

import java.util.*;
import javax.servlet.sip.*;
import javax.servlet.*;

public class Forward extends SipServlet {

    public void init(ServletConfig config) throws ServletException {
        super.init(config);
    }

    public void doInvite(SipServletRequest req) {
        ServletContext context = getServletContext();
        ServletConfig config = getServletConfig();

        /* read servlet parameters */
```

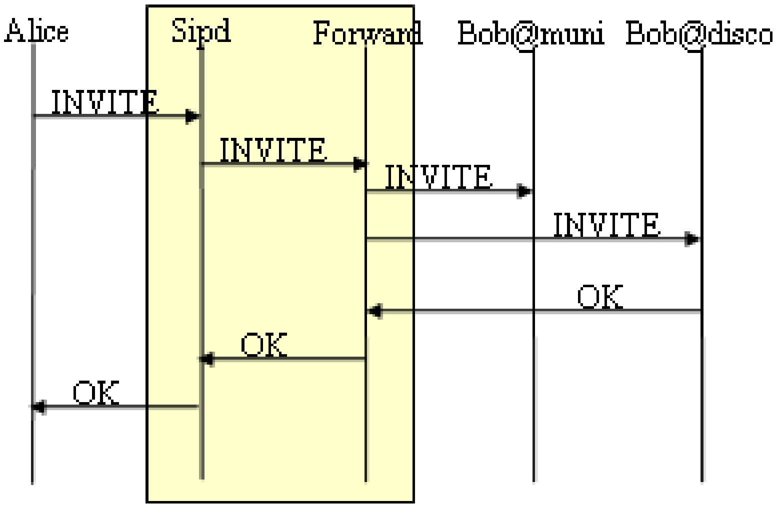


Figure 4.1: Forward SIP servlet

```
        String requestURI1 = (String)config.getInitParameter("request-uri1");
        String requestURI2 = (String)config.getInitParameter("request-uri2");
        SipFactory sipFactory
            = (SipFactory)context.getAttribute("javax.servlet.sip.SipFactory");
        try {
            URI uri1 = sipFactory.createURI(requestURI1);
            URI uri2 = sipFactory.createURI(requestURI2);

            /* send original request to two destinations */
            req.setRequestURI(uri1);
            req.send();
            req.setRequestURI(uri2);
            req.send();
        }
        catch (ServletException e) {
            System.out.println("Wrong uri");
        }
    }
}
```

Listing 1. Forward.java

```
<sip-app>
```

```
  <display-name>Forward</display-name>
  <context-param></context-param>
  <servlet>
    <servlet-name>Forward</servlet-name>
    <servlet-class>Forward2.class</servlet-class>
    <init-param>
      <param-name>request-uri1</param-name>
      <param-value>sip:disco.cs.columbia.edu</param-value>
      <param-name>request-uri2</param-name>
      <param-value>sip:muni.cs.columbia.edu</param-value>
    </init-param>
    <priority>0.1</priority>
  </servlet>
  <servlet-mapping>
    <servlet-name>Forward</servlet-name>
    <pattern>
      <and>
        <equal>
          <var>request.method</var>
          <value>INVITE</value>
        </equal>
      </and>
    </pattern>
  </servlet-mapping>
</sip-app>
```

Listing 2. Descriptor for Forward SIP servlet

4.2 Originating Call Screen SIP Servlet

OCS SIP servlet reads screening URIs from parameters, and if requestURI of original request is in the screening list, it sends FORBIDDEN(403) response. Otherwise, it sends the origianl request to requestURI.

```
/*
 * OCS.java
 *
 * First, reads screening URIs from parameters, and responds with 403 if
 * its requestURI is in the screening URIs. Otherwise, sends the request to
 * the requestURI.
 *
 * Sangho Shin (ss2020@cs.columbia.edu)
 *
 */
```

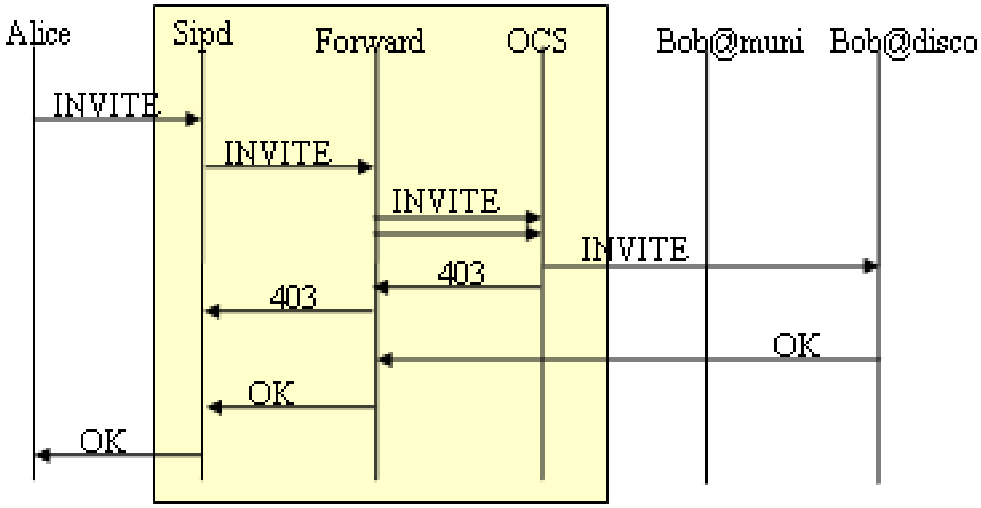


Figure 4.2: Originating Call Screen SIP servlet

```
import java.util.*;
import javax.servlet.sip.*;
import javax.servlet.*;

public class OCS extends SipServlet {
    protected int statusCode;
    protected String reasonPhrase;
    Vector screeningList;
    SipFactory sipFactory;

    public void init(ServletConfig config) throws ServletException {
        super.init(config);

        String parameterName = new String();
        String screeningURI = new String();
        ServletContext context = getServletContext();
        screeningList = new Vector();

        /* Get SipFactory */
        sipFactory
            = (SipFactory)context.getAttribute("javax.servlet.sip.SipFactory");

        /* Get screening URI from init parameters */
        String numStr = config.getInitParameter("number-of-uri");
```

```
        if (numStr == null)
            return;
        int numOfURI = (new Integer(numStr)).intValue();
        for (int i=1; i<=numOfURI; i++) {
            parameterName = "screening-uri" + i;
            screeningURI = (String)config.getInitParameter(parameterName);
            try {
                SipURI uri = (SipURI)sipFactory.createURI(screeningURI);
                screeningList.add(uri);
            }catch(ServletException e) {
            }
        }

        /* Get status code and reason phrase for response */
        String status = (String)config.getInitParameter("status-code");
        if (status == null) {
            statusCode = 403;
            reasonPhrase = "Forbidden";
        }
        else {
            reasonPhrase = (String)config.getInitParameter("reason-phrase");
            statusCode = (new Integer(status)).intValue();
        }
    }

    public void doInvite(SipServletRequest req) {
        boolean screen = false;
        SipURI requestURI = (SipURI)req.getRequestURI();
        /* If the caller URL is included in the screening list,
           reject the call */
        Iterator iter = screeningList.iterator();
        while(iter.hasNext()) {
            SipURI uri = (SipURI)iter.next();
            if (uri.getHost().equals(requestURI.getHost()))
                screen = true;
        }

        if (screen) {
            SipServletResponse res = req.createResponse(statusCode);
            res.send();
        }
        else {
            req.send();
        }
    }
}
```

Listing 3. OCS.java

```
<sip-app>
  <display-name>Originating Call Screen</display-name>
  <context-param></context-param>
  <servlet>
    <servlet-name>OCS</servlet-name>
    <servlet-class>OCS.class</servlet-class>
    <init-param>
      <param-name>number-of-uri</param-name>
      <param-value>2</param-value>
      <param-name>request-uri1</param-name>
      <param-value>sip:disco.cs.columbia.edu</param-value>
      <param-name>request-uri2</param-name>
      <param-value>sip:muni.cs.columbia.edu</param-value>
    </init-param>
    <priority>0.0</priority>
  </servlet>
  <servlet-mapping>
    <servlet-name>OCS</servlet-name>
    <pattern>
      <and>
        <equal>
          <var>request.method</var>
          <value>INVITE</value>
        </equal>
      </and>
    </pattern>
  </servlet-mapping>
</sip-app>
```

Listing 4. Descriptor for OCS SIP servlet

4.3 Forward On No Response SIP Servlet

First, FONR SIP servlet reads new requestURI and waiting-time from parameters. When INVITE request arrives, the servlet starts timer and sends the request to the original requestURI. If the servlet receives OK response from callee, it cancels the timer. If it cannot receive OK response from callee during wating-time, it sends the original request to new requestURI specified as parameter.

```
/*
 * ForwardOnNoResponse.java
 *
```

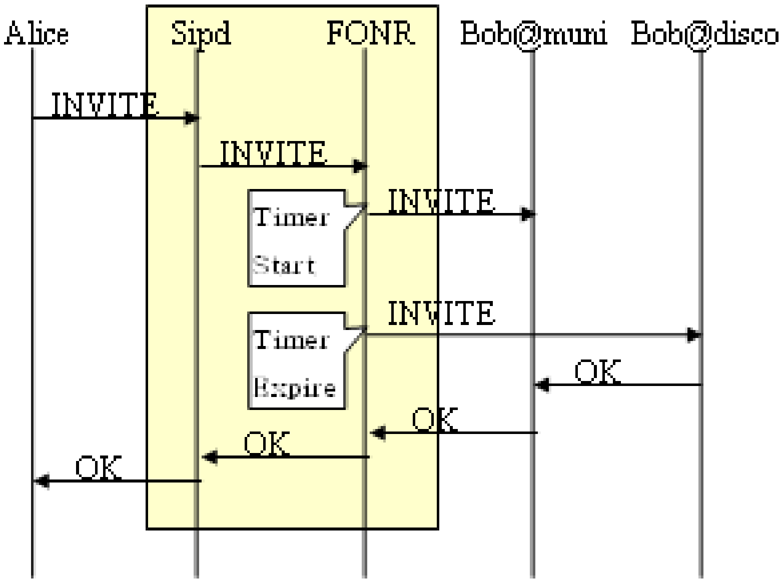


Figure 4.3: Forward On No Response SIP servlet

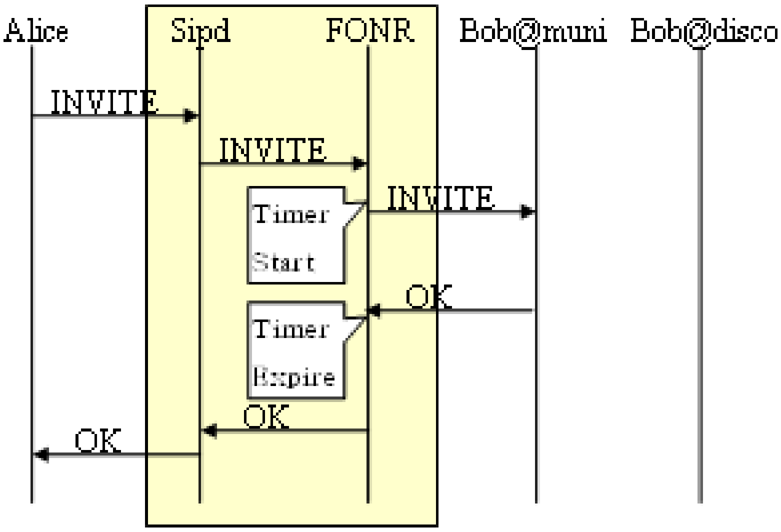


Figure 4.4: Forward On No Response SIP servlet

```
 * When request arrives, FONR SIP servlet starts timer, and send the request
 * to the original requestURI. If there is OK response from the callee,
 * the timer is cancelled. Otherwise, timer expires and the servlet sends
 * the original request to another destination that is specified as parameter.
 *
 * Sangho Shin (ss2020@cs.columbia.edu)
 *
 */
import java.util.*;
import javax.servlet.sip.*;
import javax.servlet.*;

public class ForwardOnNoResponse extends SipServlet {
    Timer timer;
    int waitingTime;
    SipServletRequest req;
    ServletConfig config;
    String requestURI;

    public void init(ServletConfig config) throws ServletException {
        super.init(config);
        config = getServletConfig();
        /* Set waiting time */
        String timeStr = (String)config.getInitParameter("waiting-time");
        requestURI = (String)config.getInitParameter("request-uri");
        if (timeStr == null)
            waitingTime = 20;
        else
            waitingTime = (new Integer(timeStr)).intValue();
    }

    public void doInvite(SipServletRequest req) {
        this.req = req;
        timer = new Timer();
        timer.schedule(new ForwardTask((SipServletRequestImpl)req, requestURI),
                       waitingTime*1000);
        req.send();
    }

    public void doResponse(SipServletResponse res) {
        if ( res.getStatus() == 200) {
            timer.cancel();
        }
    }

    public class ForwardTask extends TimerTask {
```

```
        SipServletRequestImpl request;
        String requestURI;

        public ForwardTask(SipServletRequestImpl req, String uri) {
            this.request = req;
            this.requestURI = uri;
        }

        public void run() {

            cancel(); /* cancle timer */

            ServletContext context = getServletContext();
            SipFactory sipFactory
            = (SipFactory)context.getAttribute("javax.servlet.sip.SipFactory");
            if (requestURI == null)
                return;
            try {
                URI uri = sipFactory.createURI(requestURI);
                request.setSendFinal(true);
                request.setRequestURI(uri);
                request.send();
            }catch(ServletException e) {
                System.out.println("Wrong uri");
            }
        }
    }
}
```

Listing 5. ForwardOnNoResponse.java

```
<sip-app>
  <display-name>Forward On No Response</display-name>
  <context-param></context-param>
  <servlet>
    <servlet-name>ForwardOnNoResponse</servlet-name>
    <servlet-class>ForwardOnNoResponse.class;ForwardOnNoResponse$ForwardTask.class</servlet-
    <init-param>
      <param-name>waiting-time</param-name>
      <param-value>10</param-value>
      <param-name>request-uri</param-name>
      <param-value>sip:dynasty.cs.columbia.edu</param-value>
    </init-param>
    <priority>0.0</priority>
  </servlet>
</sip-app>
```

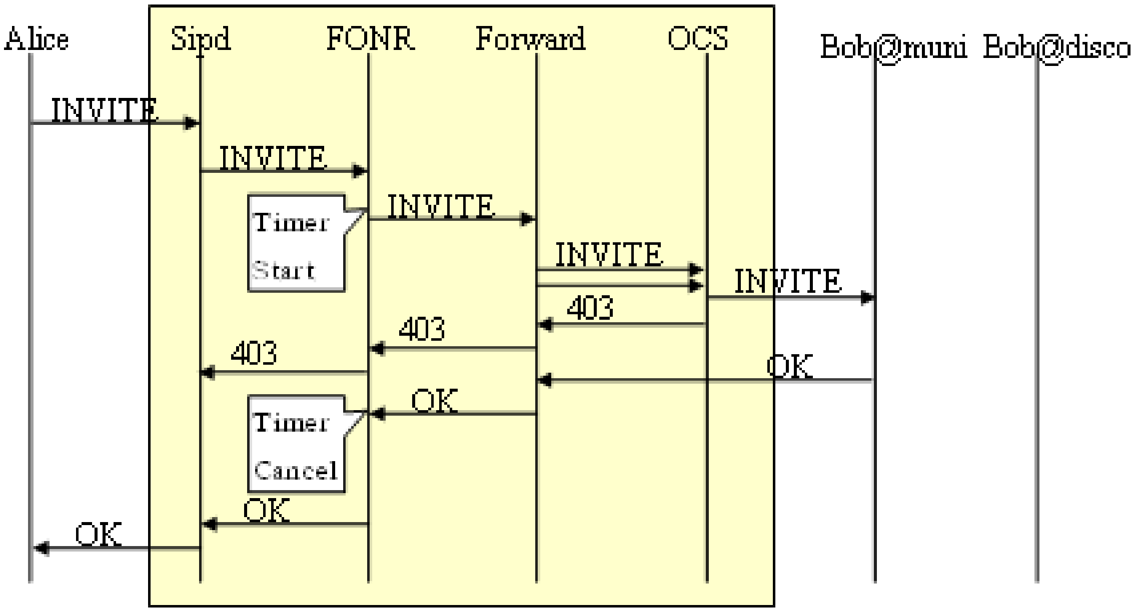


Figure 4.5: Three SIP servlet:When the first request is accepted

Listing 6. Descriptor for FONR SIP servlet

4.4 Combination of three SIP servlets

We can combine above all three SIP servlets. Servlet path is FONR→Forward→OCS. When a request arrives, FONR SIP servlet is invoked first, and it starts timer. Then, Forward SIP servlet is invoked, and it forks the request into two destinations. At last, OCS SIP servlet is invoked, and OCS screens one of the two request.

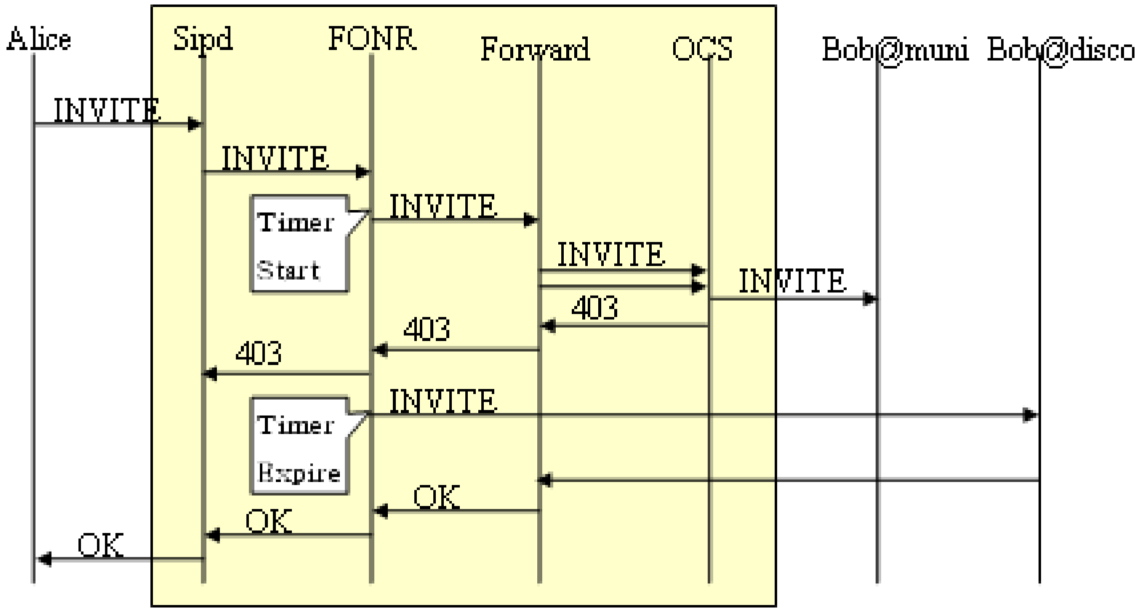


Figure 4.6: Three SIP servlet:When there is no OK response

Chapter 5

SIP Servlet API1.0 Implementation

SIP Servlet API 1.0 has a lot of features, classes and methods comparing to the previous version. However, some classes and functions are not applicable in this implementation because this SIP servlet works as a user policy of sipd. Therefore, some classes and methods are not implemented. Appendix A shows the implemented methods.
As the name of the implemented classes, I just attatched 'Impl' behind the original interface name following convention of previous version, except `SipServletConfig` class and `SipServletContext` class. These two files are implemented from `ServletConfig` and `ServletContext` interface in Java Servlet API.
This is the list of implemented classes

- ContactImpl.class
- SipAddressImpl.class
- SipApplicationSessionImpl.class
- SipFactoryImpl.class
- SipServletMessageImpl.class
- SipServletRequestImpl.class
- SipServletResponseImpl.class
- SipSessionImpl.class
- SipURIImpl.class
- TelURLImpl.class
- URIImpl.class

- SipServletConfig.class
- SipServletContext.class

These implemented classses are contained in sipservlet.jar, and original interfaces are included in sips.jar.

Chapter 6

Conclusion

This implementation is based on SIP Servlet API 1.0. Although this implementation don't use full features of the API, I believe that this SIP servlet user policy will give users the powerful control of SIP request and response.

This implementation works very well in Solaris environment. However, there are some problems in Linux because of compatibility problem in Linux JNI.
This implementation is based on SIP Servlet API 1.0 Draft 0.51, and Draft 0.7 was published already, and Draft 1.0 will be published sometime. So, this implementation needs to be modified as they are published.
I tested this implementaion with just one call at a time using `sipua`. I couldn't check how it works when multiple request arrives at a time. Scalability test will be required.
And, I just used `sip` scheme when testing. So, support of `TelURI` class or `tel` scheme is not tested yet.

Bibliography

1. A. Kreistensen, A. Byttner, The SIP Servlet API, Sept. 1999

2. Adnders Kristensen, SIP Servlet API Version 1.0 Public Draft Version 0.51, Mar. 2002.

3. Adnders Kristensen, SIP Servlet API Version 1.0 Public Draft Version 0.7, Nov. 2002.

4. Don Libes, Writing CGI scripts in Tcl.

5. Eric Armstrong, Simple API for XML, http://java.sun.com/webservices/docs/1.0/tutorial/doc/JAXPSAX.html

6. java.sun.com, Java Tutorial. Java Native Interface.

7. Jonathan Lennox, sipd: SIP proxy, redirect and registrar server.

8. Jonathan Lennox, The CINEMA LIBSIP policy API.

9. Rosenberg, Schulzrinne, Camarillo, Johnston, Peterson, Sparks, Handley, Schooler SIP: Session Initiation Protocol INTERNET DRAFT Oct. 2001

10. Sheng Liang, The Java Native Interface Programmer's Guide and Specification

11. Wenyu Jian, Jonathan Lennox, Sakanran Narayanan, Henning Shulzrinne and Kundan Singh, Towards Junking the PBX: Deploying IP Telephony

Appendix A

Implemented Classes and methods

- ContactImpl.class

```
public boolean isWildcard()
public float getQ()
public void setQ(float q)
public int getExpires()
public void setExpires(int seconds)
public String toString()
public int compareTo(Object o)
```

- SipAddressImpl.class

```
public String getDisplayName()
public void setDisplayName(String name)
public URI getURI()
public void setURI(URI uri)
public String getParameter(String name)
public void setParameter(String name, String value)
public void removeParameter(String name)
public Iterator getParameterNames()
public String toString()
public Object clone()
```

- SipApplicationSessionImpl.class

```
public void addSipSession(SipSessionImpl session)
public int getNumOfServlet()
public void setNumOfServlet(int num)
public long getCreationTime()
```

```
public long getLastAccessedTime()
public int setExpires(int deltaMinutes)
public void invalidate()
public Iterator getSessions()
public Iterator getSessions(String protocol)
public String encodeURL(String url)
public Object getAttribute(String name)
public Iterator getAttributeNames()
public void setAttribute(String name, Object attribute)
public void removeAttribute(String name)
```

- SipFactoryImpl.class

```
public URI createURI(String uri)
public SipAddress createSipAddress(String sipAddress)
public SipAddress createSipAddress(URI uri)
public SipAddress createSipAddress(URI uri, String displayName)
public Contact createContact(String contact)
public Contact createContact(URI uri)
public Contact createContact(URI uri, String displayName)
public Contact createContact(SipAddress addr)
public SipApplicationSession createApplicationSession()
```

- SipServletConfig.class

```
public String getInitParameter(String name)
public Enumeration getInitParameterNames()
public ServletContext getServletContext()
public String getServletName()
public LinkedList getContacts()
public void setInitParameter(String key, Object value)
```

- SipServletContext.class

```
public Object getAttribute(String name)
public Enumeration getAttributeNames()
public ServletContext getContext(String uriPath)
public String getInitParameter(String name)
public Enumeration getInitParameterNames()
public int getMajorVersion()
public int getMinorVersion()
public String getServerInfo()
public void removeAttribute(String name)
public void setAttribute(String name, Object object)
```

- SipServletMessageImpl.class

```
public String getCallId()
public SipAddress getFrom()
public SipAddress getTo()
public List getContacts()
public void setContacts(List contacts)
public String getMethod()
public String getProtocol()
public String getHeader(String name)
public Iterator getHeaders(String name)
public Iterator getHeaderNames()
public void setHeader(String name, String value)
public void addHeader(String name, String value)
public void removeHeader(String name)
public int getExpires()
public void setExpires(int seconds)
public int getMaxForwards()
public void setMaxForwards(int n)
public String getCharacterEncoding()
public int getContentLength()
public String getContentType()
public Object getContent()
public byte[] getRawContent()
public void setContent(Object obj, String type)
public ServletInputStream getInputStream()
public ServletOutputStream getOutputStream()
public void setContentLength(int len)
public void setContentType(String type)
public Object getAttribute(String name)
public Enumeration getAttributeNames()
public void setAttribute(String name, Object o)
public SipSession getSession()
public SipSession getSession(boolean create)
public SipApplicationSession getApplicationSession()
public SipApplicationSession getApplicationSession(boolean create)
public boolean isRequestedSessionIdValid()
public Locale getAcceptLanguage()
public Enumeration getAcceptLanguages()
public void setAcceptLanguage(Locale locale)
public void addAcceptLanguage(Locale locale)
public void setContentLanguage(Locale locale)
public Locale getContentLanguage()
public void send()
public boolean isSecure()
public boolean isCommitted()
public String getRemoteUser()
```

- SipServletRequestImpl.class

```
public URI getRequestURI()
public void setRequestURI(URI uri)
public void pushRoute(SipAddress proxyAddr)
public void pushRoute(SipURI proxyAddr)
public void send()
public boolean isInitial()
public Proxy getProxy(boolean create)
public SipServletResponse createResponse(int statusCode)
public SipServletResponse createResponse(int statusCode, String reasonPhrase)
public SipServletRequest createCancel()
public long getRequestPointer()
public Locale getLocale()
```

- SipServletResponseImpl.class

```
public SipServletRequest getRequest()
public int getStatus()
public void setStatus(int statusCode)
public void setStatus(int statusCode, String reasonPhrase)
public String getReasonPhrase()
public void send()
public SipServletRequest createAck()
```

- SipSessionImpl.class

```
public long getCreationTime()
public String getId()
public long getLastAccessedTime()
public void invalidate()
public SipApplicationSession getApplicationSession()
public String getCallId()
public SipAddress getLocalParty()
public SipAddress getRemoteParty()
public SipServletRequest createRequest(String method)
public void setHandler(String name)
public Object getAttribute(String name)
public Enumeration getAttributeNames()
public void setAttribute(String name, Object attribute)
public void removeAttribute(String name)
```

- SipURIImpl.class

```
public SipURIImpl(String scheme, String user, String pass, String host,
      int port, HashMap params, HashMap hdrs)
```

```
public SipURIImpl(String uri)
public HashMap getAllParams()
public HashMap getAllHeaders()
public String getUser()
public void setUser(String user)
public String getUserPassword()
public void setUserPassword(String password)
public String getHost()
public void setHost(String host)
public int getPort()
public void setPort(int port)
public String getParameter(String name)
public void setParameter(String name, String value)
public void removeParameter(String name).
public Iterator getParameterNames()
public String getTransportParam()
public void setTransportParam(String transport)
public String getMAddrParam()
public void setMAddrParam(String maddr)
public String getMethodParam()
public void setMethodParam(String method)
public int getTTLParam()
public void setTTLParam(int ttl)
public String getUserParam()
public void setUserParam(String user)
public boolean getLrParam()
public void setLrParam(boolean flag)
public String getHeader(String name)
public void setHeader(String name, String value)
public Iterator getHeaderNames()
public String toString()
```

- TelURLImpl.class

```
public String getTelNo()
public boolean isGlobal()
public String getParameter(String key)
public Iterator getParameterNames()
public String toString()
```

- URIImpl.class

```
public String getScheme()
public String toString()
public boolean isSipURI()
```

Appendix B

Diretory Structure

`/cinema/sipservlet` : ServletEngine & ClassLoader
`/sipservlet/api` : SIP Servlet API implementation & interfaces
`/sipservlet/wrapper` : Shared Libraries
`/sipservlet/sample-servlets`: Sample SIP servlets