Matthias Rusinski
msr52@columbia.edu

Columbia University
January, 2001

SIP SERVLET SUPPORT FOR SIPD

documentation

1. Introduction

Project described here provides SIP servlet support for SIP server (sipd). SIP servlet support developed in this project was done according to specifications described in the Internet draft: “The SIP Servlet API” by A. Kristensen and A. Byttner. The system provides the following:

- implementation of Java classes defined by SIP servlet API: Implementation adheres to specification provided in the draft, and allows users to write SIP servlets that can process requests and responses received from the server.
- SIP servlet engine: Servlet engine is provided which maintains state of transactions processed by user servlets, and which provided mechanism of passing requests and responses to user servlets.
- interface between sipd and servlet engine: Interface that allows transformation of C structures into Java objects (and Java objects into C structures) is provided (implemented with Java Native Interface (JNI)). One component of the interface is set of sipd policy functions (in C) for sip servlet (sips) which instantiates and initializes Java objects in JVM that correspond to sipd structures. The other component is found in the servlet engine where Java objects initialized to values of sipd structures are used to create Java objects based on SIP servlet API.

2. General Architecture

SIP servlet support for sipd consists of several elements. The architecture of the system spans C code internal to sipd and additional C functions for handling JNI (Java Native Interface), servlet engine class running in JVM, user provided Java classes (servlets), and native libraries which are used to execute some SIP API method back in the C environment.

The general system architecture is described based on the order in which a typical request/response would be processed by sipd and its servlet support.

The system processes requests and responses in both C and Java. Interaction between the two is handled with JNI. Thus, Java Virtual Machine must be present and running throughout the lifetime of the server. In this architecture, main routine of sipd server (main.c in sipd) would start JVM and the servlet engine which would be running in a separate process on the system. Servlet engine should be started before sipd begins to receive and process any messages, and to start the servlet engine, JVM must be started first (starting JVM is done with a function call, as is starting of servlet engine: these functions are located in file sipJVM.c, and eventually should be integrated into sipd). Sipd must provide a global pointer which will identify JVM to all functions that will be called during processing of a request (the pointer is initialized when JVM is started). From that point, any C functions written for SIP servlet support assume that JVM is present and running, and so is the servlet container (engine), and that JVM is identifiable by that global pointer (see file main.c which does this).

Transaction processing for SIP servlet support starts with C functions that define SIP servlet policy for sipd. (see sips_interface.c). Sipd should choose to execute SIP servlet policy when it finds that given user has servlet code that will process requests and responses of the transaction. It must be noted at this point that SIP servlets, from the point of view of sipd, are described by type script_t which is also used to describe other user scripts like CGI, CPL, etc. (When SIP servlet support is fully integrated into sipd, SIP servlets should be recognized by script_t by defining new disposition for servlets inside script.c). Servlet policy functions are called for initiating a new transaction, processing a request, processing a response, and destroying a transaction. Each of the functions provides a bridge between Java and C. Based of request_t, message_t and other types received from sipd, each of the functions will instantiate a corresponding object in JVM, and will initialize it to corresponding values from sipd C structure (that is done with functions from JNI_util.c). Once that is done, the policy function will call a Java method in the servlet engine and pass these objects to the engine(container) with the method (servlet engine is implemented in SipSContainer.java). The servlet engine's Java method will in turn use sipd-like objects to construct objects defined by SIP API draft. For that, the engine uses implemented classes of SIP Servlet API (these classes are accessed by both the servlet engine and user servlets). [All implemented SIP Servlet API classes carry the name of the Java interface they implement with suffix "Impl" at the end (user and his/her servlet should not be aware of this).]

Sip servlet container does not initiate any new threads within itself. In this system, the C function which already executes in its own thread attaches its thread to JVM. This assumption is taken by the servlet container. In addition, its methods are static methods (which can be called from any threads) where access to crucial and shared objects is synchronized which makes them thread-safe.

On transaction initiation, the servlet engine will construct SipTransaction class and keep its instance throughout the lifetime of the transaction. At the same time, user servlet is loaded. Servlets are loaded from file system (they are stored according to script_t rules). Special class, ServletClassLoader.java, is responsible for reading byte codes from file system and returning a Java defined class which can be instantiated (the class loader will attempt to compile the source (.java) if one is found when byte code (.class) cannot be located). The instance of the servlet is than associated with the transaction and is kept active as long as the transaction (as defined by SIP servlet API, the servlet is a listener object in that transaction). On the other hand, on transaction clean up or timeout, the transaction is destroyed and removed from the container's list of active transactions and servlet instances.

When request or response is received, servlet engine uses received request_t and message_t types to construct SipRequest or SipResponse objects and call servlet's gotRequest or gotResponse methods. At that point, servlets are free to operate on those objects with methods provided by SIP servlet API. It is worth noting that in the current implementation of servlet API, objects like SipRequest and SipResponse (which extend SipMessage) retain their message_t type (which was used to create them), and updates to headers of the message, content, etc. are also performed in the underlying message_t object (reason for this is explained next).

Special attention must be paid to "send" method of request and response objects. This method goes back to native C code to execute. First, the request_t and message_t objects are prepared (still in Java) to contain the values of the request or response to be sent (actually, message_t type is not used for this but is also sent and translated to C structure; this is done in case the method is to be expanded in the future and the need to message_t arises). Response headers are placed to "headers" field, request's URI or response's status and reason are placed into parser_t field, and socket address structures are initiated with destination. Such formed object is sent to native method (sipRequestWrapper.c or sipResponseWrapper.c). These methods are built into dynamic native libraries which are loaded and executed from a Java method (since this is the only way JNI allows this to be done) at run time. Once in C, the native method will attempt to allocate and recreate original sipd structure from values received in the Java object (see sips_wrapper.c). When done, the message can be prepared and sent over the network using sipd functions (with which the native library is built) from libsip and libcine.

3. Directory Structure

The project is located in its own directory: sips inside the cinema directory tree. 'sips' directory contains the following subdirectories:

- main: This is the directory with the crucial parts of the system. The current testing program is also run from here. The directory contains all C files for the project (main.c, sips_interface.c, JNI_util.c, sipJVM.c, sipRequestWrapper.c, and so forth). It also included Java classes for servlet engine and servlet loader. In addtion, the project is run and compiled from this directory, so it also contains compile script, makefile, and, after the compilation: executable and dynamic libraries.
- sips_api: This directory contains specification of SIP Servlet API and its compiled Java framework which must be visible (in the class_path) for both the servlet engine, user servlets, and classes that implement those interfaces.
- api_impl: This directory contains implementation of all necessary interfaces from SIP API. This implementation is used by the servlet container and servlets
- sipd_objects: Java classes that resemble sipd structures are defined and compiled this directory (they also must be visible throughout (in the class_path)).
- docs: The documentation for the project

4. Environment

Setup of environment and directory structure is of crucial importance in the project. This is dictated by the system having to access various Java classes and packages, and native libraries at run time.

First, the system requires use of Java 1.3 which can be found on CS machines in directory /usr/java1.3. The reason for this is that there are significant differences between JNI versions 1.1 and 1.2 and later, and the latest version of JNI has been used in this implementation. In addition, some classes used in Java API implementation are only available in release 1.3.

Second, particularly important are environment variables. The variables must be set in two places in order for the system to run: they must set up as system environment variables (UNIX-like environment variable) and as arguments to JVM when it is being started (see sipJVM.h for variable definitions, and sipJVM.c for how they are set).

The variables are:

- CLASSPATH: class paths must be set to the following directories within parent ‘sips’ directory of the project: sips_api, api_impl, sipd_objects, main
- LD_LIBRARY_PATH: this variable must point to directory ‘main’ within ‘sips’ which contains native libraries loaded at run time by Java methods

5. Integration into sipd

At this stage, the project implements Java classes for servlet container, SIP servlet API, JNI functions for translating the objects and structures, C functions for sipd policy handling, management of JVM and servlet engine (start up and shut down), and functions for sending request and responses over the network (which are built into native libraries).

However, integration of the project into sipd is not done. In general, the policy functions for sip servlet support must integrated into sipd (they also lack some processing logic at this stage) where they could be called from ‘execute_policy’ function. That also means that sipd must be built with all additional C functions provided by this project; startup and shutdown of JVM and servlet engine must be added to main routine in sipd.c. In addition, script_t must recognize new disposition: “sip_servlet” for properly retrieving users’ servlets and passing that information to sips policy functions. Sip servlet policy function must be supplied with additional logic for message processing for special conditions which were not taken into account when developing this project.