ClickToDial with ECharts for SIP Servlets: A Case Study

Gregory W. Bond
AT&T Labs – Research

February 6, 2008

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

Abstract

This document presents a case study using the ECharts for SIP Servlets framework to develop a click-to-dial IP telecommunications application. The click-to-dial application is a converged one, that is, it possesses both a SIP component for managing phone calls and an HTTP component for managing a web-based user interface. The goal of this study is to familiarize developers with the benefits of using the open source ECharts for SIP servlets framework; a framework to develop SIP servlets. The approach taken is to “hijack” a traditional implementation of the click-to-dial application and then replace its SIP servlet with one developed using ECharts for SIP Servlets. A comparison of the original and revised examples reveals how simple it is to develop re-usable, composeable, maintainable SIP servlets using ECharts for SIP Servlets. The document includes instructions for building, deploying and running the click-to-dial application.

Contents

1 Introduction ................................................. 2

2 An overview of the behavior and structure of the original example ................................................. 3

3 A brief overview of the revised example .................. 3
1 Introduction

Simple examples are a great way to familiarize yourself with the ins and outs of an otherwise unfamiliar environment. To this end, a simple click-to-dial application program was released on the GlassFish wiki [6] with the (presumed) goals of introducing programmers to (1) the open source SailFin application server [21], (2) SIP servlet programming, (3) combining HTTP servlets and SIP servlets into a single, converged application, and (4) developing converged applications with the NetBeans IDE [18]. Click-to-dial is a good application for this purpose. It has a SIP component to manage phone calls, a web component providing the user interface, and a database component for accessing user information.

The goal here is narrower in scope: to familiarize developers with the benefits of using the open source ECharts for SIP servlets framework [13] to develop SIP servlets. To achieve this goal we will "hijack" the original click-to-dial example and replace its SIP servlet with a servlet developed using the ECharts for SIP servlets framework. Then we will compare the revised example with the original showing how simple it is to develop re-usable, composeable, maintainable SIP servlets using ECharts for SIP Servlets. These benefits extend beyond the click-to-dial example to any other non-trivial converged application. Familiarity with the original click-to-dial example is not required since we’ll go over it in this discussion. However, familiarity with SIP servlets and HTTP servlets will be helpful for following the discussion.

This discussion also includes instructions for building, deploying and running the revised example on a SailFin container. The revised example currently only runs
on SailFin since the original example relies on Java Persistence API (JPA) support provided by EJB 3 [16]. JPA/EJB 3 are not (directly) supported by BEA’s WLSS container v3.1. While IBM claims that the IBM WebSphere application server v6.1 supports EJB 3, ECharts for SIP Servlets is not supported on that container yet.

If all you are interested in is building, deploying and running the revised example then you should proceed directly to Section 12, skipping the overview, benefits and comparison sections.

2 An overview of the behavior and structure of the original example

In general, the click-to-dial application works as follows: a user (call him Bob) clicks a button on a web page indicating who the user wants to call (call her Alice). The application reacts by first calling Bob at some predefined number. After Bob answers his phone, the application calls Alice at some predefined number. When Alice answers, Bob and Alice are able to talk to one another. To provide this behavior the SailFin click-to-dial application includes:

1. an HTTP servlet and two JSPs to handle the web interface and to make the initial call e.g. to Bob
2. two SIP servlets: one to receive SIP registration requests from user’s phones (so a user’s name can be associated with their phone number), and another to receive the response to the initial call (e.g. to Bob) and to handle the second call (e.g. to Alice), and
3. a JPA data model for storing and retrieving the registration info.

The original click-to-dial example is structured as a NetBeans project although, as we will explain later, it is not necessary to use NetBeans to build, deploy or run the example. The nbproject directory contains the configuration files for the build. These are normally maintained by NetBeans but can be manually edited. The lib directory contains the file ssa-api.jar, which contains the SIP servlet API class files. This jar is necessary for building the example. The top-level web directory contains two JSP files and a JavaScript library file used by the application’s web interface. The WEB-INF directory contains the two deployment descriptor files: sip.xml and web.xml. The src/conf directory contains the JPA descriptor file: persistence.xml. The src/java/clicktodial web, sip and model subdirectories contain the HTTP servlet, SIP servlet and JPA model class definitions, respectively.

3 A brief overview of the revised example

The revised version of the SailFin click-to-dial example is distributed as part of the latest ECharts for SIP Servlets development kit. It can be found in EDK_HOME/examples/ClickToDial. The significant change relative to the original version is that the application’s sip.xml specifies that the application use a different SIP servlet to handle the two outbound calls. This means the SIP servlet that originally handled this task, CallSipServlet, is ignored in the revised example. The registration SIP
The servlet remains unchanged. A few other minor changes were made as well. The PlaceCallServlet HTTP servlet was modified to initiate calls via the new SIP servlet – this is discussed in more detail shortly. The file lib/ssa-api.jar was removed since it is provided as part of the SailFin distribution. Some changes were made to the build configuration files in the nbproject directory – more on that later. Lastly, persistence.xml has been modified to refer to a different container data source. This change allows the example to be run without NetBeans.

The allocation of responsibilities between the HTTP and SIP servlets has changed a bit in the revised version. In the original version, the HTTP servlet, PlaceCallServlet, initiated the call to the first party (e.g. Bob), and the SIP servlet, CallSipServlet, was responsible for receiving the first party’s response and handling the call to the second party (e.g. to Alice) in its entirety. In the revised example, the SIP servlet that replaces CallSipServlet handles both calls in their entirety i.e. it also initiates the call to the first party. This means that the HTTP servlet, PlaceCallServlet, no longer initiates the call to the first party directly. Instead it initiates the call indirectly by calling a method on the replacement SIP servlet. We justify this re-allocation of behavior by appealing to the tenets of re-useability and separation of concerns – the topic of the next section.

4 Re-usable telecom services

Consider the task of developing an IP-based system that will offer a number of telecom services to the user. AT&T’s CallVantage service is a good example [2]. This service offers a number of standard features such as Call Waiting and Call Forwarding, and a number of advanced features such as Do Not Disturb and Locate Me. When embarking on a multi-service development project like this the system architect is faced with a decision to either: (1) combine the logic for all the services into a single monolithic program or (2) develop the services as independent (hopefully reusable) modules and then utilize an overarching framework to compose them. All other things being equal, the second alternative is clearly preferable for all the reasons that modular and re-usable components are standard practice for software development today. For this reason, our research group here at AT&T has been inventing ways to develop and compose modular, re-usable IP-based telecom services. One result of our work, application routing, is a significant aspect of the next SIP Servlet specification, specified by the JSR 289 working group [1]. Application routing provides a framework for composing modular telecom services. Another result of our work is ECharts for SIP Servlets [23], a framework and language for developing modular, reusable SIP servlet-based telecom services. The ideas underlying these two results have been put to the test many times, notably to support AT&T’s CallVantage service which was beta-tested in 2003 and rolled out as a product in 2004 [3].

5 Applications, features and fragments

We view the universe of re-usable IP telecom services as being divided into three parts: applications, features and fragments.

An application is a stand-alone service, deployable as a war file. It has a telecom component (e.g. to initiate, receive or forward calls) and logic that isn’t directly
related to telecom (e.g. business logic or a web interface). The SailFin click-to-dial example is a good example of a converged application: it has logic for registering phones and handling calls (the telecom component), logic to update and access the data model and logic that provides a web-based user interface.

A feature is a complete, general-purpose telecom component that is designed to be utilized by a wide variety of applications. A feature is complete in the sense that it satisfies all of an application’s telecom requirements. A feature usually defines interfaces for exchanging information between itself and the application logic that exists outside itself. The CallSipServlet in the original SailFin example is not a good example of a feature since it does not handle all telecom operations. In particular, it does not initiate the call to the first party. In contrast, the SIP servlet that replaces the CallSipServlet in the revised SailFin example is a good example of a feature. This feature handles all call-related tasks (including initiating the call to the first party) and it provides interface methods for initiating the calls and notifying interested parties of its progress. We will look at this feature in more detail shortly.

A fragment is a simple, re-usable telecom component designed to be utilized by a wide variety of features. A collection of fragments can be thought of as a library that supports commonly-performed telecom operations e.g. receiving a call, initiating a call, switching a call. As we will see, the SIP servlet that replaces the CallSipServlet in the revised SailFin example utilizes a number of fragments.

In the latest ECharts for SIP Servlets development kit, the directory EDK_HOME/examples contains applications (including the revised click-to-dial application), the directory EDK_HOME/features contains features (including the feature used by the revised click-to-dial application) and the directory EDK_HOME/src/org/echarts/servlet/sip/machines contains fragments (including fragments used by the click-to-dial feature). To browse the available fragments, see the ECharts for SIP Servlets DK API Javadoc included with DK or online [11].

6 The click2DialFlow1 feature

Instead of using the CallSipServlet, the revised click-to-dial application uses the click2DialFlow1 feature included with the latest ECharts for SIP Servlets development kit. You may be curious why this feature’s name is not simply ‘click2Dial’. This is because there are a number of ways for a third party to establish calls between two parties. These are enumerated in IETF RFC 3725 “Third Party Call Control” [20]. The approach taken by the original and revised click-to-dial examples corresponds to “Flow 1.”

The click2DialFlow1 feature is located in EDK_HOME/features/click2DialFlow1. This directory and its subdirectories were generated by the ECharts for SIP Servlets appgen tool. Using the appgen tool is a convenient way to set-up a new ECharts for SIP Servlets feature or a new application. For more information, see the ECharts for SIP Servlets screencast [12] and the ECharts for SIP Servlets manual [24]. All of the click2DialFlow1 source files are located in click2DialFlow1/src/org/echarts/servlet/sip/features/click2DialFlow1.

Of primary interest is the ECharts machine source file Click2DialFlowMachine.ech. This machine defines the behavior that was formerly provided by the CallSipServlet in the original example. We’ll examine the machine’s logic in more
detail in a moment.

The feature also includes two interface classes supporting communication between the feature and any application it may be embedded within. One class, `JavaToClick2DialFlow1Machine`, defines a method, `initiate()`, for initiating calls to the two parties. The revised version of the `PlaceCallServlet` HTTP servlet no longer directly initiates the call to the first party. Instead it calls `JavaToClickToDialFlow1Machine.initiate()` to indirectly initiate the call via the click2DialFlow1 feature. The other interface class, `ClickToDialFlow1MachineToJava`, defines methods called by the feature to notify an application of its progress i.e. when both parties are connected and when both parties are disconnected. This default behavior of this class is to simply log the progress events with the feature’s ECharts log. The feature also includes two JSPs, `SarContent/call.jsp` and `SarContent/drop.jsp`, for unit testing the feature.

7 Integrating the feature with the application

As explained above, the significant revision to the original example was to modify `sip.xml` so that the application utilizes a different SIP servlet to handle the calls to the two parties. Here’s the relevant part of the modified `sip.xml`:

```xml
<servlet>
  <servlet-name>click2DialFlow1</servlet-name>
  <servlet-class>org.echarts.servlet.sip.EChartsSipServlet</servlet-class>
  <init-param>
    <param-name>machineClassName</param-name>
    <param-value>org.echarts.servlet.sip.features.click2DialFlow1.Click2DialFlow1Machine</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

Instead of the servlet-class `clicktodial.sip.CallSipServlet`, `sip.xml` now refers to `org.echarts.servlet.sip.EChartsSipServlet`. `EChartsSipServlet` is a wrapper servlet used by ECharts for SIP Servlet features. The actual servlet behavior is specified by an ECharts for SIP Servlets machine. In this case, `sip.xml` specifies the servlet’s `machineClassName` to be `org.echarts.servlet.sip.features.click2DialFlow1.Click2DialFlow1Machine`. A few other changes needed to be made to `sip.xml` in order to support the ECharts for SIP Servlets framework. See the ECharts for SIP Servlets screencast and manual for more information [12, 24]. As explained in detail below, building the revised click-to-dial application is as simple as including the click2DialFlow1 jar file in the application’s war file.

8 The Click2DialFlow1Machine

Utilities included with the ECharts for SIP Servlets framework integrate graphical depictions of ECharts machines with the generated Javadoc documentation. Figure 1 shows the Javadoc page for the `Click2DialFlow1Machine`. Clicking on the `machine` button reveals an interactive graphical depiction of the `Click2DialFlow1Machine` shown in Figure 2.
Figure 1: Javadoc Page for the Click2DialFlow1Machine
As you can see, the Click2DialFlow1Machine is a simple machine. It defines only four transitions and three states. Where the machine gets its power is from the re-useable machine fragment, ThirdPartyCCFlow1FSM, that is embedded in its C2D state. This embedded machine, in turn, calls upon two other fragments, CallFSM and TransparentFSM, and TransparentFSM uses the fragment TransparentHandleRequestFSM. All of these fragments are located in EDK_HOME/src/org/echarts/servlet/sip/machines. We’ll take a closer look at the fragments in a moment.

The “sunny day” scenario for the Click2DialFlow1Machine is that it first waits in its INIT state for a request to initiate calls. This request will arrive in the form of an object sent to the machine as a result of a call to the JavaToClickToDialFlow1Machine.initiate() static method. Then the machine transitions to its C2D state which causes the embedded ThirdPartyCCFlow1FSM to begin executing. The ThirdPartyCCFlow1FSM will initiate a call to the first party and then, once the first party responds, initiate a call to the second party. When the second party is connected, the ThirdPartyCCFlow1FSM completes the connection with the first party and transitions to its TRANSPARENT state. This causes the second transition in the parent Click2DialFlow1Machine to fire, resulting in a 'connected' notification being logged, indicating that the two parties are now connected. When either party hangs up, the ThirdPartyCCFlow1FSM disconnects the other party and then transitions to its END state. This causes the parent Click2DialFlow1Machine to log a 'disconnected' notification and transition to its END state via its third transition. Whereas the first three transitions support setting up calls, the fourth transition supports asynchronously dropping both calls regardless of their respective call states. Calling the JavaToClickToDialFlow1Machine.dropCalls() instance method triggers this behavior. This method is not used in the revised click-to-dial example.

How about “rainy day” scenarios? For example, what if the first party doesn’t answer? What if the first party hangs up before the second party answers? What if
the second party is busy? What if the second party provides an early media response? These scenarios are taken care of by the embedded fragments or by the ECharts for SIP Servlets framework.

It is important to realize that the developer of the click2DialFlow1 feature did not need to understand the detailed behavior of the underlying ThirdPartyCCFlow1 fragment. It was enough to know which of its states represented when parties were connected and disconnected. All the feature developer needed to do was was write the Click2DialFlow1Machine and its two interface classes – a straightforward task.

Because the ThirdPartyCCFlow1 fragment is re-useable it can be used in other features such as Mid-Call Announcement, described in [20]. Since “Flow1” is intended to be used when the second party is guaranteed to answer immediately, the developer may choose to use “Flow4” instead [20]. In this case, the developer need only replace ThirdPartyCCFlow1 with the ThirdPartyCCFlow4 fragment in the Click2DialFlow1Machine. Incidentally, the ThirdPartyCCFlow4 fragment is distributed with the latest ECharts for SIP Servlets development kit in the EDK_HOME/src/org/echarts/servlet/sip/machines directory. The fragments distributed with the ECharts for SIP Servlets DK can be browsed by consulting the API Javadocs [11].

9 The ThirdPartyCCFlow1FSM

Although the developer of the click2DialFlow1 feature didn’t need to understand the detailed behavior of the ThirdPartyCCFlow1FSM, we’ll take a quick look at it here to highlight some of the features of the ECharts for SIP Servlets framework. Related sources of information are the ECharts for SIP Servlets manual [24] and our IPTComm 2007 paper [23].

A graphical depiction of the machine, generated as part of the ECharts for SIP Servlets API Javadocs [11], is shown in Figure 3. Figure 4 shows a screen-snapshot taken while interacting with the image using an SVG-aware browser [15]. By holding the cursor over a transition, it is highlighted (in orange) and a tooltip displays the Javadoc comment associated with the transition.

Here’s an overview of the sunny day scenario. The CallFSM fragment is used in the CALL_FIRST_PARTY state to initiate a connection with the first party. When the first party answers, the ThirdPartyCCFlow1FSM machine transitions to the CALL_SECOND_PARTY state. This state contains two concurrently running machines (indicated by dashed lines): the CALLER machine and the CALLEE machine. While the CALLEE machine initiates a call to the second party (via another CallFSM instance), the CALLER machine handles any messages received from the first party. When the call to the second party succeeds, the call to the first party is completed (shown by the highlighted transition in Figure 4). At this point, both the CALLER and the CALLEE machines are in their CONNECTED states. This causes a transition to fire in the parent ThirdPartyCCFlow1FSM that changes the machine state to TRANSPARENT. This state uses the TransparentFSM fragment to propagate any messages received from one party to the other party. When one party hangs up, the appropriate messages are forwarded between the two parties until both parties are disconnected from one another and the TransparentFSM enters a terminal state. This causes the ThirdPartyCCFlowFSM to transition to its END state.

In ECharts for SIP Servlets, SIP messages are sent and received via ECharts ports.
Figure 3: The ThirdPartyCCFlow1FSM
Figure 4: Interacting with ThirdPartyCCFlow1FSM Diagram
In the ThirdPartyCCFlow1FSM, two ports are defined: one for exchanging messages with the first party and one for exchanging messages with the second party. So to send a message to, say, the first party, a machine sends the message via the port handling first party messages. Conversely, when the first party sends a message, the ECharts for SIP Servlets framework conveys the message to the first party port for processing by a machine.

Now let’s look at one simple rainy day scenario since it highlights the elegant way that ECharts for SIP Servlets supports timed events. Consider the scenario where the first party doesn’t answer (perhaps because the wrong phone number is registered for the first party). To handle this, there is a ‘delay’ transition from the CALL_FIRST_PARTY state to the END state. If the first party doesn’t answer after a time out interval specified in the ThirdPartyCCFlow1FSM constructor, then the delay transition fires causing the machine to enter its (terminal) END state.

10 Benefits of the ECharts for SIP Servlets Framework

The discussion so far has revealed a number of the benefits of using the ECharts for SIP Servlets framework:

- SIP servlet logic is defined in terms of state machines;
- there is support for automatic call teardown, regardless of call state;
- applications are developed with re-useable features and fragments;
- customized behavior is possible by overriding submachine transitions;
- there is support for communication between a SIP servlet and its application environment;
- there is support for timed events;
- the generated Javadoc documentation includes interactive graphical depictions of ECharts machines.

A comprehensive overview of the benefits of using the ECharts for SIP Servlets Framework can be found in our IPTComm 2007 paper [23].

11 A comparison of the original and revised examples

Now for a comparison of the SIP servlets used in the original and revised examples. The intention here is point out the high level differences that exist between any “traditional” SIP servlet and an ECharts-based SIP servlet. Nevertheless, a familiarity with the original servlet will be helpful to appreciate some of these differences.

Original: The first party and second party logic are combined together in the servlet’s doRequest and doResponse methods. The programmer had to use conditional statements in these methods to explicitly query the values of stored attributes (SecondPartyAddress, LinkedSession) to determine whether a message is associated with the first party or the second party. The programmer also had to set the
value of these attributes in the first place. In general this approach to handling multiple SIP sessions will be harder to manage and maintain as more sessions are managed by a servlet.

Revised: The first party and second party logic is separated by virtue of being defined in terms of state machines. Furthermore, there is no need for the programmer to manage special attributes to associate messages with SIP sessions and to associate sessions with one another – the ECharts for SIP Servlet framework automatically takes care of this for the programmer.

Original: The servlet code can’t readily be re-used by another application. Naturally, this is because the code wasn’t designed to be re-used. On the contrary, it was designed to highlight SailFin features such as JSR 289-style resource injection. Used in this fashion, resource injection effectively tightens the coupling between the SIP and HTTP servlets thereby limiting re-use.

Revised: Not only are the fragments utilized by the example re-useable (ThirdPartyCCFlow1FSM and its submachines), but the feature code itself is re-usable (Click2DialFlow1Machine).

Original: The servlet captures only a subset of the complete third party call control “Flow1” behavior [20]. No rainy day scenarios are handled. It is interesting to consider why that is the case. The obvious reason is that the servlet is only meant to be an example and that adding the logic to handle rainy day scenarios would make it overly complicated.

Revised: The servlet handles rainy day scenarios yet, we would claim, it is still not so complex that it can’t be used as an example. Much of the complexity associated with the rainy day scenarios is either handled by low-level re-usable fragments or by the ECharts for SIP Servlets framework. This means that the sunny day logic is most prominent. It also means that the sunny day logic is what the developer of the ThirdPartyCCFlow1 fragment was able to focus on, knowing that the fragments and ECharts for SIP Servlets framework would largely take care of the rainy day scenarios.

12 How to build and deploy the revised example

This section describes how to build and deploy the revised example from the command line or using the NetBeans 6.0 IDE. The next section explains how to run the example.

You will first need to install the latest ECharts for SIP Servlets DK [8]. Simply unzip the downloaded file to install it. Due to unfortunate license restrictions [7] you also need to download and install a third-party jar file required for the example: the NIST implementation of the JSR 141 SDP API. Download sip-sdp.jar from the NIST JAIN-SIP nightly builds page [19] and move it to EDK_HOME/lib.

As we stated in the introduction, both the original and revised examples require SailFin because of their reliance on JPA. Next we need to set up SailFin (milestone 2 or greater) to run ECharts for SIP Servlets applications. We’ll only cover the settings required to run this example. For additional settings to support, for example, application routing, see [14]. First it is necessary to add some JVM options. In the SailFin home directory enter the following command with SailFin running:

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

Next create a file to control how the Java logging utility will manage ECharts for SIP Servlets logging. Create the file SAILFIN_HOME/domains/domain1/config/logging.properties with the following contents:

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

Now create a directory where ECharts for SIP Servlets log files will be created and updated:

```
mkdir SAILFIN_HOME/domains/domain1/echlogs
```

Once you restart SailFin, it is ready to run ECharts for SIP Servlets applications!

There are two approaches to building and deploying the revised example: from the command line or using the NetBeans IDE.

### 12.1 Command Line

The next thing to do is build the click2DialFlow1 feature that is used by the click-to-dial application.

Edit the file EDK_HOME/features/click2DialFlow1/env.properties so the ext.classpath property points to the jar files for the SIP and HTTP servlet API’s. These jars are included with the SailFin distribution. Just substitute /Users/greg/sailfin with the path to your SailFin home directory in the following example:

```
ext.classpath = /Users/greg/sailfin/lib/ssa-api.jar:/Users/greg/sailfin/lib/javaee.jar
```

From the directory EDK_HOME/features/click2DialFlow1 invoke ant to build click2DialFlow1.jar:

```
ant jar
```

If you would like to generate the feature’s Javadoc (including the interactive graphical machine depiction), use:

```
ant javadoc
```

The generated Javadoc is located in EDK_HOME/features/click2DialFlow1/doc/api.

Now we shift our attention to the revised click-to-dial application itself. First it’s necessary to edit EDK_HOME/examples/ClickToDial/nbproject/private/private.properties in order that the build classpath reference your local SailFin jar files. Substitute all occurrences of /Users/greg/sailfin in this file with the path to your SailFin installation. Also substitute all occurrences of /Users/greg/Projects/src/EChartsSipServlet_DK_2.3-beta with the path to your EDK.

Next, build the click-to-dial application war file ClickToDial.war.

```
ant
```
The resulting war file, located in \texttt{EDK\_HOME/examples/ClickToDial/dist}, includes the \texttt{click2DialFlow1.jar} file we built earlier as well as the supporting jar files from the ECharts for SIP Servlets DK.

To deploy the click-to-dial application first ensure that both SailFin and SailFin’s database are running. Then deploy the application. From the SailFin home directory:

\texttt{bin/asadmin deploy EDK\_HOME/examples/ClickToDial/dist/ClickToDial.war}

### 12.2 NetBeans

If you haven’t already, you must first install the latest version of the NetBeans SIP Application Development module and configure NetBeans to use SailFin as a server [17].

The first task is to build the \texttt{click2DialFlow1} feature that is used by the click-to-dial application. To do this open the file \texttt{EDK\_HOME/features/click2DialFlow1/\_\_\_env.properties} and update the \texttt{ext.classpath} paths to reflect the location of your SailFin installation. After you save these changes, use NetBeans to open the file \texttt{EDK\_HOME/features/click2DialFlow1/build.xml} and right-click on the “jar” target that appears in the lower left hand panel and select “Run Target” from the contextual menu. This will build the \texttt{click2DialFlow1.jar} file required by the click-to-dial application. If you would like to generate the feature’s Javadoc (including the interactive graphical machine depiction), run the “javadoc” target. The generated Javadoc is located in \texttt{EDK\_HOME/features/click2DialFlow1/doc/api}.

The next task is to build the click-to-dial application war file. In NetBeans, open the NetBeans project \texttt{EDK\_HOME/examples/click2Dial}. You most likely will see an error message reporting unresolved references. This is because the project properties reference two jar files and a library that haven’t been defined for your installation yet. To fix these problems, right-click on the ClickToDial project icon and choose “Resolve Reference Problems...” from the contextual menu. A window like the one shown in Figure 5 should appear. Select the \texttt{click2DialFlow1.jar} entry in this window and click on the “Resolve” button. Then browse to \texttt{EDK\_HOME/features/click2DialFlow1/click2DialFlow1.jar} and then click “OK”. This will resolve the first reference problem. Then select the \texttt{ssa-api.jar} entry and click the “Resolve” button. Browse to \texttt{SAILFIN\_HOME/lib/ssa-api.jar} and then click “OK”. This will resolve the second reference problem. Finally select the \texttt{EChartsSipServlet} entry and click on the “Resolve” button. Then click on the “New Library...” button to create the missing library. The library name should be “EChartsSipServlet” and it should be of type “Class Libraries”. Then add the following jar files to the library: \texttt{EDK\_HOME/lib/approuter.jar}, \texttt{EDK\_HOME/lib/echarts.jar}, \texttt{EDK\_HOME/lib/echarts-sipservlet.jar}, and \texttt{EDK\_HOME/lib/sip-sdp.jar}. After you finish creating this library your last reference problem should be resolved. To build the war file right-click on the ClickToDial project icon and choose “Build” from the contextual menu.

Finally, deploy the application to SailFin by clicking on the ClickToDial project icon and choosing “Undeploy and Deploy” from the contextual menu. This will start up SailFin and the SailFin database if they weren’t already running and then deploy the application.
13 How to run the revised example

In this section we’ll describe how to run the example using SIPp (version 2 or greater) clients [22] and we’ll also show an example of the useful logger output generated by the ECharts for SIP Servlets framework. Another way to run the example, described on the GlassFish wiki [6], uses X-Lite clients instead of SIPp clients.

To confirm the application was successfully deployed, bring up http://localhost:8080/ClickToDial in a browser window. You should see the application login screen. The application requires that you log in as both Bob and Alice prior to registering phone numbers for them so be sure to do this before proceeding to the next step.

We’ve added the directory EDK_HOME/examples/ClickToDial/test to the original directory structure. In this directory you will find four SIPp scripts. Two of them, sipp-register-alice.xml and sipp-register-bob.xml are used to register Alice and Bob’s phones, respectively. The other two, sipp-uas-alice.xml and sipp-uas-bob.xml play the role of the Alice and Bob’s respective phones. Alice’s phone simply answers when called and hangs up after Bob hangs up. Bob’s phone also answers when called but it hangs up after some specified time interval. In our test, Bob will play the role of the first party (the click-to-dial initiator) and Alice will play the role of the second party. Bob will also initiate disconnecting the call with Alice.

We’ll assume you’re running SIPp on the same machine that SailFin is running on. First run Alice and Bob’s registration scripts by entering the following commands. These commands should complete quickly.

```
sipp 127.0.0.1:5060 -sf sipp-register-alice.xml -m 1 -i 127.0.0.1 -p 21060
sipp 127.0.0.1:5060 -sf sipp-register-bob.xml -m 1 -i 127.0.0.1 -p 22060
```

Now refresh the web interface. If you see SIP addresses associated with Bob and Alice, then the registration succeeded.

The next step starts up Alice and Bob’s phones (user agent servers in SIP parlance). Since SIPp outputs text when running a script and since these are long-lived scripts you’ll want to dedicate a separate window to run each command.
Now both Alice and Bob’s phones are active and waiting for a call. To initiate a call, login as Bob at the web interface and click on the link to call Alice. In Bob’s SIPp window you should see the SIP INVITE request arriving and the 180 and 200 responses being sent. Next, in Alice’s window you should see the INVITE arriving, the 180 and 200 being sent and the ACK being received. Back in Bob’s window you should see an ACK being received and then Bob’s script should pause. The pause interval is dictated by the -d parameter value specified when running Bob’s SIPp script. We’ve specified 10,000 milliseconds here. After pausing for 10 seconds, Bob’s script will send a BYE which will be forwarded to Alice. Alice will respond by sending a 200 which will be forwarded to Bob. At this point both scripts should terminate.

To see how the click2DialFlow1 feature behaves during this test you can examine the ECharts log for the Click2DialFlow1Machine generated in SAILFIN_HOME/domains/domain1/echlogs/monitor-click2DialFlow1.log.0. This log shows when each machine transition fired along with the message that caused it to fire. The log also shows SIP message input and output events including message headers and message contents. For example, consider the following log excerpt:

```plaintext
# initial log entry
time: 2008.01.09 21:39:42.945 EST (1199993272945)
event: MessageTransitionEvent
root: 55d80f0b:117618d5758::7fd0
machine: 55d80f0b:117618d5758::7fcd
state path: :(FeatureBoxTerminationHandlerFSM).RUN_APPLICATION:(Click2DialFlow1Machine)
sequence: 55d80f0b:117618d5758::7fcf
transition: [INIT] - BoxPort ? Click2DialFlow1Bean (party1 = sip:Bob@127.0.0.1:22060
party2 = sip:Alice@127.0.0.1:21060
timeoutSec = -1
earlyMediaReinvite = true
uid = -f157694:117c35c2b06::7fa8
) -> [C2D]
local state:

... C2D:ThirdPartyCCFlow1FSM
CALL_FIRST_PARTY:CallFSM
INIT

...```

```plaintext
# later log entry
Equivalent SIP message:
INVITE sip:Bob@127.0.0.1:22060 SIP/2.0
Max-Forwards: 70
From: <sip:Alice@Lap-Bat-Pro.local>;tag=fb8ozwug-3
```
This log excerpt shows two events. The first event, a MessageTransitionEvent, corresponds to the first transition firing in the Click2DialFlow1Machine. This transition fires upon receipt of an object (bean) containing the parameters specified by the HTTP PlaceCallServlet when it calls the Java2Click2DialFlow1Machine.initiate() method. The transition changes the machine’s state from INIT to C2D. Recall that the C2D state contains the ThirdPartyCCFlow1FSM which, upon activation, sends an INVITE message to Bob. The details of this message are shown by the SipOutputEvent.

The logging support provided by the ECharts for SIP Servlets framework greatly simplifies debugging an application during development because logging information is presented in terms of ECharts programming abstractions such as SIP messages, transitions and ports.

14 Concluding Remarks

This case study provides concrete evidence of the benefits offered by the ECharts for SIP Servlets framework. We’ve focused on the benefits derived from modularizing the click-to-dial application and expressing its telecom logic in terms of state machines. Another benefit we did not consider here is that the ECharts for SIP Servlets framework also supports application composition [9, 4, 5, 1].

The original click-to-dial application is not a good example for explaining application composition. Although the registration servlet bundled with the application should really be bundled separately in its own application the issue of application composition doesn’t arise because the registration servlet is invoked for REGISTER messages and the click-to-dial servlet is invoked for INVITE messages. As such, we intend to make application composition the subject of a future case study.

Another issue we did not explore in any detail here is the support provided by the ECharts for SIP Servlet framework for interfacing between a SIP feature and its external application environment. This will also be the subject of a future document.

We encourage you to take a closer look at ECharts for SIP Servlets [10]. Once you start working with it we think you’ll agree that it’s the preferable approach to developing telecom applications.

15 Acknowledgments

We would like to thank Sun and Ericsson for releasing their click-to-dial application code and the SailFin container as open source under the Common Development and Distribution License.

References


