# ECharts for SIP Servlets User Manual

Gregory W. Bond  Eric Cheung  Thomas M. Smith
Venkita Subramonian
*AT&T Labs – Research*

Version 2.5 Beta
April 9, 2009

The latest version of this document is available from:
http://echarts.org

**Abstract**

This document describes the ECharts for SIP Servlets (E4SS) framework and development kit. E4SS makes use of the ECharts programming language as an alternative programming model built on the SIP Servlet API defined by the Java Community Process. Instead of overriding servlet methods and managing sessions, the programmer defines the application in terms of feature boxes, SIP message ports, and state-machine logic specified in the ECharts language. The use of ECharts to specify application logic greatly encourages component re-use and manages complexity, making it well-suited for Back-to-Back User Agent applications. Such applications can be composed with one another and with suitable proxy applications in order to provide feature-rich services while maintaining feature modularity, through use of a SIP Servlet Application Router. This document explains the E4SS framework abstractions and provides example-driven descriptions of the tools and libraries included with the E4SS development kit. The document also provides instructions on how to configure two common SIP containers, SailFin and OCCAS, for use with E4SS.

# Contents

# 1 Introduction

The SIP Servlet API defined by JSR 289 provides a standard programming model for SIP applications [1]. Building on the Generic Servlet framework, this API provides a familiar approach for Java developers coming from the world of HTTP servlets. Application logic is controlled by overriding methods like `doInvite` and `doSuccessResponse`.

Telecommunications services in general lend themselves to specification by finite state machines (FSM). By providing an adaptation layer between the SIP Servlet API and the ECharts state-machine programming language, we can bring to bear the expressive power of ECharts to servlet containers that are compliant with the SIP Servlet API. ECharts features such as machine parameterization and well-defined transition priorities then make re-use of FSM logic modules easy and attractive [2].

Such an adaptation layer is described here. A programmer need not know the details of the JSR 289 standard to use ECharts for SIP Servlets, though none of the features of the API are hidden from programmers who wish to use them. In this adaptation, a complete SIP application can be built by specifying one or more ECharts Machines that describe the desired logic. The ECharts code is translated into Java and compiled and deployed as a native Java SIP Servlet application. This document describes the programming model and the process of building such an application, as well as additional capabilities for application composition and automatic termination handling.

For the remainder of the document, familiarity with ECharts and the SIP Servlet programming model is assumed.

# 2 Programming Model

The state-machine formulation of ECharts makes it well-suited for back-to-back user agent applications. These applications generally need to manage the dialog state of two or more SIP dialogs, in addition to application state. As such, it is recommended that ECharts machines be used for such applications. However it is often convenient to compose such applications with proxy applications for such applications as routing. In Section 3 we discuss use of these proxy applications; however for the remainder of the document, B2BUA-style applications are assumed.

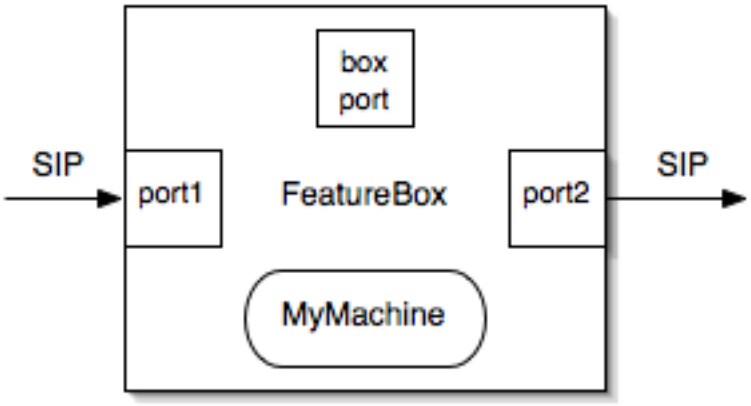



Figure 1: A sample FeatureBox with two ports

SIP Servlet applications make heavy use of *sessions*. A `SipSession` is used to maintain state related to a particular SIP dialog. A `SipApplicationSession` groups together related `SipSessions` as well as other application-level data.

ECharts for SIP Servlets exposes a different object model. `Ports` are used as entry and exit points for SIP messages between the application and a remote SIP entity, and as such subsume the role of the `SipSession`. A `FeatureBox` contains one or more ports, as well as a `Machine` that specifies the actions that should occur based on events processed by the ports, and therefore maps to a `SipApplicationSession`.

A special `SipServlet` subclass is provided as a gateway to ECharts. This servlet, `EChartsSipServlet`, does not need to be modified or subclassed by the application developer. The only piece that needs to be supplied is a custom ECharts `Machine` that creates the required ports and specifies the application logic. All other components are defined and supplied by the adaptation layer.

Note that this software does not currently allow for `distributable` servlet applications.

## 2.1 FeatureBox

The `FeatureBox` is a container for all components necessary for an application instance. In this context, *application instance* refers to a specific instantation and execution of the application logic in reaction to the environment — simplistically, a single "call." This is in contrast to a *servlet instance*, which may serve a large number of "calls." See Section 2.6 of JSR 289 for further details.

A simple though common `FeatureBox` configuration is shown in Figure 1. The box includes two ports for SIP messaging, as well as a custom ECharts `Machine` specified in `MyMachine`. For the simple case of a back-to-back user agent (B2BUA), `port1` could process messages from and to the calling party, and `port2` could process messages for the called party.

When an initial request is received by the `EChartsSipServlet`, a new instance of a `FeatureBox` is created and the request passed to the box on a special port called the `BoxPort`, of which there is exactly one per box. The reason for this is explained in Section 2.2.2.

## 2.2 Ports

Ports are the mechanism by which messages are received and sent. The bulk of the work in an application is done by processing SIP messages on `SipPorts`. However there are three other important types of ports, `BoxPort`, `TimerPort`, and `NonSipPort`. All are described below.

### 2.2.1 SipPort

A `SipPort` is associated with a single SIP dialog (and hence a single `SipSession` as defined in JSR 289). Messages received on a dialog are processed by defining message transitions on the corresponding `SipPort`. For example, the transition below will fire if a `180 Ringing` message is received on a port named `callee`.

```
// absorb a 180 Ringing message from callee
transition INVITE_SENT - callee ? ProvisionalResponse180
-> INVITE_SENT;
```

Messages to be sent on a `SipPort` must be created by calling a method on that port. This will ensure that the message is created on the appropriate `SipSession` to satisfy the requirements of JSR 289. In addition, they must be sent via the ECharts syntax shown below, instead of the `message.send()` syntax of the SIP Servlet API. For instance, to create and send a BYE message on a port named `caller`, the following ECharts action could be used:

```
caller ! caller.createRequest("BYE")
```

Recall that this is the ECharts syntax for:

```
send the message returned by caller.createRequest("BYE")
on the caller port.
```

The processes of creation and sending are separated, so that the message may be manipulated (or saved) after creation and before sending.

### 2.2.2 BoxPort

The `BoxPort` is a special port (one per box) that is used to handle initial requests. This allows the application the flexibility to decide which `SipPort` to associate with the request at execution time. *This is a mandatory step,* though the step may be embedded in a child machine, as it is in the B2BUA sample. In the code snippet below, a received INVITE is bound to the `caller` port; all subsequent messages on the corresponding `SipSession` will be associated with the `caller` port.

```
boxPort = box.getBoxPort();
...

transition INIT - boxPort ? Invite /
    savedInvite = message;
    caller.bind(message);
-> PROCESS;
```

### 2.2.3 TimerPort

In addition to SIP messages, the other environmental event that can fire transitions is a timeout event, resulting in a `timed transition`. These delays are fired on anonymous `TimerPorts`. The following transition will fire after a specified ring-no-answer timeout:

```
transition RINGING - delay(ringNoAnswerTimeout) -> NO_ANSWER;
```

The programmer does not explicitly declare and reference `TimerPort` objects; using the `delay` syntax above causes the appropriate code to be generated to create and manage the required ports and messages.

In order to use timed transitions, the SIP application's deployment descriptor must be configured correctly. See Section 5.2 for details.

### 2.2.4 NonSipPort

This class is included to enable the possibility of receiving messages from sources other than SIP messages. For instance, this mechanism could be used to support converged (SIP + HTTP ) applications.

## 2.3 Messages

The SipServlet API defines a base message interface (`SipServletMessage`) and two subinterfaces (`SipServletRequest` and `SipServletResponse`). These interfaces may be used in message transitions. However, for programmer convenience and expressive power, a number of additional message classes have been defined. So instead of writing:

```
transition CONNECTED
    - callee ? SipServletRequest [ message.getMethod().equals("BYE") ]
-> TEARDOWN;
```

the more concise (and readable) form shown below can be used:

```
transition CONNECTED - callee ? Bye -> TEARDOWN;
```

| Message Class | Implemented Interface |
|---|---|
| `Invite` | `SipServletRequest` |
| `Reinvite` | |
| `Ack` | |
| `Bye` | |
| `Cancel` | |
| `Info` | |
| `Register` | |
| `Options` | |
| `Message` | |
| `Subscribe` | |
| `Notify` | |
| `Response` | `SipServletResponse` |
| `Prack` | |
| `ProvisionalResponse` | |
| `ProvisionalResponse180` | |
| `ProvisionalResponse183` | |
| `FinalResponse` | |
| `SuccessResponse` | |
| `SuccessResponse200` | |
| `RedirectResponse` | |
| `ErrorResponse` | |
| `ErrorResponse486` | |
| `ErrorResponse487` | |

Table 1: Additional Message Classes

---

E4SS wraps an incoming message in the most appropriate message class but stores messages in their unwrapped form. The additional E4SS message classes are shown in Table 1, as well as the interface that each implements. The `Response` class has a subclass hierarchy which can be inferred from the class names.

For messages that are *sent* rather than *received*, an appropriate `createXXX` method must be called on a `SipPort` to ensure that the message is part of the correct `SipSession`. The methods include `createInvite`, `createRequest`, and `createResponse`. See the javadoc for details.

## 2.4 Automatic Termination Handling

Telecommunications applications have widely divergent purposes, but all such applications must eventually terminate their calls. In SIP, the message used to terminate a call depends on the state of the SIP dialog. Depending on the state, a call might be properly terminated by sending BYE, sending CANCEL, or sending an error response to a received INVITE, and the resulting messages that are expected in response vary in each case. To support this universal need while reducing application complex-

ity, we have used ECharts for SIP Servlets to create a framework to automatically terminate calls under certain conditions.

If an application uses a standard B2BUA pattern (one call in, one call out, and either party may end the call), then the B2BUA software supplied in the open-source distribution will handle termination just like any other mid-call request, by simply relaying a received BYE message to the other party, and then relaying the received response. However more complex applications may involve more than two parties or may need to terminate calls based on an application-specific trigger. To support these cases, we have created a framework for automatic termination handling. This behavior applies *only* to `INVITE` dialogs.

The termination handling framework tracks the `INVITE` dialog state of each `SipPort` in a `FeatureBox`. Updates to the state are made automatically based on SIP messages sent and received by the application. If a trigger condition occurs indicating the need to terminate the application, then a new termination machine is dynamically created for each port, all executing concurrently to tear down all the application's calls according to the SIP protocol. This port termination machine can also be used explicitly in an application to selectively tear down the calls on one or more ports.

There are three conditions which trigger automatic termination behavior:

1. *Application-initiated teardown* - This condition is used when the application wishes to initiate teardown on its own. This could occur, for example, when a prepaid calling card balance goes to zero. This condition is initiated when the application machine transitions to a terminal state, i.e., a simple state with no outgoing transitions.

   ```
   /** end call if timer expires
    */
   transition IN_PROGRESS -
       delay( maxTime )
   -> END;

   /** terminal state
    */
   state END;
   ```

   Tying termination behavior to a terminal state is reasonable, as the application has declared that it has no further actions to take.

2. *Unhandled teardown message* - Another condition which results in automatic termination behavior is the receipt of a message that indicates that a call should be torn down (BYE or CANCEL), in the case where the application does not specify how to handle such a message. This frees the application developer from having to specify how to properly tear down each call at each point in the application. If the application should *not* terminate when it receives a teardown signal (such as in a sequence calling application), the developer is free to specify other (non-default) behavior.

3. *Exception handling* - The final condition under which automatic termination behavior is initiated is in the event of a programming error (specifically an uncaught exception). We chose this because there is no general way to judge the programmer's intent, and it is prudent in this case to clean up the calls to release resources.

# 3 Use of Proxy Applications

In addition to the B2BUA-style applications developed in the ECharts language, it is often useful to compose SIP proxy applications to provide routing or other pre-connection application logic. We have supplied a special proxy servlet that works with the application router and provides logging capabilities that are consistent with other ECharts applications. This `EChartsProxyServlet` can be subclassed to provide application-specific logic. See the E4SS javadocs for more details.

# 4 Application Composition

For reasons of software modularity or system integration, it is often necessary or desirable to have more than one SIP application in the call path. This capability is addressed by the application routing mechanism specified by JSR 289 [1]. The E4SS development kit includes a powerful application router component, called the DFC application router (DFC AR), that can be used if desired to compose E4SS (or non-E4SS) applications. For detailed information on the DFC application router, see [4]. For information on how to configure a container to use the DFC application router, see Section A in this manual.

## 4.1 DFC AR Configuration

The DFC AR rules are specified using XML in a file named *approuter.xml*. See `[EDK_HOME]/etc/approuter.sample.xml` in the ECharts for SIP Servlets development kit for an example. The deployment location of the approuter.xml file is container-dependent. See Section A for details.

### 4.1.1 Subscription Rules

Two sets of rules are specified in the file: one for applications in the originating region and another for applications in the terminating region. For each rule (enclosed within a `<originating-region-mapping>` or `<terminating-region-mapping>` element), an address pattern is specified based on which an initial request is routed to the appropriate application. For an application subscribed to in the originating region, the address in the *From* header in the initial request is used to match against address patterns specified in the originating region. For an application subscribed to in the terminating region, the Request-URI in the initial request is used to match against address patterns specified in the terminating region. Standard Java regular expressions can be used to specify address patterns.

Consider an example where the goal is to have an initial request routed to an application *apporig*, when the caller is Alice whose SIP address is `''Alice'' <sip:alice@foo.com>`. Moreover, the initial request should subsequently get routed to another application *appterm*, when the callee is Bob whose SIP URI is `sip:bob@foo.com`.

To achieve this routing, we specify a `<originating-region-mapping>` based on Alice's SIP address so that an initial request would be routed to *apporig* application. When the SIP address in the *From* header of a initial request matches the address pattern specified in this rule, the application router inserts an *app* parameter in the Request-URI with a value of "origapp". *Note that while specifying the address*

*pattern on the originating side, the display name part of a SIP address should also be accounted for.* This is different from address pattern specification for the terminating region which specifies a pattern for matching the SIP URI (as opposed to a SIP address) in the Request-URI.

```
<originating-region-mapping>
  <mapping>
    <address-pattern>.*<sip:alice@.*</address-pattern>
    <app-name>origapp</app-name>
  </mapping>
</originating-region-mapping>
```

Note that after the above rule is matched by the application router, the actual routing of an initial request is done by the servlet mapping mechanisms within the container.

To achieve routing to the *appterm* application on the terminating side, we specify a `<terminating-region-mapping>` based on Bob's URI so that an initial request would be routed to the *appterm* application.

```
<terminating-region-mapping>
  <mapping>
    <address-pattern>sip:bob@.*</address-pattern>
    <app-name>termapp</app-name>
  </mapping>
</terminating-region-mapping>
```

### 4.1.2 Precedence Rules

When there are multiple applications subscribed to within one region, then a precedence relationship among these applications can be specified using the `<precedence>` section as shown below.

```
<precedence>
  <terminating-region>
    <ordering>
      <app-name>app4</app-name>
      <app-name>app3</app-name>
    </ordering>
 </terminating-region>
  <originating-region>
    <ordering>
      <app-name>app1</app-name>
      <app-name>app2</app-name>
    </ordering>
 </originating-region>
</precedence>
```

The precedence relationship specifies the order of invocation for the applications within a region. For example, the above fragment from a *approuter.xml* file specifies that application *app1* should be invoked before *app2* in the originating region and *app3* should be invoked before *app4* in the terminating region. *Note that the first application in the list for the originating side has the highest precedence and the last*

*application in the list for the terminating side has the highest precedence.* In either region, the application closest to the subscriber is specified as the first application and the application furthest from the subscriber is specified as the last application.

# 5 Creating an Application

Creating an application is as simple as defining an ECharts `Machine` encapsulating the desired logic and configuring an appropriate deployment descriptor. Each step is explained below. In Section 7 we describe a tool called `appgen` that assists with these steps. In Section 8 we describe a sample application that is included with the ECharts for SIP Servlets development kit.

## 5.1 Defining the Machine

Machines are created in `.ech` files as described in the ECharts Manual [2]. A typical machine declaration is:

```
package examples;

// machine that rejects all calls with "486 Busy Here"
public machine BusyFSM {
    ...
}
```

The machine will contain one or more constructors, state declarations, state transition rules, and any supporting Java code required to support the desired logic.

### 5.1.1 Creating Ports

The machine logic is required to create any ports required for the application (except for the `BoxPort`, which is automatically supplied to each box). The machine constructor provides a convenient place to create the required ports, as shown here:

```
package examples;

// machine that rejects all calls with "486 Busy Here"
public machine BusyFSM {
    <* SipPort caller; *>

    // top-level machines are called with this constructor
    public BusyFSM(FeatureBox box,
                   Properties servletProps,
                   ServletContext context) {
        this.caller = box.createSipPort("caller");
    }
    ...
}
```

### 5.1.2 Machine Constructors

A machine may have multiple constructors. This can promote reuse of the machine by allowing it to be easily invoked in more than one context. One typical use of multiple constructors is to allow a machine to be invoked directly both by the `EChartsSipServlet` (thus making it a *top-level* machine) or by another machine (thus making it a *nested* machine). A significant difference between the two contexts is that top-level machines generally create the ports they need, while nested machines may have ports passed in as parameters.

The example below shows multiple constructors, and adds the application logic as well:

---

```
package examples;

import java.util.Properties;
import javax.servlet.ServletContext;
import org.echarts.servlet.sip.*;
import org.echarts.servlet.sip.messages.*;

// machine that rejects all calls with "486 Busy Here"
public machine BusyFSM {
    <*
        FeatureBox box;
        SipPort caller;
        BoxPort boxPort;
    *>

    // top-level machines are called with this constructor
    public BusyFSM(FeatureBox box,
                       Properties servletProps,
                       ServletContext context) {
        this.box    = box;
        this.caller = box.createSipPort("caller");
        boxPort = box.getBoxPort();
    }

    // allow caller port to be passed in by another machine
    public BusyFSM(FeatureBox box, SipPort caller) {
        this.caller = caller;
        boxPort = box.getBoxPort();
    }

    initial state INIT;

    transition INIT - boxPort ? Invite / {
        caller.bind(message);
        caller ! caller.createResponse(486);
    } -> REJECTED;

    state REJECTED;

}
```

For such a simple application, there is no advantage to using the ECharts framework. Once the feature logic gets more complex, and the library of well-designed and debugged machines grows, the advantages become clearer.

## 5.2 Creating the Deployment Descriptor

In order to deploy an application, an appropriate deployment descriptor must be crafted. The servlet to invoke is always `EChartsSipServlet`; the `Machine` to invoke is specified by the `machineClassName` initialization parameter to the servlet. A special `listener` is required if any machines implement timed transitions.

Here is a sample `sip.xml` file for an E4SS application.

```
<?xml version="1.0" encoding="UTF-8"?>
<sip-app xmlns="http://www.jcp.org/xml/ns/sipservlet"
    xmlns:javaee="http://java.sun.com/xml/ns/javaee">

<app-name>busyapp</app-name>

<servlet>
<javaee:servlet-name>busyapp</javaee:servlet-name>
<javaee:servlet-class>
            org.echarts.servlet.sip.EChartsSipServlet
        </javaee:servlet-class>
        <javaee:init-param>
            <javaee:param-name>machineClassName</javaee:param-name>
            <javaee:param-value>examples.BusyFSM</javaee:param-value>
        </javaee:init-param>
        <!-- any other init-params are passed to machine
             as servletProps -->
        <javaee:load-on-startup>1</javaee:load-on-startup>
</servlet>

    <servlet-selection>
<servlet-mapping>
  <servlet-name>noAnswerTimeout</servlet-name>
  <pattern>
<equal>
  <var>request.method</var>
  <value>INVITE</value>
</equal>
  </pattern>
</servlet-mapping>
    </servlet-selection>

<session-config>
<javaee:session-timeout>0</javaee:session-timeout>
</session-config>

<listener>
<javaee:listener-class>
            org.echarts.servlet.sip.TransitionTimerManager
```

```
        </javaee:listener-class>
</listener>
<listener>
<javaee:listener-class>
            org.echarts.servlet.sip.EChartsSipServlet
        </javaee:listener-class>
</listener>
    <javaee:display-name>Busy Application</javaee:display-name>
</sip-app>
```

---

# 6 Creating Test Cases

The ECharts for SIP Servlets development kit includes the KitCAT (Kit for Converged Application Testing) testing framework [5]. KitCAT is a Java library for functional testing of SIP/RTP applications and is released open source (available from http://echarts.org). The following are some of the key features supported by KitCAT for functional testing:

- A single test case hosting multiple test SIP/RTP endpoints.
- High level primitives (*e.g.,* call, answer) that hide the test writer from protocol details; Primitives to support modifications to SIP messages such as adding/updating a header, where necessary
- Assertion primitives to validate conditions on endpoint states and messages sent/received by agents
- Ability to send/receive RTP stream (G.711 mu-law codec with 20ms packetization) to/from remote peer.
- Ability to send DTMF key sequence specified as a string.

More information about KitCAT can be obtained from http://echarts.org/KitCAT.

# 7 The Application Generator Tool

The ECharts for SIP Servlets development kit includes a handy tool for quickly generating the directories and files necessary for developing and testing a SIP servlet application: either a back-to-back user agent or proxy. After installing the development kit, set the environment variable `EDK_HOME` to the development kit's path and modify the system executable search path to include the development kit's *bin* directory. The *bin* directory contains the `appgen` executable. For example, on a Unix-based system running BASH use the commands:

```
export EDK_HOME=/usr/local/share/EChartsSipServlet_DK_2.5-beta
export PATH=$EDK_HOME/bin:$PATH
```

Then run the `appgen` command in the directory you wish to create your application directory in. The `appgen` tool will prompt you to answer a short series of questions and then generate a skeleton application directory. Here's an example of using it to generate a back-to-back user agent application:

```
% appgen
SIP ECharts APPLICATION CREATION WIZARD

Application type: b2bua or proxy?  [b2bua]
----> b2bua

Enter fully-qualified package name (last component is app name)
E.g., com.example.sipecharts.features.myApp
----> examples.features.myApp

Enter class name for main Machine
E.g., MyAppFSM
----> MyAppFSM

Enter a title for Javadocs
E.g., My Application
----> My Application

Creating source tree ...

APPLICATION CREATED in myApp directory
```

In this example, a subdirectory called *myApp* is created in the directory `appgen` was run in. The *myApp* directory contains a skeleton ECharts machine called *MyAppFSM.ech*, located in the application directory's *src/examples/features/myApp* directory, that is ready for you to customize. The *myApp* directory also contains the `ant` build files required to compile the application, generate a war file and, optionally, deploy the application. To customize the build to suit your environment edit the *build.properties* and *env.properties* files in the top-level application directory, and the *sip.xml* and *web.xml* files in the application's *SarContent/WEB-INF* directory. *appgen* also generates KitCAT-based test case templates and associated build files under the *myApp/test* directory. These test cases are JUnit-based and can be built and run by invoking *ant test* from the command line.

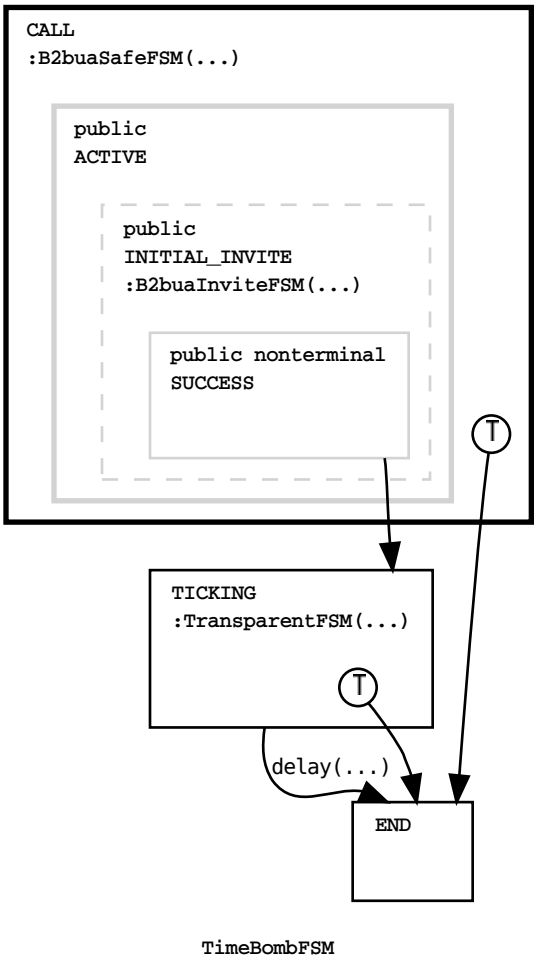
# 8 Sample Application

A non-trivial example application, called *TimeBomb*, is included in the *examples/timeBomb* directory of the development kit. This application allows two parties to talk for at most 5 seconds. *TimeBomb* is a back-to-back user agent application which, prior to timing out, modifies the Request-URI of the outgoing INVITE and then acts transparently.

---

```
/*************************************************************************
*                                                                        *
*               This software is part of the ECharts package             *
*                  Copyright (c) 2006-2009 AT&T Corp.                    *
*                      and is licensed under the                         *
*                  Common Public License, Version 1.0                    *
*                            by AT&T Corp.                               *
*                                                                        *
*************************************************************************/
package examples.timeBomb;
```

```
import java.util.Properties;
import javax.servlet.ServletContext;

import org.echarts.servlet.sip.*;
import org.echarts.servlet.sip.messages.*;
import org.echarts.servlet.sip.machines.*;

/** Example machine that places B2BUA call, then tears down call 5 seconds
 *  after call is connected.
 */

public machine TimeBombFSM  {
<*
    FeatureBox box;
    SipPort    caller;
    SipPort    callee;
*>

    /** Used when this FSM is specified as machineClassName by EChartsSipServlet
    */
    public TimeBombFSM(FeatureBox box, Properties servletProps, ServletContext context) {
            this.box    = box;
            this.caller = box.createSipPort("caller");
            this.callee = box.createSipPort("callee");
    }

    /** Initially, place a B2BUA call using a simple RequestModifier.
     */
    initial state CALL : B2buaSafeFSM(box, caller, callee, box.getDefaultModifier());

    /** Once the call is CONNECTED, we want to start the timer.
     */
    transition CALL.ACTIVE.INITIAL_INVITE.SUCCESS --> TICKING;

    /** While the timer is running, behave transparently.
     */
    state TICKING : TransparentFSM(caller, callee);

    /** After 5 sec in TICKING state, end calls by going to terminal state.
     */
    transition TICKING - delay(5000) -> END;

    /** Terminal state: simple state with no outgoing transitions.
     */
    state END;

    /** If the B2BUA reaches a terminal state before CONNECTED, then end execution.
     */
    transition CALL.TERMINAL --> END;

    /** If the call ends normally before the timer fires, end execution.
     */
    transition TICKING.TERMINAL --> END;
}
```




TimeBombFSM

---

The *TimeBomb* application makes use of several other general-purpose ECharts machines, either directly or indirectly, also included in the ECharts for SIP Servlets development kit.

- `org.echarts.servlet.sip.machines.B2buaSafeFSM` - This machine receives, binds, and saves the initial INVITE from `caller`, sends a new INVITE to `callee`, and manages resulting SIP messages until the `SUCCESS` state is reached, indicating that the caller and callee are connected. This state uses a nested `TransparentFSM` to propagate subsequent requests to the other port.
- `org.echarts.servlet.sip.machines.TransparentFSM` - This machine waits for requests on either port. Each time one is received, it dynamically spawns a new instance of `TransparentHandleRequestFSM`. If any of those instances reach the `END_BYE` state, then this machine reaches its `END` state, which in turn causes a transition in the parent machine to a terminal state.
- `org.echarts.servlet.sip.machines.TransparentHandleRequestFSM` - relay the request to the other port and handle the resulting SIP transaction. Depending on the type of request, various conditions are required for this machine to reach a terminal state. Each time it does so, its parent machine makes a `DEEP_HISTORY` transition to clean up the now-terminated machine (except in

the case of `END_BYE` which was described above.)

As you can see, by reusing the B2BUA logic it becomes a simple matter to provide default B2BUA behavior (which handles SIP errors setting up the call, CANCEL messages sent by the caller, re-INVITEs, etc.) while specifying an override for non-default behavior.

## 8.1 Building and Testing the Sample

The *TimeBomb* application directory structure, build files and deployment descriptors were created using the `appgen` tool described in Section 7. In this example we will show how to build the application and test it using the KitCAT test framework included with the E4SS development kit.

Prior to building the sample, change to the *test* subdirectory and then modify *build.properties* and *env.properties* as desired. In particular, you should modify the `ext.classpath` property in *env.properties* to include the two jar files necessary to build the example. The first jar file includes the classes defined in the `javax.servlet` package. The second jar file includes the classes defined in the `javax.servlet.sip` package. These jar files are included with any SIP servlet container. You must also either modify the *edk.home* property to point to your ECharts for SIP Servlets development kit location, or set the `EDK_HOME` environment variable to point to its location.

To build the sample type `ant fatsar`. This will build a war file with the name specified in `build.properties`. This war file can be deployed to your JSR 289–compliant container (for information on configuring a container to use E4SS, see Appendix A).

Prior to testing the sample, modify the file *test.properties* as necessary. Then to test the sample, type `ant test`. This will run the application test cases defined in `test/src/testcases/examples/timeBomb/test/TimeBombFSMTest.java`. For more information on application testing see Section 6.

Assuming E4SS logging is configured for your container (see Appendix A), then the application's log will contain a complete record of the SIP messages input and output by the application as well as a transcription of the transitions that fired during the execution of the application's ECharts machine.

## 8.2 Generating Documentation

To generate javadoc documentation for the example, you need to install the open source `graphviz` package, available from the *graphviz.org* website. This package is used to generate a graphical representation of application machines and integrates them with the javadoc pages. Then enter the command `ant javadoc`. This will generate the documentation in the *doc/api* directory. See the ECharts User Manual [2] for more information related to generating and viewing javadoc documentation generated this way.

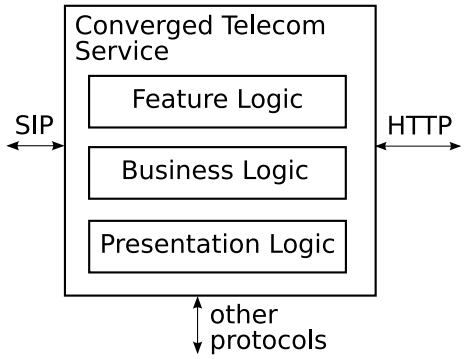



Figure 2: Converged telecom service components

# 9 Developing Converged Applications

A collection of telecom features, modular or monolithic, rarely stands alone. As illustrated in Figure 2, telecom features are normally an integral part of a larger service that may, for example, access a database or provide a web-based user interface. Services consisting of SIP and non-SIP components are called *converged* services.

In this section we describe a lightweight framework included wth ECharts for SIP Servlets that enables the development of modular, reusable telecom features suitable for integration into converged services. We show how the framework supports interaction between an ECharts for SIP Servlets feature and its external environment and how it facilitates the discovery task in different scenarios. Following an overview of the framework we discuss two features that use the framework. Both features are included with the ECharts for SIP Servlets Development Kit. Finally, we discuss a few technical details of the framework useful to developers.

## 9.1 Interfacing

To support development of a SIP feature and the non-SIP external components that it interacts with, an interface must be defined between them. The interface should expose only what is strictly necessary to the accessing component in order to minimize component inter-dependencies. To ensure that the interface is light-weight and that it accommodate the widest range of possible non-SIP environments, we have chosen to demarcate the SIP/non-SIP boundary close to the SIP feature logic. This means that the only constraint on the environment is that it be accessible via Java. Furthermore, if a heavier-weight interface technology is desired, for example SOAP, then it can be added on top of the lower level Java interface.

There are two common types of interaction between a feature and its non-SIP environment:

1. Interactions initiated by the feature such as reading provisioned data from a database or notifying the environment of service status. We call this type of interactions *SIP-to-Java* interactions.
2. Interactions initiated by non-SIP components such as initiating the creation of the feature in the first place, or exerting control over its behavior. We call this interaction type *Java-To-SIP* interactions.

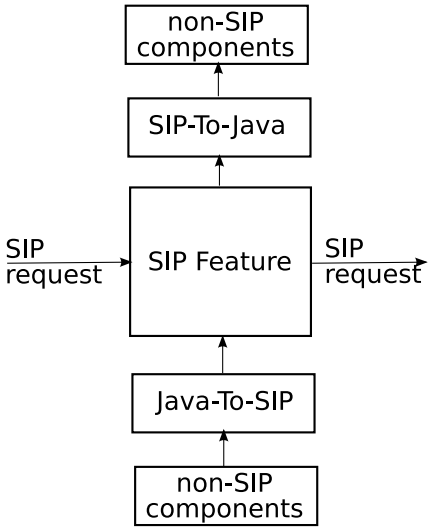



Figure 3: SIP service interfaces

Both interaction types are embodied as method calls. For example, a method would be called by the feature to read provisioned data from a database or a method would be called by the environment to exert control over a feature's behavior. Thus, for a given feature the two interaction types can be defined in terms of two interfaces: (1) method signatures comprising its SIP-To-Java interface and (2) method signatures comprising its Java-To-SIP interface. These two interfaces are illustrated in Figure 3. Both interfaces are defined by the feature itself since the decision of how a feature interacts with its environment affects the feature's internal design.

The implementation of a feature's Java-To-SIP interface methods is clearly the responsibility of the feature itself. This is because these methods will most likely access or manipulate the feature's internal state. On the other hand, the implementation of a feature's SIP-To-Java interface is clearly the responsibility of the external non-SIP components that interact with the feature. For example, when a feature calls a SIP-To-Java method to indicate its own status, the method will most likely update the state of some external component. Because the SIP-To-Java interface implementation is not the responsibility of the feature, the identity of the external component and the nature of the interaction with the external component needn't be known by the feature although these will be known by the SIP-To-Java interface implementation.

## 9.2 Discovery

Specifying interfaces between a SIP feature and non-SIP components constitutes only part of the solution. In general there can be many feature instances, each associated with their own calls. There may also be different external components associated with different calls. For this reason, an external component needs to be able to find the Java-To-SIP interface associated with a particular feature instance, and a feature instance needs to be able to find the appropriate non-SIP components via its SIP-To-Java interface. The nature of these discovery tasks differ depending on whether a feature instance is created in response to (1) the receipt of a SIP request, or (2) a

request by a non-SIP component.

In the first case, graphically depicted in Figure 4, the feature instance is created by the receipt of a SIP request from a SIP user agent, independent of non-SIP components. Most traditional call services are created this way, for example Call Forwarding, Call Waiting, etc. The feature instance first gets an instance of its SIP-To-Java interface from the framework. Since the implementation of this interface is the responsibility of external components, the interface implementation instance has embedded within it the information required to contact the appropriate external components. The feature instance then uses its SIP-To-Java interface to publish identifying information required by an external component to access its Java-To-SIP interface. This way, an external component can use this information to obtain the feature instance's Java-To-SIP instance from the framework.

The framework is responsible for creating, storing and retrieving interface instances; these actions are hidden from features and external components. Instead, these components request to `get` an interface instance using a unique feature instance identifier, denoted by `sasId` in Figure 4, as an argument. This identifier is actually the SipApplicationSession ID assigned to a SipServletApplicationSession instance by a container. There is exactly one SipApplicationSession instance associated with an EChartsSipServlet feature instance. The framework uses the `sasId` as a key to store and retrieve an interface instance associated with a SIP feature instance. When a component requests to get an interface instance, the framework returns the currently stored copy. If none exists then the framework first creates a new interface instance and stores it for subsequent retrieval.

In the second case, graphically depicted in Figure 5, an EChartsSipServlet feature instance is created by a non-SIP component via a call to `doNonSip()`, as would be the case with a Click-To-Dial service, for example. In this case, the feature instance needn't publish its `sasId` for discovery by a non-SIP component since an `sasId` is returned by `doNonSip()`. With this identifier, the non-SIP component is able to obtain a Java-To-SIP interface instance from the framework.

As shown in Figure 6, since the non-SIP component initiates the creation of the feature instance, we also permit the non-SIP component to provide the framework with a SIP-To-Java interface instance where the instance is created by the external environment itself. In this case, the `doNonSip()` call accesses the framework to store the specified interface instance value in addition to creating an EChartsSipServlet feature instance. This is a convenience that allows a non-SIP component to directly initialize a SIP-To-Java interface instance. The dual scenario in the SIP-initiated case, where a SIP feature sets the value of the non-SIP environment's Java-to-SIP interface instance is not applicable because the non-SIP environment is intended to be outside the purview of the feature in order to support feature reuse.

## 9.3 ClickToDial Feature

Figure 7 graphically depicts the Click2DialFlow1Machine that is included as part of the click2DialFlow1 feature distributed with the ECharts for SIP Servlets Development Kit. A comprehensive discussion of this feature can be found in [3]. In summary, the click2DialFlow1 feature is a converged feature that is initiated by non-SIP means. Once initiated, the feature calls a specified first party address and then, after the first party is connected, calls a specified second party address. When the second party is

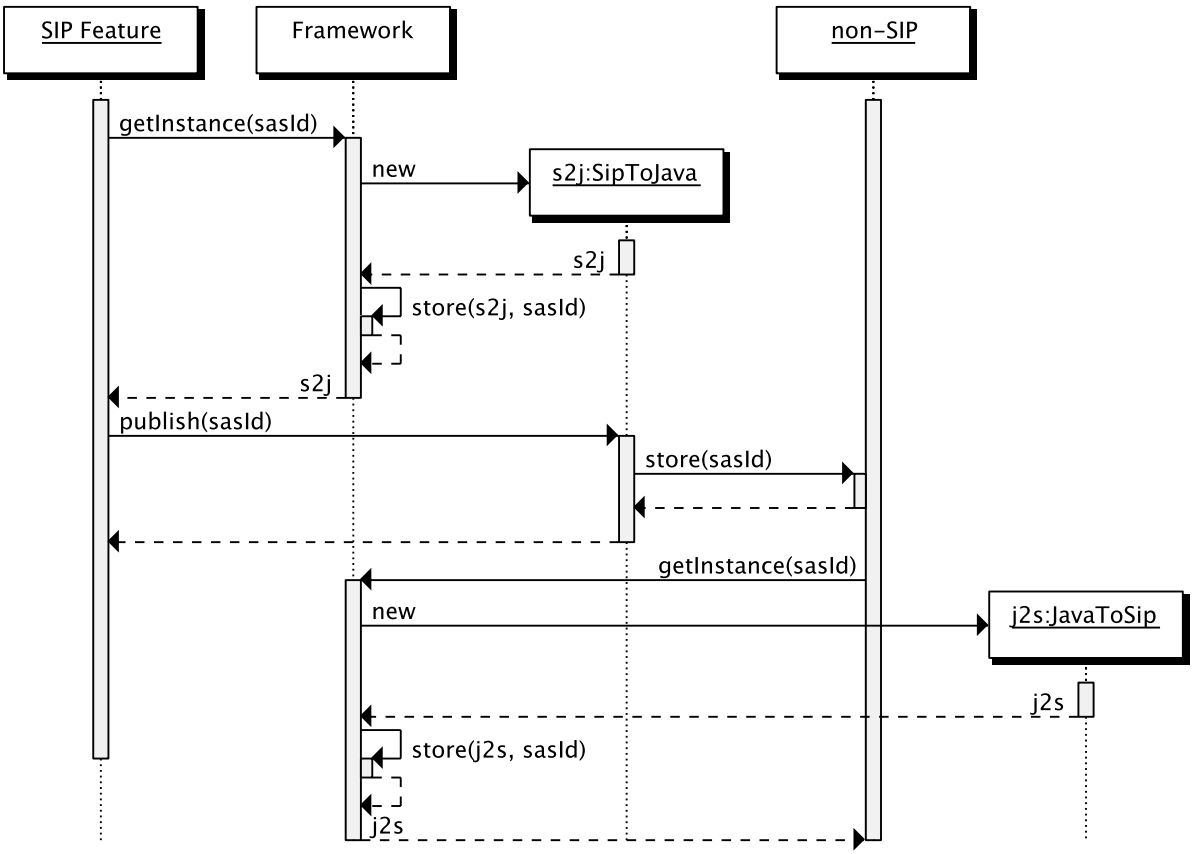



Figure 4: SIP-initiated interface discovery

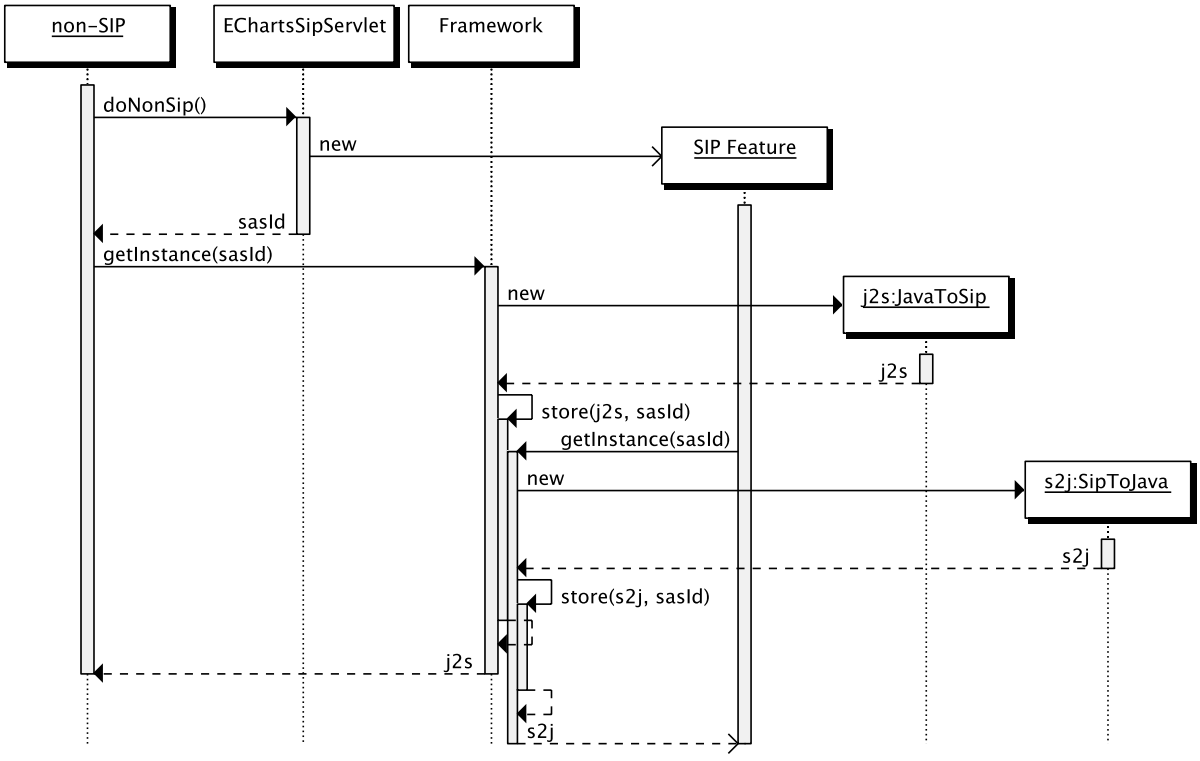



Figure 5: Non-SIP-initiated interface discovery - Scenario 1

connected, the two parties are connected to one another.

The click2DialFlow1 feature includes both a Java-To-SIP and a SIP-To-Java interface. Following our naming convention, the former is defined by the class JavaToClick2DialFlow1Machine and the latter by the interface Click2DialFlow1MachineToJava. A default implementation of the Click2DialFlow1MachineToJava interface is also included: Click2DialFlow1MachineToJavaImpl. The JavaToClick2DialFlow1Machine interface exposes a method, `dropCalls()`, for dropping its calls, and the Click2DialFlow1MachineToJava interface includes methods to indicate call status: `connected()` and `disconnected()`.

Since the click2DialFlow1 feature is initiated by non-SIP means then either of the non-SIP initiated discovery scenarios shown in Figures 5 and 6 can apply. The ClickToDialFlow1Machine provides a number of static `initiate()` methods that support both scenarios. All the `initiate()` methods call `EChartsSipServlet.doNonSip()` to create a EChartsSipServlet instance and its associated ClickToDialFlow1Machine instance. The Click2DialFlow1Machine instance calls `EChartsMachineToJava.getInstance()` in its constructor to obtain its Click2DialFlow1MachineToJava interface instance. The `initiate()` method also calls `JavaToEChartsMachine.getInstance()` to obtain a JavaToClick2DialFlow1Machine interface instance and returns the instance for subsequent use by the non-SIP caller.

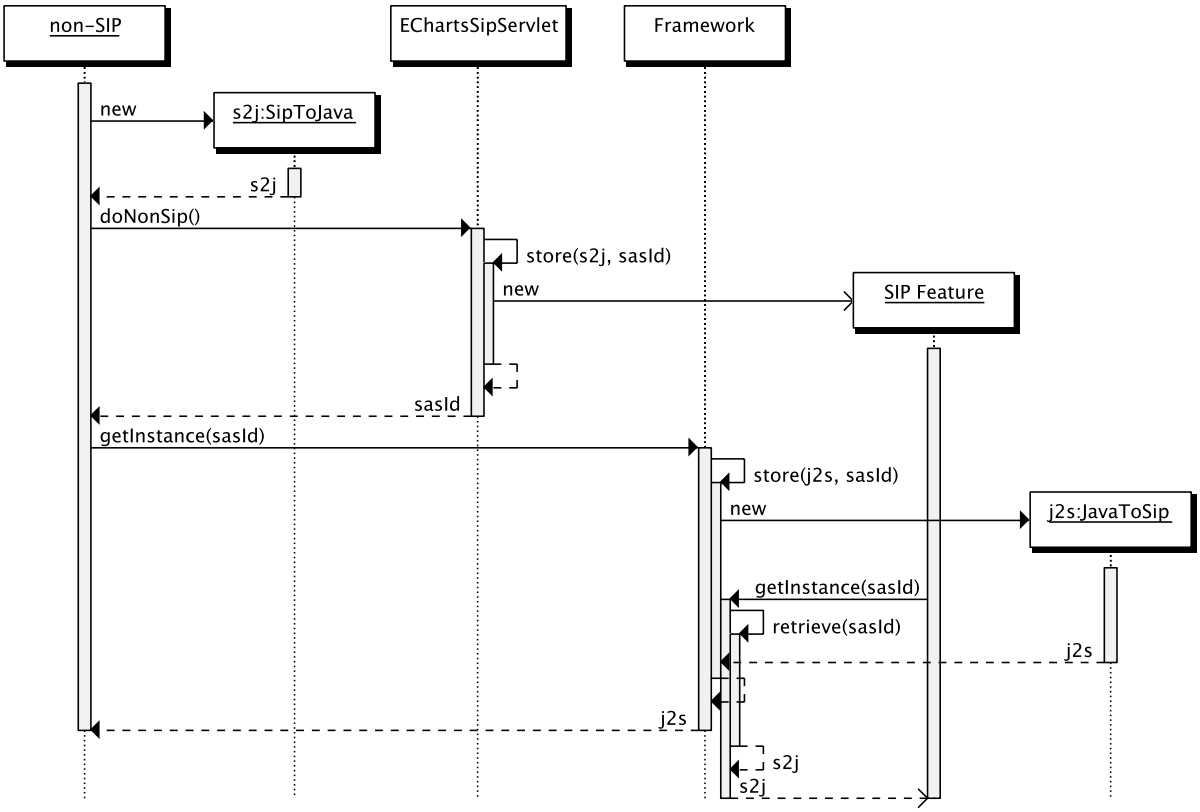



Figure 6: Non-SIP-initiated interface discovery - Scenario 2

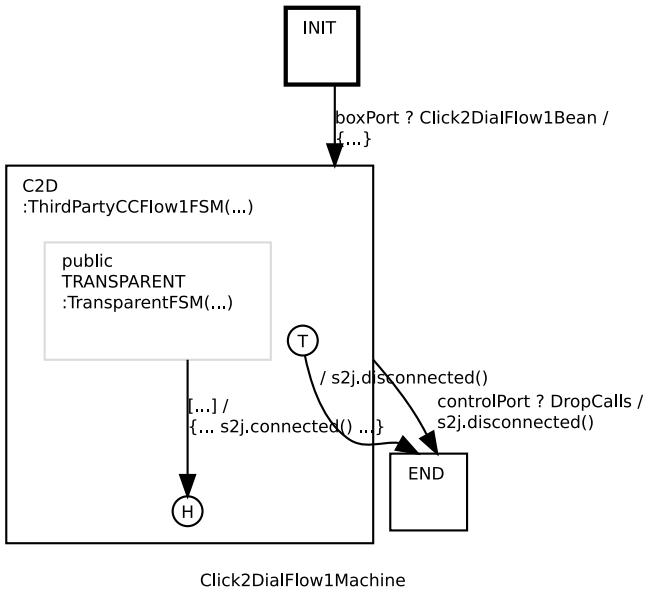



Figure 7: Click2DialFlow1Machine

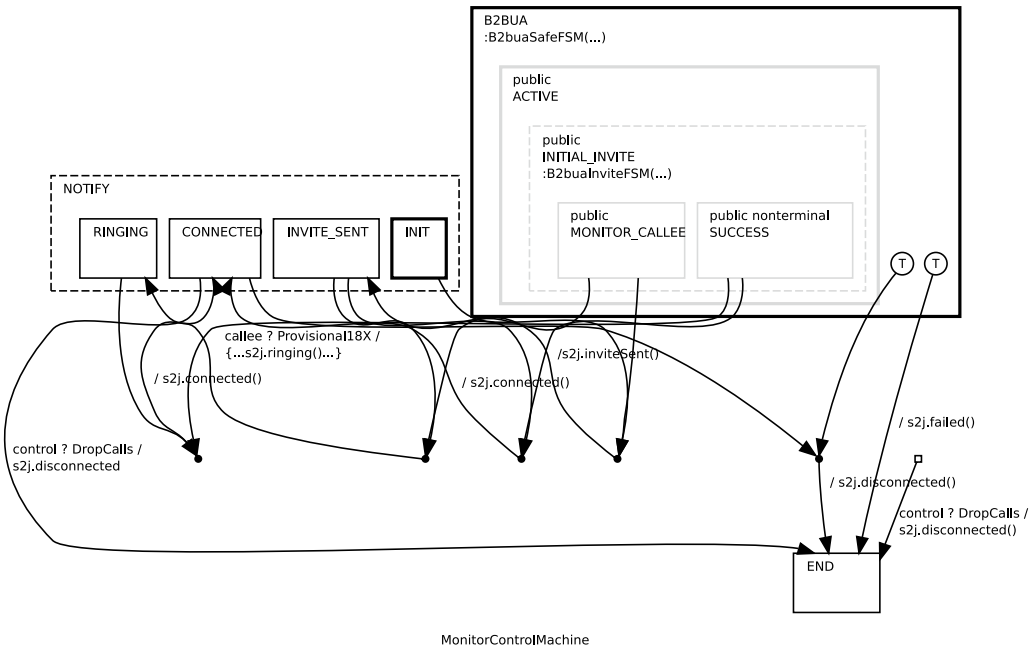



Figure 8: MonitorControlMachine

## 9.4 MonitorControl

Given the similarity to the previous example, we only briefly mention an example of a converged, SIP initiated feature. Such a feature is monitorControl, distributed with the ECharts for SIP Servlets Development Kit. This feature specifies both SIP-To-Java and Java-To-SIP interfaces and therefore follows the discovery scenario shown in Figure 4. Following the naming convention, the former is defined by MonitorControlMachineToJava and the latter by JavaToMonitorControlMachine. As shown in Figure 8, the MonitorControlMachine monitors call state and provides control over call state similar to the ClickToDial feature. In particular, the feature notifies the non-SIP environment of current call state (e.g. ringing, connected, disconnected) via its SIP-To-Java interface, and drops any in-progress call in response to a Java-To-SIP method call by the non-SIP environment.

Since the monitorControl feature is SIP-initiated and since it possesses a Java-To-SIP interface, it is the responsibility of the feature to publish its existence via a customized instance of its SIP-To-Java interface. To support publishing the identity of a feature box instance, the E4SS convergence framework automatically calls a method `publish()` when the machine gets its own SIP-To-Java instance via a call to `EChartsMachineToJava.getInstance()`. The default implementation of `publish()` does nothing so it is up to the developer to override the method in the MonitorControlMachineToJava interface implementation.

## 9.5 Using the Framework

This section provides more details for developers wishing to use the framework.

### 9.5.1 Framework Classes

First of all, the developer-visible parts of the "framework" are a number of static methods defined over three classes in the org.echarts.servlet.sip package: EChartsMachineToJava, JavaToEChartsMachine and EChartsProxyToJava. The first two classes constitute versions of SIP-To-Java and Java-To-SIP specialized for use with ECharts for SIP Servlets machines as defined by the EChartsSipServlet class. The last class is a specialized version of SIP-To-Java for ECharts SIP proxies as defined by the EChartsSipProxy class. We do not currently support Java-To-SIP interfaces for ECharts SIP proxies. The ECharts machine classes include a public `getInstance()` method to obtain an interface instance. The ECharts proxy class includes a similar public `newInstance()` method. See the ECharts for SIP Servlets javadocs for details on method parameters.

### 9.5.2 Java-To-SIP Interfaces

A Java-To-SIP interface class for an ECharts machine must abide by the following naming convention: if the ECharts machine is named `org.echarts.servlet.sip.features.monitorControl.MonitorControlMachine` then the interface class must be named `org.echarts.servlet.sip.features.monitorControl.JavaToMonitorControlMachine`.

When developing a Java-To-SIP interface class for a machine, the class should extend the JavaToEChartsMachine class in the org.echarts.servlet.sip package. The advantage of this is that the interface subclass will inherit a `putEvent()` method for logging events to the machine's monitor. It will also inherit an `init()` method that will be called when a new interface instance is created. This provides an opportunity to intialize the instance with machine-specific field values. See the JavaToEChartsMachine javadocs for more details.

### 9.5.3 SIP-To-Java Interfaces

As discussed above, a feature's SIP-To-Java interface must be implemented by the developer in order to customize it for its environment. By default, the framework attempts to use an interface implementation whose class name is related to the associated feature machine/proxy class name. For example, if the feature's machine is `org.echarts.servlet.sip.features.monitorControl.MonitorControlMachine`, then the framework will look for a SIP-To-Java interface class named `org.echarts.servlet.sip.features.monitorControl.MonitorControlMachineToJava`. To specify that the framework use an implementation class with the non-default name, its fully qualified class name must be provided as the value of an `init-param` element of the associated servlet's sip.xml file. See the EChartsMachineToJava and EChartsProxyServletToJava javadocs for more details.

When implementing a new feature with a SIP-To-Java interface, there is no need to include a default implementation of the interface with the feature. If the framework finds no implementation using the rules described above then it will dynamically create a default implementation of the interface that using Java's java.lang.reflect.Proxy class. Any interface methods that define return values will return null values when called.

When implementing a SIP-To-Java interface the developer can choose for the implementation to extend the EChartsMachineToJava (or the EChartsProxyServletToJava) class. The advantage of extending the EChartsMachineToJava class is that the implementation will inherit a `putEvent()` method for logging events to the machine's monitor (or the servlet's monitor, for an ECharts proxy servlet).

As mentioned above in Section 9.4, extending the EChartsMachineToJava class also has the advantage of calling a `publish()` method for you when an EChartsMachineToJava instance is created. This is useful if your E4SS feature also defines a JavaToEChartsMachine interface.

Finally, another useful utility class for E4SS proxies or machines is the EChartsSipSession class. This class defines a static method that provides the caller with the SIP message associated with the calling thread. This method means that it is not necessary to include the SIP message as a parameter in SIP-To-Java interface methods. Another static method in this class provides the caller with the SIP request that initiated an E4SS application session. This is useful when an application receives more than one initial request in its lifetime.

# References


[1] *SIP Servlet API Version 1.1.* JSR289, 2007. Available from: http://jcp.org/en/jsr/detail?id=289. 3, 9

[2] Gregory W. Bond. An introduction to ECharts: The concise user manual. Technical Report TD-6NKLR2, AT&T, 2006. Available from: http://echarts.org. 3, 11, 18, 28, 31

[3] Gregory W. Bond. Click-To-Dial With ECharts for SIP Servlets: A Case Study. Technical Report TD-7BEKZH, AT&T, 2008. Available from http://echarts.org. 21

[4] Eric Cheung and K. Hal Purdy. An application router for SIP servlet application composition. In *Proceedings of the IEEE International Conference on Communications (ICC 2008)*, 2008. Available from http://echarts.org. 9

[5] Venkita Subramonian. KitCAT User Manual. Technical report, AT&T, 2008. Available from: http://echarts.org. 14


# A Container Configuration

The following section explains how to configure a SIP container to use E4SS. We include instructions for SailFin and OCCAS. This instructions include information on what JVM options need to be set for the container, how to deploy the DFC application router (included with the E4SS development kit), and how to configure logging.

### A.0.4 JVM Options Summary

The following JVM options can be set for a container. The only necessary option is the transitionTimerManager, the rest are optional.

```
org.echarts.system.transitionTimerManager.class = \
         org.echarts.servlet.sip.TransitionTimerManager
org.echarts.debugging = true              # ECharts debug info
org.echarts.servlet.sip.debugging=true    # E4SS debug info
org.echarts.machine.debugging.globalStateOutput=false # more ECharts debug info
org.echarts.servlet.sip.messagelog=true   # E4SS log (equivalent) SIP messages
java.util.logging.config.file=mylogging.properties # E4SS logging config file
org.echarts.servlet.sip.logdir=e4sslogs   # E4SS log files directory
```

---

For more information on ECharts debugging options, see the ECharts user manual [2].

## A.1 Configuring SailFin

The following configuration instructions have been tested with E4SS on SailFin build 60g.

### A.1.1 Container Configuration

First change your working directory to your SailFin home directory and start SailFin:

```
cd [SAILFIN_HOME]
bin/asadmin start-domain domain1
```

Then you can use SailFin's administration console, edit the file `[SAILFIN_HOME]/domains/domain1/config/domain.xml`, or use the `bin/asadmin` command to set container JVM options. Here's how to use the command line (where `\` indicates the command is continued on the next line):

```
bin/asadmin create-jvm-options \
'-Dorg.echarts.system.transitionTimerManager.class=\
org.echarts.servlet.sip.TransitionTimerManager:\
-Dorg.echarts.debugging=true:\
-Dorg.echarts.servlet.sip.debugging=true:\
-Dorg.echarts.machine.debugging.globalStateOutput=false:\
-Dorg.echarts.servlet.sip.messagelog=true:\
-Dorg.echarts.servlet.sip.logdir=${com.sun.aas.instanceRoot}/e4sslogs:\
-Djava.util.logging.config.file=logging.properties'
```

Note that ${com.sun.aas.instanceRoot} resolves to `[SAILFIN_HOME]/domains/domain1`.

Create the directory referred to by `org.echarts.servlet.sip.logdir`:

```
mkdir domains/domain1/e4sslogs
```

Create the logging properties file referred to by `java.util.logging.config.file` (for more information see Section A.3):

```
cat > domains/domain1/config/logging.properties << EOF
java.util.logging.FileHandler.limit = 500000
java.util.logging.FileHandler.count = 100
java.util.logging.FileHandler.append = false
EOF
```

Lastly, re-start SailFin for all the changes to take effect:

```
bin/asadmin stop-domain domain1
bin/asadmin start-domain domain1
```

### A.1.2 Application Deployment

To deploy an (E4SS) application on SailFin, use:

```
bin/asadmin deploy [pathToWarFile]/myapp.war
```

To list currently deployed applications, use:

```
bin/asadmin list *module*
```

To undeploy an application, use:

```
bin/asadmin deploy myapp
```

### A.1.3 Application Router Deployment

To deploy the DFC application router included with the E4SS development kit (see Section 4), use:

```
bin/asadmin deploy [EDK_HOME]/lib/dfcar.jar
```

The DFC application router requires its own configuration file `approuter.xml`. Once you have edited this file to suit your needs, it should be placed in the `[SAILFIN_HOME]/domains/domain1/config` directory. This file will be re-read by the DFC approuter whenever there is an application deployment change (deploy or undeploy).

## A.2 Configuring OCCAS

The following configuration instructions have been tested with E4SS on Oracle Communications Converged Application Server (OCCAS) 4.0. In the following we'll denote the root directory of an OCCAS domain as `[OCCAS_DOMAIN]`.

### A.2.1 Container Configuration

To set the OCCAS JVM container options you can edit the file `[OCCAS_DOMAIN]/bin/setDomainEnv.sh` so that the script variable `JAVA_OPTIONS` includes the following (where \ indicates the command is continued on the next line):

```
-Dorg.echarts.system.transitionTimerManager.class=\
org.echarts.servlet.sip.TransitionTimerManager:\
-Dorg.echarts.debugging=true:\
-Dorg.echarts.servlet.sip.debugging=true:\
-Dorg.echarts.machine.debugging.globalStateOutput=false:\
-Dorg.echarts.servlet.sip.messagelog=true:\
-Dorg.echarts.servlet.sip.logdir=$com.sun.aas.instanceRoot/e4sslogs:\
-Djava.util.logging.config.file=logging.properties
```

Create the directory referred to by `org.echarts.servlet.sip.logdir`:

```
mkdir [OCCAS_DOMAIN]/e4sslogs
```

Create the logging properties file referred to by `java.util.logging.config.file` (for more information see Section A.3):

```
cat > [OCCAS_DOMAIN]/logging.properties << EOF
java.util.logging.FileHandler.limit = 500000
java.util.logging.FileHandler.count = 100
java.util.logging.FileHandler.append = false
EOF
```

### A.2.2 Disable Persistence

BEA WLSS 3.1 MP1 introduced a new change, and this change is also in OCCAS 4.0. From the release notes:

> CR367390: This release improves server performance on Intel Quad Core processors when running in a non-replicated configuration. The code was modified to enable the database persistence flag by default, which in turn enables local serialization of call state data. This setting is ignored for replicated configurations.

This means that by default, call state data is serialized and stored to database. Because E4SS stores non-serializable objects in the SipApplicationSession (e.g. OpaqueFeatureBox), this default behavior causes a runtime exception.

To disable this behavior so E4SS applications can run on OCCAS 4.0, `[OCCAS_DOMAIN]/config/custom/sipserver.xml` needs to be modified as follows:

```
<persistence>
  <default-handling>none</default-handling>
  <db-enabled>false</db-enabled>
  <geo-enabled>false</geo-enabled>
</persistence>
```

Alternatively, this can be configured on the web administrative interface.

### A.2.3 Application Router Deployment

To deploy the DFC application router included with the E4SS development kit (see Section 4), first create an `approuter` directory and copy the application router jar file to the directory:

```
mkdir [OCCAS_DOMAIN]/approuter
cp [EDK_HOME]/lib/dfcar.jar [OCCAS_DOMAIN]/approuter
```

Then, from the OCCAS administrative interface, go to:

```
- Sip Server
  - Configuration pane
    - Application Router pane
      - check 'Use custom AR'
      - enter dfcar.jar for 'Custom AR filename'
```

To confirm that the preceding configuration step was successful, check that the file `[OCCAS_DOMAIN]/config/custom/sipserver.xml` has the following lines:

```
<app-router>
  <use-custom-app-router>true</use-custom-app-router>
  <app-router-config-data></app-router-config-data>
  <custom-app-router-jar-file-name>
      dfcar.jar
  </custom-app-router-jar-file-name>
  <default-application-name></default-application-name>
</app-router>
```

The DFC application router requires its own configuration file `approuter.xml`. Once you have edited this file to suit your needs, it should be placed in the `OCCAS_DOMAIN` directory. This file will be re-read by the DFC approuter whenever there is an application deployment change (deploy or undeploy).

## A.3 Logging

The default E4SS logger utilizes the `java.util.logging` package. Monitor logfiles will be created in the specified location. It is advisable to set up a custom set of logging properties in order to get the desired behavior for rotation of monitor logfiles. All messages are logged at the `INFO` level. The following settings would specify that logfiles be rotated after 500,000 bytes, that 100 files be preserved, and that new files should be created upon application initialization:

---

```
java.util.logging.FileHandler.limit = 500000
java.util.logging.FileHandler.count = 100
java.util.logging.FileHandler.append = false
```

---

For more information about ECharts monitoring and logging see the ECharts User Manual [2].