Implementation of SIP Servlet User Policy

Sangho Shin

Columbia University

Department of Computer Science

ss2020@cs.columbia.edu

Contents

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 module's 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.

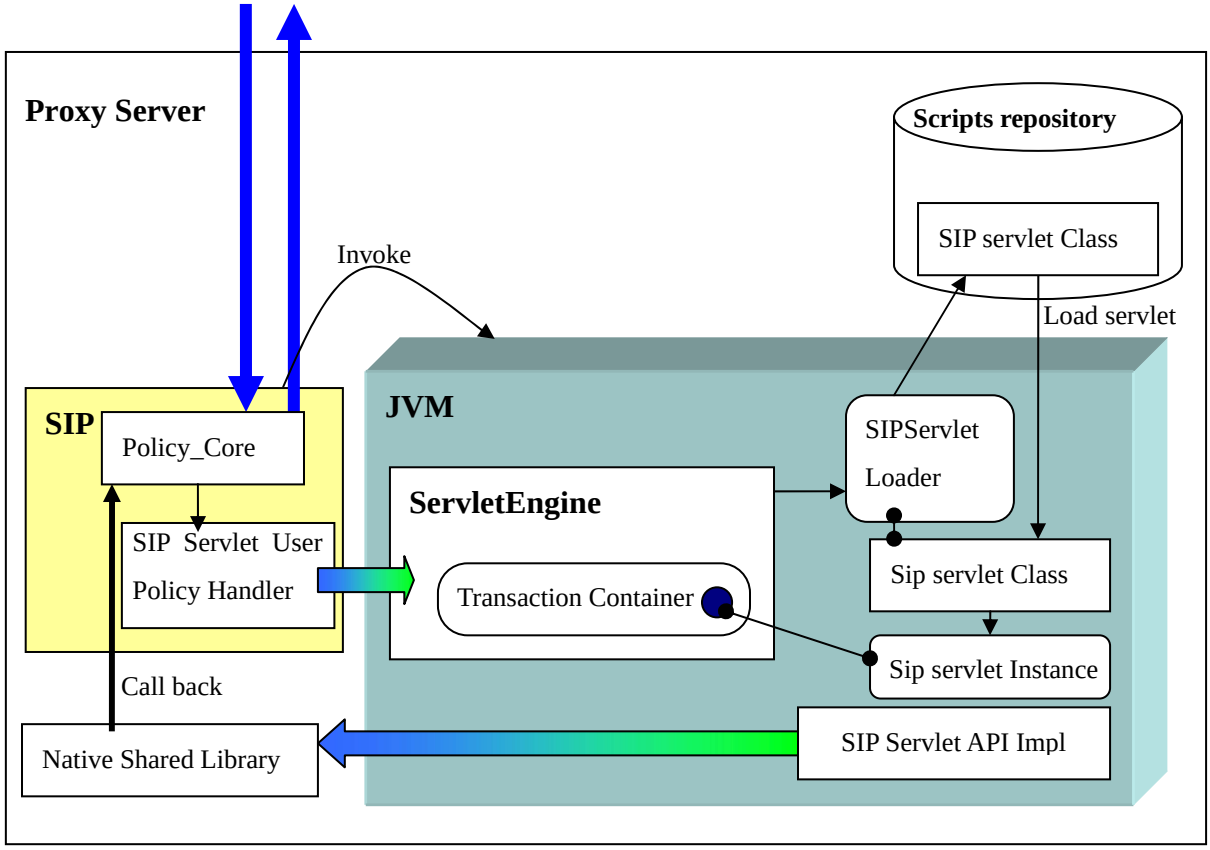
1.2 SIP servlet

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 is 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). When I started to do this project, only first version was published. So, this implementation is based on the first version of SIP servlet API. Please refer to my document, SIP servlet API version 1.0 Review [SIPServletAPI1.pdf] for their difference.

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 engine for SIP servlets.



[Fig 1] SIP servlet user policy architecture

2.1 User policy architecture

Policy core invokes policy functions and provides the basic state machinery for policy invocation.

When SIP message arrives to 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

sipsevlet user policy handler implements the following 6 functions.

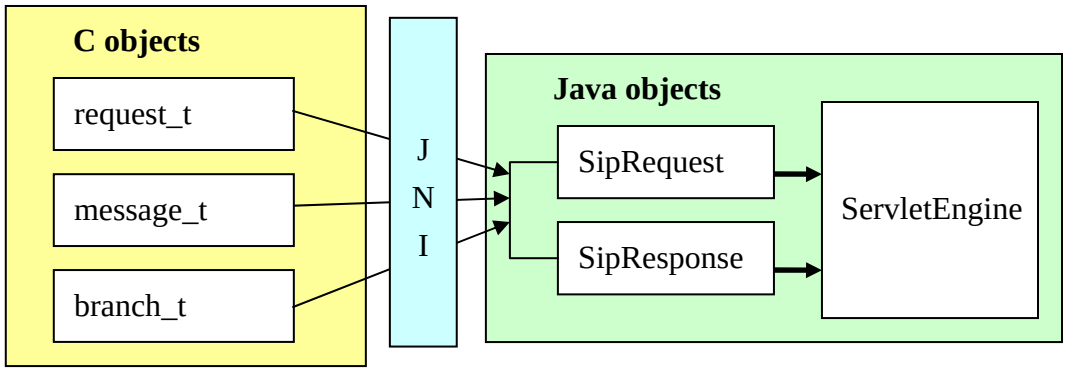
```
user_policy_ret sipservlet_init(request_t *r, const policy_info_t *reg, int status);
user_policy_ret sipservlet_handle_initial_request(request_t *r);
user_policy_ret sipservlet_handle_subsequent_request(request_t *r, message_t *m);
user_policy_ret sipservlet_handle_response(request_t *r, message_t *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 script object pointer, callback function pointers for native shared library, original request object pointer, and calleeID. This user_policy_info is stored in request_t object for later use. I will describe all about those later.

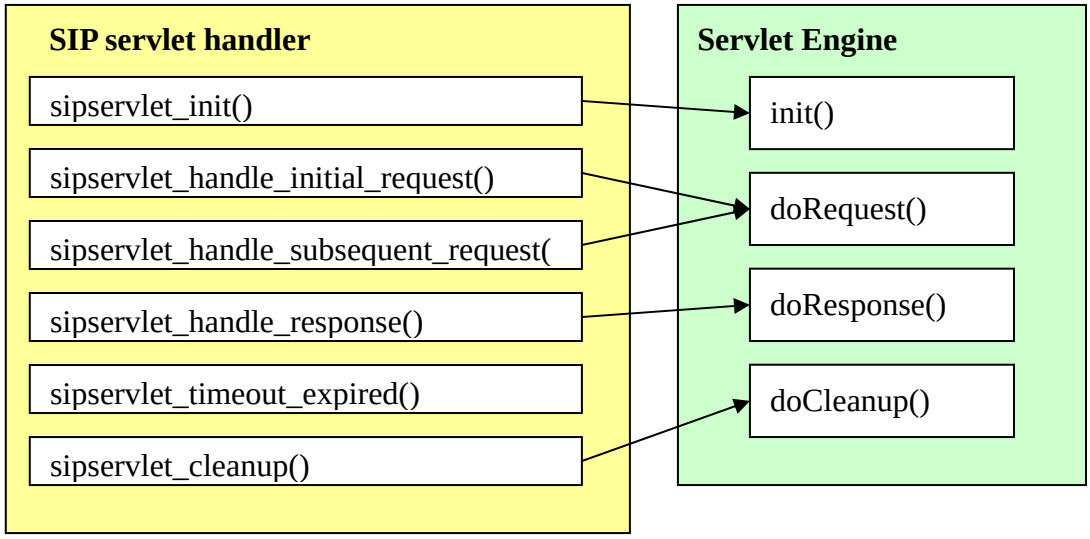
```
struct sipservlet_info {
   script_t *script;
   void (*send_request_to)(request_t *r, const uri_t *uri, message_t *m,
                           void *policy_info, policy_type type);
   void (*send_new_response)(request_t *r, int status, const char *reason);
   void (*contact_insert_contact)(cinema_table *table, const contact_table_entry *entry,
                                  db_err *err, char **reason);
   request_t *original_request;
   char *callee_id;
```

};

After constructing user_policy_info, it constructs SipResquest 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.



[Fig 2] SIP message objects conversion



[Fig 3] Function mapping between SIP servlet handler and Servlet Engine

xxx_request(), xxx_response(), and xxx_cleanup() functions constructs constructs SipRequest and SipResponse objects and call mapping function of ServletEngine.

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 transaction container (Hashtable). Whenever new SIP transaction is invoked from SIP servlet handler of sipd, Servlet Engine constructs a SipTransaction object and registers the SIP servlet object as its listener, and store the SIP transaction to the transaction container. Originally, multiple servlets can be registered as listener of a SipTransaction. However, this feature is open issues in this version of SIP servlet API, and only one servlet can be registered in this implementation.

2.3.1 Initialization (init())

SIP servlet handler does not give servlet class name, and it passes just servlet class path. Servlet Engine checks the servlet class path, and gets the servlet class name. In the servlet path, only one public class should exist, but multiple inner classes can exist. If there is no public class in the pass, the init function will return false.

After getting the servlet class name, ServletEngine loads the servlet class into JVM using ServletLoader and constructs its instance. Then, it constructs a SipTransaction object and register the servlet as its listener, and put the transaction object into transaction container (Hashtable class). Finally, it calls init() function of SIP servlet.

2.3.2 doRequest and doResponse

When the two functions are called, SipRequest object or SipResponse objects are passed to ServletEngine. ServletEngine searches the SipTransaction objects with the passed object (in real, CallID is the search key) from transaction container, and gets the listener SIP servlet. If Servlet Engine cannot find SipTransaction, return false.

If Servlet Engine succeeded in getting listener SIP servlet, it calls gotRequest() or gotResponse() function.

2.3.3 Cleanup

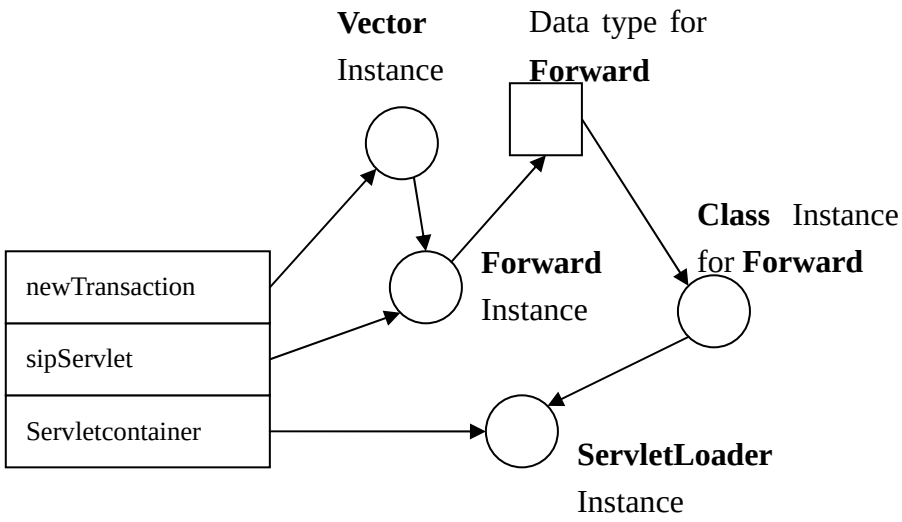
doCleanup function is called from sipd by user policy cleanup process. Serlvet Engine searches the SipTransaction object from the container and removes the transaction from the container. And, the listener SIP servlet of the transaction is removed by calling destroy() of the SIP servlet.

2.4 Servlet Loader

Servlet Loader loads SIP servlet class into JVM from the 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 shor report, Implementation of SIP Servlet Loader for the specific implementation.

SIP Servlet can be updated at anytime by user. Furthermore, various SIP servlets can be built by many users, and can be registered in proxy server. So, the SIP servlet name can be conflicted. Therefore, Servlet Engine should be able to load new byte code for the servlet from disk without restarting Java Virtual Machine when the SIP servlet class is updated, or the SIP Servlet of a different user should be loaded.
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 Servlet Engine.
However, we have a problem still. Once one 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.

We cannot explicitly unload the class from JVM, rather we should delete all the references to the class and make the class garbage-collected.
In my Servlet Engine and Servlet Loader has lots reference to the class when a servlet class is loaded.



[Fig 4] References of Forward SIP servlet class

So, we should set all the variables that refer to the class to null, including ServletLoader instance, and reload the class from disk.
And, we cannot always load all classes from disk, so we should check whether the class is updated first. Only when the class is updated, we should make new instance of ServletLoader.
Hence, when init function of ServletEngine, ServletEngine checks the servlet class is updated or not, and if the servlet class is updated, we should release the current ServletLoader instance and construct new ServletLoader instance for this reason.

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 SipReqeust class and SipResponse 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 become very large. So, we convert the function pointer to long value and store them to user_policy_info object, and the

function pointers are converted to Java long type, and passed to shared library through JVM.

Native share library is composed of three libraries. libContactDBWrapper.so for ContactDatabase class, libSipResponseWrapper.so for SipResponse class, and libSipRequestWrapper.so for SipRequest class.

2.5.1 libContactDBWrapper.so

The library supports getContacts() and addContact() function of ContactDatabase class. getContacts() function get all the registered contacts info from DB, and addContact() function add a contact for the user to DB.

SIP servlet API can use JDBC to get the contacts from DB and add a contact to DB. But, to decrease the dependency on DB of the API implementation, the API calls the native function. By using native function, the API implementation cannot be modified even if the table name of fields name of the contact info in DB.

2.5.2 libSipResponseWrapper.so, libSipRequestWrapper.so

These libraries supports send() function of SipResponse class and SipReqeust 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 the argument of the callback function. Otherwise, sipd cannot identify that any response or request is sent, and will ignore the response to the response or request.

3 CINEMA web interface for sip servlet

[Fig 5] CINEMA Web interface for SIP servlet user policy

Users can upload SIP servlet using CINEMA web interface as likely as SIP CGI and CPL. However, SIP servlet web interface has different behavior from the SIP CGI and CPL.
All the scripts are written in Tcl language. For the Tcl for CGI, please refer to “Writing CGI scripts in Tcl”.

3.1 SIP servlet display [Scripts.cgi]

If SIP servlet is registered, the SIP servlet class file name is displayed. If SIP servlet source file is registered, the source file is displayed and the source can be edited

3.2 SIP servlet upload and edit [ScriptEdit.cgi]

Users can upload not only SIP servlet class file but also SIP servlet source file. Of course, users can write or paste the SIP servlet source file into TextBox. When user uploads SIP servlet source file or writes source in TextBox, web server compiles the source file, and displays error message in case of compile error. Otherwise, the

compiled class file is registered as SIP servlet user policy.
Users can use inner class in SIP servlet, and upload the 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. The reason is that only one SIP servlet can be registered as SIP transaction listener in this implementation, and if multiple public class file is uploaded, ServletEngine cannot know which one is real class file.
The SIP servlet class name is not stored anywhere. So, ServletEngine checks the servlet path, and knows SIP servlet class name.
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, Upload cgi checks public class name and decides the source file name. If the upload cgi can’t extract class name from the source, error message will be displayed.

3.3 SIP servlet delete. [ScriptDelete.cgi]

When user deletes the public class file, the source file is deleted if any. However, all the inner class files should be deleted separately.

4 Simple scalability test

I measured the process time of the SIP servlet user policy to test its scalability. For the more exact scalability test, I should have measured the time when a number of SIP transaction is processed by the SIP servlet user policy simultaneously, but I couldn't.
In stead of it, I set the SIP servlet class that does nothing but just returning false, and its pure object conversion time and SIP servlet function call time using JNI.

1 st call			
Init	sipservlet_init()	ServletEngine.init()	sipd
	361msec	304msec	
Request	sipservlet.handle_request()	ServletEngine.doRequest()	sipd
	4msec	11 msec	
Response	sipservlet.handle_response()	ServletEngine.doResponse()	sipd
	11msec	9msec	

2nd call			
Init	sipservlet_init()	ServletEngine.init()	sipd
	361msec	304msec	
Request	sipservlet.handle_request()	ServletEngine.doRequest()	sipd
	4msec	11 msec	
Response	sipservlet.handle_response()	ServletEngine.doResponse()	sipd
	11msec	9msec	

3 rd call			
Init	sipservlet_init()	ServletEngine.init()	sipd
	361msec	304msec	
Request	sipservlet.handle_request()	ServletEngine.doRequest()	sipd
	4msec	11 msec	
Response	sipservlet.handle_response()	ServletEngine.doResponse()	sipd
	11msec	9msec	

[Table 1] Simple scalability test results

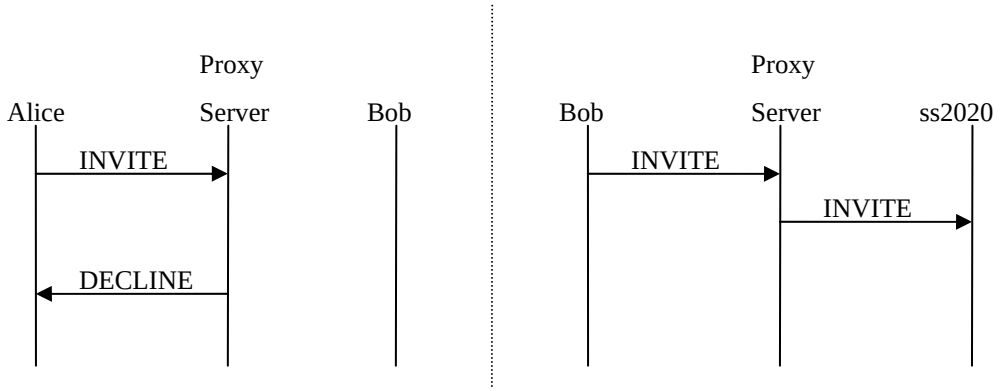
When SIP message arrived, policy_core calls sipservlet_init(). This function convert C sipd objects(request_t, message_t etc) to Java objects(SipRequest, SipResponse), and

calls init() function of ServletEngine in JVM using JNI.
In first call, the conversion process of sipservlet_init() takes 361 msec. The reason is that all the classes required for conversion should be loaded into JVM. In the second call, it takes just 4 msec because all the classes are loaded into JVM already. So, we can ignore the first 361 msec delay because all the classes should be loaded into JVM after the first call.
This init() of ServletEngine initialize the ServletLoader, and loads SIP servlet from disk using the ServletLoader. In first call, the init() function takes 304msec.
This first call delay occurs only once until sipd is killed, so we can ignore the first call delay. However, we can decrease the first call delay by loading all the required classes into JVM when sipd is initialized.
Right after initialization of user policy handler, policy core calls sipservlet_handle_initial_request() function. This function converts all the C objects to Java object as likely as the init process, and call doRequest() of ServletEngine using JNI. However, it takes just 4 msec because all the objects are loaded into JVM.
doRequest() of Servlet Engine does nothing and just returns false. So, we can say that 4msec is purely Java function call dealy.
Seeing that 2nd and 3rd call logs, we can find that all the conversion process and function call take below 10 msec except ServletEngine.init(). The reason is that init() function of ServletEngine checks if the servlet is updated or not by accessing DISK even if the servlet class is loaded into JVM already.
I tested another case. After executing DoNothing SIP servlet user policy, I changed the SIP servlet to Forward.class. This SIP servlet forwards the request to another branch. So, ServletEngine should load the new SIP servlet class into JVM from DISK, and we can measure the pure SIP servlet class loading time, because all the other classes are loaded into JVM already. It took 14 seconds. And, the doRequest() of Forward class process time was below 10 msec. However, it depends on how to implement the SIP servlet.

5 Sample SIP servlets

5.1 Call screening

Screening list : Alice@cs.columbia.edu



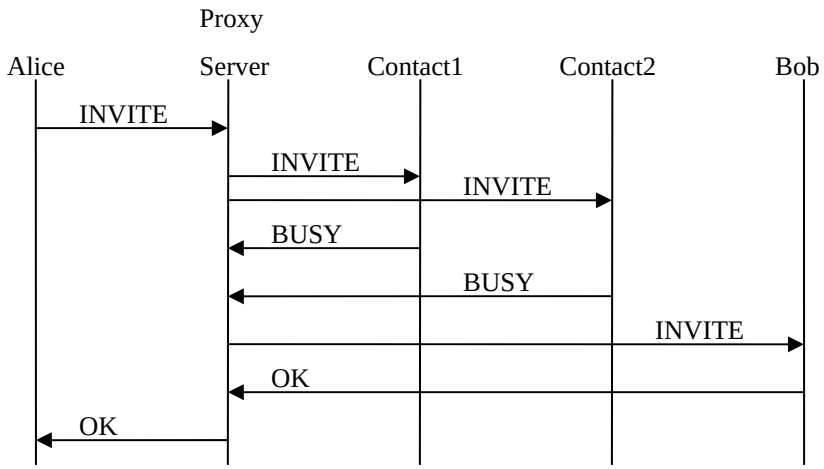
.

This servlet reject a call from a user who is included in call screening list. doRequest() function checks if the caller is in the call screening list, and sends BUSY HERE response to the caller if the caller is in the list. Otherwise, it will defer the handling to sipd by returning false.

```
import org.ietf.sip.*;
import java.util.Vector;

public class CallScreen extends SipServletAdapter {
   protected int statusCode;
   protected String reasonPhrase;
   Vector screeningList;
   SipFactoryImpl factory = new SipFactoryImpl();

   public void init(ServletConfig config) {
      super.init(config);
      factory = new SipFactoryImpl();
      screeningList = new Vector();
      try {
         /* You can also read screening list from DB or files */
         SipURL url =
```

```
            factory.createSipURL("sip:ss2020@disco.cs.columbia.edu");
         screeningList.add(url.toString());
      }catch(ParseException e) {
      }
      statusCode = SC_DECLINE;
      reasonPhrase = "Decline";
   }

   public boolean doInvite(SipRequest req) {
      String uri = req.getFrom().getSipURL().toString();
      /* If the caller URL is included in the screening list, reject
         the call */
      if (screeningList.contains(req.getTo().getURI())) {
         SipResponse res = req.createResponse();
         res.setStatus(statusCode, reasonPhrase);
         res.send();
         return true;
      }
      else
         return false;
   }
}
```

5.2 ForwardOnBusy Servlet



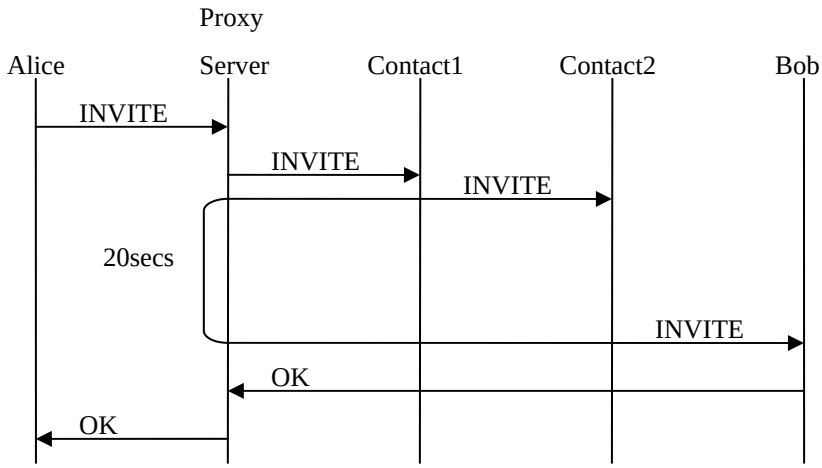
This servlet forwards the request when all the braches are busy or reject the call. doRequest() function gets the registered contacts from DB, and fork the request to the all contacts. doResponse() function checks if all the branches sends BUSY or rejects the call, and forward the original request to new branch. If one of the contacts accepts the call, defer the handling to sipd by returning false.

```
import java.util.*;
import org.ietf.sip.*;

public class ForwardOnBusy extends SipServletAdapter {
   SipServletContext context;
   boolean forwarded = false;
   Hashtable branches;

   /* init() */
   public void init(ServletConfig config) {
      super.init(config);
      branches = new Hashtable();

      /* Get context */
      context = config.getServletContext();
   }

   /* doInvite()
    * Get registered contacts and forward the request to the branches.
    */
   public boolean doInvite(SipRequest req) {

      /* Get contact info for the callee */
      SipAddress callee =
           context.createSipAddress(req.getRequestURI());
      List contacts = context.getContacts(callee);

      /* proxy request to all contacts */
      if (contacts!= null) {
         Iterator e = contacts.iterator();
```

```
        while(e.hasNext()) {
            ContactImpl contact = (ContactImpl)e.next();
            try {
                SipURL sipURL =
                        context.createSipURL(contact.getURI());
                Object requestToken = req.send(sipURL);
                branches.put(requestToken, contact.getURI());
            }catch(ParseException ex) {
                return false;
            }
        }
    }
    return true;
} /* doInvite */

/* gotResponse()
 * Check if this response is from branches that this servlet sent
 * to. If not, return false.
 * If so, remove the request token from hashtable and check if all
 * requestTokens are removed.
 * If all requestTokens are removed from hashtbale, then forward
 * the original request to the new branch.
 */
public boolean gotResponse(SipResponse res) {
    int status = res.getStatus();
    Object requestToken = res.getRequestToken();
    SipURL fwdURL;

    switch (status) {
    case SC_BUSY:
    case 486:  /* This is not defined in current SipServlet API */
    case SC_DECLINE:
        if (forwarded) return false;
        String branchURI = (String)branches.get(requestToken);
        if (branchURI == null)
            return false;
```

```
            else {
                branches.remove(requestToken);
                if (branches.size() != 0)
                    return false;
            }

            /* Get the original request for the current response */
            SipRequestImpl newRequest =
          new SipRequestImpl((SipTransactionImpl)res.getTransaction());

            /* Get Sip URL of the branch for forwarding */
            try {
                fwdURL =
              context.createSipURL("sip:ss2020@diamond.cs.columbia.edu");
            }catch(ParseException e) {
                return false;
            }

            /* proxy the origianl request */
            newRequest.send(fwdURL);
            forwarded = true;
            break;

          case SC_TRYING:
            return false;
          case SC_OK:
            return false;
          }
          return true;
       }/* gotResponse */
}
```

5.3 ForwardOnNoResponse



This servlet is very similar to the previous one except it has a timer. doRequest() starts a timer and defer the handling to sipd by returning false. Then, sipd forwards the call to the registered two branches. After 20 seconds later, there was no response from any contacts, it will forwards the original request to new branch. For a timer, I used Timer class and TimerTask class, which is supported in JDK1.2 or higher version.

```
import java.util.*;
import org.ietf.sip.*;

public class ForwardOnNoResponse extends SipServletAdapter {
   Timer timer;
   int waitingTime;

   /*
    * init()
    */
   public void init(ServletConfig config) {
      super.init(config);
      /* Set waiting time */
      waitingTime = 30;
   }
```

```
    /*
     * doInvite()
     * Start timer.
     */
    public boolean doInvite(SipRequest req) {
        timer = new Timer();
        timer.schedule(new             ForwardTask((SipRequestImpl)req),
waitingTime*1000);
        return false;
    }

    /*
     * gotResponse
     */
    public boolean gotResponse(SipResponse res) {
        int status = res.getStatus();
        Object requestToken = res.getRequestToken();
        SipURL fwdURL;

        switch (status) {
        case SC_BUSY:
        case 486:  /* This is not defined in current SipServlet API */
        case SC_DECLINE:
        case SC_OK:
            timer.cancel();
        }
        return true;
    }
}

import org.ietf.sip.*;
import java.util.*;

public class ForwardTask extends TimerTask {
    SipRequestImpl req;
```

```
    public ForwardTask(SipRequestImpl req) {
        this.req = req;
    }

    public void run() {
        SipURL fwdURL;
        SipServletContext context = config.getServletContext();
        try {
            fwdURL =
        context.createSipURL("sip:ss2020@diamond.cs.columbia.edu");
        }catch(ParseException e) {
        }
        req.send(fwdURL);
        cancel();
    }
}
```

6 Acknowledgement

I would like to thank Professor Henning Shulzrinne, Jonathan Lennox, Kundan Sign, Sankaran Narayan for their comments and advices.

7 References

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. Don Libes, Writing CGI scripts in Tcl.

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

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

6. Jonathan Lennox, The CINEMA LIBSIP policy API.

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

8. Sheng Liang, The Java Native Interface Programmer’s Guide and Specification

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