The SIP Servlet Tutorial

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 820–3007–10
January, 2009

090119@21808

Contents

Preface

This is *The SIP Servlet Tutorial*, a tutorial that describes how to develop telecommunications applications that use the session initialization protocol (SIP) on the Java EE platform. This tutorial also covers how you can integrate SIP applications with other Java EE technologies, like web applications and enterprise beans. Here we cover all the things you need to know to make the best use of this tutorial.

Who Should Use This Book

This tutorial is intended for programmers who are interested in developing and deploying SIP applications on the Sun Java System Communications Application Server 1.5, a Java EE server that integrates a SIP servlet container. Communications Application Server 1.5 is based on the open-source GlassFish and SailFin projects.

This tutorial is intended for the following readers:

- Java programming language developers interested in learning about how to create SIP applications.
- SIP application developers who are new to server-side Java programming language development.
- Anyone interested in how SIP applications work, and how they can be integrated in with traditional web applications and Java EE components.

This tutorial assumes you are conversant in reading Java programming language source code, and you have a basic understanding of client/server network applications.

About the Examples

This section tells you everything you need to know to install, build, and run the examples included in the tutorial bundle.

Required Software

The following software is required to run the examples.

Java Platform, Standard Edition

To build, deploy, and run the examples, you need a copy of Java Platform, Standard Edition 5.0 (Java SE 5.0) or higher. You can download the Java SE 5.0 software from `http://java.sun.com/javase/downloads/index_jdk5.jsp`. Download the current JDK update that does not include any other software (such as the NetBeans IDE or Java EE).

Communications Application Server 1.5

Communications Application Server 1.5 is targeted as the build and runtime environment for the tutorial examples. Communications Application Server 1.5 is based on the GlassFish and SailFin open-source projects.

NetBeans IDE

The NetBeans integrated development environment (IDE) is a free, open-source IDE for developing Java programming language applications, including enterprise applications. NetBeans IDE supports the Java EE 5 platform. You can build, package, deploy, and run the tutorial examples from within NetBeans IDE.

SIP Modules for NetBeans IDE

Integrate the SailFin plug-in modules, which add SIP application development functionality to NetBeans IDE. The modules are bundled with Communications Application Server 1.5.

1. In NetBeans IDE, select Tools→Plugins.
2. Click the Downloaded tab and click Add Plugins.
3. Navigate to the *Install*`/tools/netbeans` directory and select all the files in this directory.
4. Click Install, then Next.
5. Select I Agree in the License Agreement window and click Install.
6. Click Continue to install the unsigned modules, then click Finish.

Sample Applications

The tutorial uses several sample applications available on the SailFin website.

1. Go to the following URL:
   `http://wiki.glassfish.java.net/gfwiki/Wiki.jsp?page=SipExamples`.
2. Follow the instructions to download the `SipProxy` and `SIP Servlet 1.1 Click-To-Dial` sample applications.

SIPp

SIPp is an application to test SIP clients and servers. It is available from `http://sipp.sourceforge.net/`.

X-Lite Soft Phone

X-Lite is a free multi-platform soft phone used in the examples. It is available from `http://www.counterpath.com/x-lite.html[amp ]active=4`.

Apache Ant

Ant is a Java technology-based build tool developed by the Apache Software Foundation (`http://ant.apache.org`), and is used to build, package, and deploy the tutorial examples. Ant is included with the Communications Application Server 1.5. To use the ant command, add *JAVAEE_HOME*`/lib/ant/bin` to your PATH environment variable.

Building the Examples

The tutorial examples are distributed with a configuration file for either NetBeans IDE or Ant. Directions for building the examples are provided in each chapter. Either NetBeans IDE or Ant may be used to build, package, deploy, and run the examples.

▼ Building the Examples Using NetBeans IDE

To run the tutorial examples in NetBeans IDE, you must register your Communications Application Server 1.5 installation as a NetBeans Server Instance. Follow these instructions to register the Communications Application Server 1.5 in NetBeans IDE.

1. **Select Tools-->Server Manager to open the Server Manager dialog.**
2. **Click Add Server.**
3. **Under Server, select Sun Java System Application Server and click Next.**
4. **Under Platform Location, enter the location of your Application Server installation.**
5. **Select Register Local Default Domain and click Next.**
6. **Under Admin Username and Admin Password, enter the admin name and password created when you installed the Application Server.**
7. **Click Finish.**

Tutorial Example Directory Structure

To facilitate iterative development and keep application source separate from compiled files, the tutorial examples use the Java BluePrints application directory structure.

Each application module has the following structure:

- `build.xml`: Ant build file
- `src/java`: Java source files for the module
- `src/conf`: configuration files for the module, with the exception of web applications
- `web`: JSP and HTML pages, style sheets, tag files, and images
- `web/WEB-INF`: configuration files for web applications
- `nbproject`: NetBeans IDE project files

The Ant build files (`build.xml`) distributed with the examples contain targets to create a `build` subdirectory and to copy and compile files into that directory; a `dist` subdirectory, which holds the packaged module file; and a `client-jar` directory, which holds the retrieved application client JAR.

Related Third-Party Web Site References

Third-party URLs are referenced in this document and provide additional, related information.

Note – Sun is not responsible for the availability of third-party web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused or alleged to be caused by or in connection with use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

- The SailFin project home (`https://sailfin.dev.java.net`)
- The GlassFish project home (`https://glassfish.dev.java.net`)
- JSR 289: SIP Servlet 1.1 Specification (`http://jcp.org/en/jsr/detail?id=289`)
- SIP Servlet 1.1 Javadocs

Documentation, Support, and Training

The Sun web site provides information about the following additional resources:

- Documentation (`http://www.sun.com/documentation/`)
- Support (`http://www.sun.com/support/`)
- Training (`http://www.sun.com/training/`)

Typographic Conventions

The following table describes the typographic conventions that are used in this book.

TABLE P–1 Typographic Conventions

Typeface	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`** `Password:`
aabbcc123	Placeholder: replace with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*. A *cache* is a copy that is stored locally. Do *not* save the file. **Note:** Some emphasized items appear bold online.

Shell Prompts in Command Examples

The following table shows the default UNIX® system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell	`machine_name%`
C shell for superuser	`machine_name#`
Bourne shell and Korn shell	`$`
Bourne shell and Korn shell for superuser	`#`

◆ ◆ ◆ CHAPTER 1

Overview of Session Initiation Protocol (SIP) Application Development

This chapter describes the SIP protocol and the background needed for developing SIP applications using the Java programming language.

About the SIP Protocol

The session initiation protocol (SIP) is a simple network signalling protocol for creating and terminating sessions with one or more participant. The SIP protocol is designed to be independent of the underlying transport protocol, so SIP applications can run on TCP, UDP, or other lower-layer networking protocols.

Typically, the SIP protocol is used for internet telephony and multimedia distribution between two or more endpoints. For example, one person can initiate a telephone call to another person using SIP, or someone may create a conference call with many participants.

The SIP protocol was designed to be very simple, with a limited set of commands. It is also text-based, so human can read the SIP messages passed between endpoints in a SIP session.

SIP Requests

The SIP protocol defines some common request types:

TABLE 1–1 Common SIP Requests

SIP Request	Description
`INVITE`	initiate a session between two participants
`ACK`	the client acknowledges receiving the final message from an `INVITE` request

TABLE 1–1 Common SIP Requests *(Continued)*

SIP Request	Description
`BYE`	terminates a connection
`CANCEL`	cancels any pending actions, but does not terminate any accepted connections
`OPTIONS`	queries the server for a list of capabilities
`REGISTER`	registers the address in the `To` header with the server

SIP requests are codes used to indicate the various stages in a connection between SIP-enabled entities.

See "SIP Requests" on page 41 for a list of all SIP requests.

SIP Responses

The SIP Protocol uses response codes similar to the HTTP protocol. Some common response codes are as follows:

- 100 (Trying)
- 200 (OK)
- 404 (Not found)
- 500 (Server internal failure)

See "SIP Responses" on page 42 for more information on SIP responses.

What Are SIP Servlets?

A *SIP servlet* is a Java programming language server-side component that performs SIP signalling. SIP servlets are managed by a SIP servlet container, which typically are part of a SIP-enabled application server. SIP servlets interact with clients by responding to incoming SIP requests and returning corresponding SIP responses.

SIP servlets are built off the generic servlet API provided by the Java Servlet Specification.

Differences Between HTTP Servlets and SIP Servlets

SIP servlets differ from typical HTTP servlets used in web applications in the following ways:

- HTTP servlets have a particular context (called the context-root) in which they run, while SIP servlets have no context.
- HTTP servlets typically return HTML pages to the requesting client, while SIP servlets typically connect SIP-enabled clients to enable telecommunications between the client and server.

- SIP is a peer-to-peer protocol, unlike HTTP, and SIP servlets can originate SIP requests, unlike HTTP servlets which only send responses to the originating client.
- SIP servlets often act as proxies to other SIP endpoints, while HTTP servlets are typically the final endpoint for incoming HTTP requests.
- SIP servlets can generate multiple responses for a particular request.
- SIP servlets can communicate asynchronously, and are not obligated to respond to incoming requests.
- SIP servlets often work in concert with other SIP servlets to respond to particular SIP requests, unlike HTTP servlets which typically are solely responsible for responding to HTTP requests.

SIP Servlets and Java EE Components

This section describes how SIP servlets can integrate with other Java EE components in a *converged application*. A converged application has one or more SIP servlets and one or more Java EE components, such as HTTP servlets, JavaServer Faces applications, enterprise beans, or web services.

Converged applications allow you to integrate SIP functionality into Java EE applications and frameworks. For example, a web application that acts as a front-end to an employee contact information database could be enhanced by allowing users to make a Voice Over Internet Protocol (VOIP) call to the employee for whom the user is searching. Or, an application could route incoming calls to employees based on their schedule in a calendar server.

SIP Servlet Methods

A SIP servlet is a Java programming language class that extends the `javax.servlet.sip.SipServlet` class, optionally overriding `SipServlet`'s methods. These methods correspond to the SIP protocol's requests, and are named `do`*Request* where *Request* is a SIP request name. For example, the `doRegister` method will respond to incoming SIP `REGISTER` requests. See "SIP Requests" on page 11 for a list of all request methods.

`SipServlet` also defines several response methods: `doProvisionalResponse` for SIP 100 series responses; `doSuccessResponse` for SIP 200 series responses; `doRedirectResponse` for SIP 300 series responses; and `doErrorResponse` for SIP 400, 500, and 600 series responses. See "SIP Responses" on page 12 for more information about SIP responses.

All the response methods in `SipServlet` are empty, and a typical SIP servlet will override these methods. All the other request methods defined in `SipServlet` will reject any incoming corresponding SIP requests with a SIP 500 error (server error) response if the request method is not overridden.

SIP Annotations

SIP Servlet 1.1 defines four annotations that may be used in SIP applications. Using these annotations simplifies SIP application development by making the `sip.xml` deployment descriptor optional. See "The `sip.xml` Deployment Descriptor" on page 26.

TABLE 1–2 SIP Annotations

Annotation	Description
`@SipServlet`	Marks the class as a SIP servlet.
`@SipListener`	Marks the class as an implementation class of one of the SIP listeners.
`@SipApplication`	An application-level class to define a collection of SIP servlets.
`@SipApplicationKey`	Associates an incoming request and SIP session with a particular `SipApplicationSession`.

Using the `@SipServlet` Annotation

The `javax.servlet.sip.annotation.SipServlet` class-level annotation is used to mark the class as a SIP servlet.

EXAMPLE 1–1 Example of the `@SipServlet` Annotation

```
@SipServlet
public class MyServlet extends SipServlet {
    ...
}
```

`@SipServlet` has the following elements:

TABLE 1–3 `@SipServlet` Elements

Element	Description
`applicationName`	Explicitly associates the SIP servlet with a particular SIP application. This element is optional.
`description`	An optional description of this SIP servlet.

TABLE 1–3 `@SipServlet` Elements *(Continued)*

Element	Description
`loadOnStartup`	An `int` value representing the order this SIP servlet should be loaded on application deployment. The default value is `-1`, meaning the SIP servlet will not load until the container receives a request that the servlet handles. The lower the non-negative integer value in `loadOnStartup`, the earlier the SIP servlet will be initialized.
`name`	An optional name for this SIP servlet.

Using the `@SipListener` Annotation

The `javax.servlet.sip.annotation.SipListener` class-level annotation is used to mark the class as an implementation class of one of the SIP event listener interfaces. See "SIP Listeners" on page 20 for information on SIP listeners.

TABLE 1–4 `@SipListener` Elements

Element	Description
`applicationName`	Explicitly associates the SIP listener with a particular SIP application. This element is optional.
`name`	An optional name for this SIP listener.

Using the `@SipApplication` Annotation

The `javax.servlet.sip.annotation.SipApplication` application-level annotation is used to define a collection of SIP servlets and SIP listeners with a common configuration. `@SipApplication` is annotated at the package level, and all SIP servlets or listeners within the package are part of the defined SIP application unless the SIP servlet or listener explicitly sets the `applicationName` element in the `@SipServlet` or `@SipListener` annotation, respectively.

`@SipApplication` should be annotated either in a `package-info.java` file in a package hierarchy, or before the package definition in a particular source file.

EXAMPLE 1–2 Example of `@SipApplication` Annotation in a `package-info.java` File

```
@SipApplication(name="MySipApplication")
package com.example.sip;
```

TABLE 1–5 `@SipApplication` Elements

Element	Description
`name`	The name of the logical collection of SIP servlets and listeners. This element is required.
`description`	Optional description of the SIP application.
`displayName`	Optional name for displaying in container administration tools. Defaults to the value of the `name` element.
`distributable`	Optional boolean indicating whether the application may be distributed by the container in a clustered environment. The default value is `false`.
`largeIcon`	An optional `String` indicating the location, relative to the root path of the archive, of a large icon for representing this application in container administration tools.
`mainServlet`	The optional name of the main SIP servlet for this application.
`proxyTimeout`	An optional `int` value indicating the number of whole seconds before a timeout for all proxy operations in this SIP application.
`sessionTimeout`	An optional `int` value indicating the number of whole minutes before an application session timeout for all application sessions in this SIP application.
`smallIcon`	An optional `String` indicating the location, relative to the root path of the archive, of a small icon for representing this application in container administration tools.

Using the `@SipApplicationKey` Annotation

The `javax.servlet.sip.annotation.SipApplicatonKey` method-level annotation associates an incoming request with a particular `SIpApplicationSession` instance. The method annotated by `@SipApplicationKey` must:

- be public.
- be static.
- return a `String`.
- define a single argument of type `SipServletRequest`.
- not modify the passed-in `SipServletRequest` object.

The returned `String` is the key used to associate the request with a `SipApplicationSession` instance.

EXAMPLE 1–3 Example of `@SipApplicationKey`

```
@SipApplication
package com.example.sip;
...
public class MySipApplication {
    @SipApplicationKey
    public static String sessionKey (SipServletRequest req) {
        return hash(req.getRequestURI() + getDomain(req.getFrom());
    }
}
```

Only one `@SipApplicationKey` method should be defined for a particular SIP application.

TABLE 1–6 `@SipApplicationKey` Elements

Element	Description
`applicationName`	Explicitly associates the SIP application key with a particular SIP application. This element is optional.

Using `SipFactory` to Create SIP Servlet Instances

The `javax.servlet.sip.SipFactory` interface defines several abstractions useful in SIP applications. SIP applications use the container's `SipFactory` instance to create :

- requests using the `createRequest` methods.
- address objects such as `URI`, `SipURI`, `Address`, and `Parameterable` instances.
- application sessions.

For a full description of `SipFactory`'s methods, see the SIP Servlet 1.1 Javadocs.

Use the `javax.annotations.Resource` annotation to inject an instance of `SipFactory` in a class.

EXAMPLE 1–4 Injecting an Instance of `SipFactory` into a Class

```
@Resource
SipFactor sf;
```

You may also look up the container's `SipFactory` instance through the servlet context.

EXAMPLE 1–5 Looking Up `SipFactory`

```
SipFactory sf =
        (SipFactory) getServletContext().getAttribute("javax.servlet.sip.SipFactory");
```

SIP Sessions

SIP servlets, like HTTP servlets, are stateless, meaning that they do not store data across requests. SIP sessions allow SIP servlets to associate SIP messages with data stored by the SIP container. This allows an application to provide functionality across a number of discreet requests, and associating that series of requests with a single client.

The `javax.servlet.sip.SipSession` interface is SIP the equivalent of `javax.servlet.http.HttpSession` interface. Instances of `SipSession` store SIP session data and associate SIP user-agents so that they may communicate in a multiple-request dialog.

Many SIP applications, however, use multiple protocols (for example, a converged web and SIP application uses both HTTP and SIP sessions), provide functionality across dialogs (for example, a teleconferencing application involving multiple user-agents), or are used in concert with other applications for a single VOIP call. The type of data stored in an instance of `SipSession` does not cover these complicated use-cases. The `javax.servlet.sip.SipApplicationSession` interface defines methods for storing protocol information for both SIP and other protocols (for example, HTTP), and storing session data for the entire application. `SipApplicationSession` instances represent application instances, and the all the data and protocol information needed to provide the functionality in an application.

`SipApplicationSession` Methods

`SipApplicationSession` defines a number of methods for managing application sessions and session data.

`SipApplicationSession` Data Methods

Storing and retrieving session data is accomplished by using the following methods:

TABLE 1–7 `SipApplicationSession` Data Methods

Method	Description
`getAttributes(String id)`	Returns the object bound to the specified ID. Returns null if no such object ID exists.
`getAttributeNames()`	Returns an `Iterator` over the String IDs of the objects bound to this application session.
`setAttribute(String name, java.lang.Object attribute)`	Binds an object to the session using the specified String as the object's ID for later retrieval.
`removeAttribute(String name)`	Removes an object from the session by specifying the bound object's ID.

`SipApplicationSession` Protocol Methods

Instances of `SipApplicationSession` typically have multiple protocol sessions contained within the application session. Such protocol sessions are called *child sessions.* The following table lists the methods defined in `SipApplicationSession` for managing child sessions:

TABLE 1–8 Child Session Methods in `SipApplicationSession`

Method	Description
`getSessions()`	Retrieves an `Iterator` over all valid child protocol sessions.
`getSessions(String protocol)`	Retrieves an `Iterator` over all valid child sessions for a particular protocol. For example, passing SIP to `getSessions` will return all SIP protocol sessions.
`getSipSession(String id)`	Retrieves a particular session by its ID.
`getSession(String id, String protocol)`	Retrieves a particular session associated with the specified protocol by its ID.

`SipApplicationSession` Lifecycle Methods

The following table lists the methods defined in `SipApplicationSession` for managing the SIP application session lifecycle:

TABLE 1–9 `SipApplicationSession` Lifecycle Methods

Method	Description
`getCreationTime()`	Returns the time that the `SipApplicationSession` instance was created as a `long` value representing the number of milliseconds since midnight January 1, 1970 GMT.
`getExpirationTime()`	Returns the time that the `SipApplicationSession` will expire as a `long` value representing the number of milliseconds since midnight January 1, 1970 GMT.
`getInvalidateWhenReady()`	Returns a boolean value specifying whether the container will notify the application when the `SipApplicationSession` instance is ready to be invalidated.
`getLastAccessedTime()`	Returns the time that the `SipApplicationSession` instance was last accessed as a `long` value representing the number of milliseconds since midnight January 1, 1970 GMT.
`setInvalidateWhenReady(boolean invalidateWhenReady)`	Tells the container to notify the application when the `SipApplicationSession` instance is ready to be invalidated.
`invalidate()`	Explicitly invalidates the SIP application session and unbinds any objects bound to the session.

TABLE 1–9 `SipApplicationSession` Lifecycle Methods *(Continued)*

Method	Description
`isReadyToInvalidate()`	Returns a boolean value specifying whether the `SipApplicationSession` instance is ready to be invalidated.
`isValid()`	Returns a boolean value specifying whether the `SipApplicationSession` instance is valid.
`setExpires(int deltaMinutes)`	Extends the time of expiry for the `SipApplicationSession` instance by the number of minutes specified by `deltaMinutes`. If `deltaMinutes` is 0 or a negative number, the session will never expire. Returns an `int` value of the number of minutes by which the session was extended. If it returns 0, the session was not extended.

Using `SipSessionsUtil` to Manage SIP Sessions

The `SipSessionsUtil` interface defines utility methods for managing SIP sessions in a converged application. Use the `javax.annotations.Resource` annotation to inject the container's `SipSessionsUtil` implementation class in your SIP servlets:

EXAMPLE 1–6 Example of Injecting `SipSessionsUtil` into a Class

```
@Resource
SipSessionsUtil sipSessionsUtil;
```

You may also manually look up `SipSessionsUtil` through the servlet context.

EXAMPLE 1–7 Example of Looking Up `SipSessionsUtil`

```
SipSessionsUtil sipSessionsUtil =
    (SipSessionsUtil) getServletContext().
        getAttribute("javax.servlet.sip.SipSessionsUtil");
```

For more information, see the SIP Servlet 1.1 Javadocs

SIP Listeners

SIP application listeners are Java servlet application listeners that listen for SIP-specific events. SIP applications implement the SIP event listener interfaces by marking the implementation class with a `javax.servlet.sip.annotation.SipListener` annotation.

EXAMPLE 1–8 Example of `@SipListener`

```
@SipListener
public class MyListener implements SipServletListener {
    ...
```

EXAMPLE 1–8 Example of `@SipListener` *(Continued)*

```
}
```

Sip servlet classes may also implement the SIP event listener interfaces.

EXAMPLE 1–9 Example of SIP Listener in SIP Servlet Class

```
@SipListener
@SipServlet
public class MySipServlet extends SipServlet implements SipServletListener {
    ...
}
```

SIP Servlet Listeners

The following SIP servlet listeners, in package `javax.servlet.sip`, are available to SIP servlet developers:

TABLE 1–10 SIP Servlet Listeners

Listener	Description
`SipServletListener`	Implementations of `SipServletListener` receive notifications on initialization of `SipServlet` instances. See the SIP Servlet 1.1 Javadocs for more information.

SIP Application Session Listeners

The following SIP application listeners, in package `javax.servlet.sip`, are available to SIP servlet developers:

TABLE 1–11 SIP Application Listeners

Listener	Description
`SipApplicationSessionListener`	Implementations of `SipApplicationSessionListener` receive notifications when `SipApplicationSession` instances have been created, destroyed, timed out, or are ready to be invalidated. See the SIP Servlet 1.1 Javadocs for more information.
`SipApplicationSessionAttributeListener`	Implementations of `SipApplicationSessionAttributeListener` receive notifications when attributes are added, removed, or modified in `SipApplicationSession` instances. See the SIP Servlet 1.1 Javadocs for more information.

TABLE 1–11 SIP Application Listeners *(Continued)*

Listener	Description
`SipApplicationSessionBindingListener`	Session attributes that implement `SipApplicationSessionBindingListener` receive notifications when they are bound or unbound to `SipApplicationSession` instances. See the SIP Servlet 1.1 Javadocs for more information.
`SipApplicationSessionActivationListener`	Implementations of `SipApplicationSessionActivationListener` receive notifications when `SipApplicationSession` instances are activated or passivated. See the SIP Servlet 1.1 Javadocs for more information.

SIP Session Listeners

The following SIP session listeners, in package `javax.servlet.sip`, are available to SIP servlet developers:

TABLE 1–12 SIP Session Listeners

Listener	Description
`SipSessionListener`	Implementations of `SipSessionListener` receive notifications when `SipSession` instances are created, destroyed, or ready to be invalidated. See the SIP Servlet 1.1 Javadocs for more information.
`SipSessionActivationListener`	Implementations of `SipSessionActivationListener` receive notifications when `SipSession` instances are activated or passivated. See the SIP Servlet 1.1 Javadocs for more information.
`SipSessionAttributeListener`	Implementations of `SipSessionAttributeListener` receive notifications when attributes are added, removed, or modified in `SipSession` instances. See the SIP Servlet 1.1 Javadocs for more information.
`SipSessionBindingListener`	Attributes that implement `SipSessionBindingListener` receive notifications when they are bound or unbound from `SipSession` instances. See the SIP Servlet 1.1 Javadocs for more information.

SIP Error Listeners

The following SIP error listeners, in package `javax.servlet.sip`, are available to SIP servlet developers:

TABLE 1–13 SIP Error Listeners

Listener	Description
`SipErrorListener`	Implementations of `SipErrorListener` receive notifications when an expected `ACK` or `PRACK` SIP message is not received. See the SIP Servlet 1.1 Javadocs for more information.

SIP Timer Listeners

The following SIP timer listeners, in package `javax.servlet.sip`, are available to SIP servlet developers:

TABLE 1–14 SIP Timer Listeners

Listener	Description
`TimerListener`	Implementations of `TimerListener` receive notifications when `ServletTimer` instances have fired. See the SIP Servlet 1.1 Javadocs for more information.

For information on SIP timers, see "SIP Timers" on page 23.

SIP Timers

The SIP timer service is provided by the SIP servlet container to allow SIP applications to schedule and manage timers, and receive notifications when timers expire. Timers are events that can be scheduled to run once at a specific time, or to repeat at configurable intervals. Timers may be persistent, in which case the timer will be preserved across Communications Application Server 1.5 restarts. Persistent timers will be fired on server startup if the server was shut down when the timer was supposed to fire.

Repeating timers can be either fixed-delay or fixed-rate. Both fixed-delay and fixed-rate timers will fire at approximately regular intervals, but fixed-delay timers will fire regardless of whether previous timer firings were late. Fixed-rate timers are rescheduled based on the absolute time.

Managing SIP Timers

The container provides a `javax.servlet.sip.TimerService` implementation that allows you to create timers, which are `javax.servlet.sip.ServletTimer` instances. The `TimerService` interface defines the following methods for creating timers:

TABLE 1–15 `TimerService` Timer Creation Methods

Method	Description
`createTimer(SipApplicationSession session, long delay, boolean isPersistent, Serializable info)`	Creates a single, optionally persistent timer associated with the specified SIP application session. The `delay` parameter is the time in milliseconds before a timer fires. The info parameter is the application information delivered when the timer expires.
`createTimer(SipApplicationSession session, long delay, long period, boolean fixedDelay, boolean isPersistent, Serializable info)`	Creates a recurring, optionally persistent timer associated with the specified SIP application session. The `delay` parameter is the time in milliseconds before the timer first fires. The `period` parameter is the time in milliseconds after the first timer firing that the timer will fire again. The `fixedDelay` parameter specifies whether the timer is fixed-delay or fixed-rate. The info parameter is the application information delivered when the timer expires.

The `ServletTimer` interface defines the following methods for managing a particular timer:

TABLE 1–16 `TimerService` Timer Management Methods

Method	Description
`cancel()`	Cancels the timer.
`getApplicationSession()`	Returns the `SipApplicatonSession` instance the timer is associated with.
`getId()`	Returns the ID of the timer as a String.
`getInfo()`	Returns a `Serializable` object of the information specified when the timer was created.
`getTimeRemaining()`	Returns a `long` representing the number of milliseconds until the timer is scheduled to next fire.
`scheduledExecutionTime()`	Returns a `long` representing the most recent time the timer was scheduled to fire. If the timer has not yet fired, the return value is undefined.

For more information on the `TimerService` interface, see the SIP Servlet 1.1 Javadocs.

Back-to-Back User Agent Applications

A *back-to-back user agent* (B2BUA) is a SIP element that acts as an endpoint for two or more SIP dialogs, forwarding requests and responses between the dialogs. B2BUA applications are extremely common SIP applications, and SIP Servlet 1.1 defines a helper class, `javax.servlet.sip.B2buaHelper`to simplify the creation of B2BUA applications. B2BUA

applications have the potential to break end-to-end communication between endpoints because they sit between two endpoints in a communication chain. Using `B2buaHelper` minimizes some of the risk of breaking the signaling between two endpoints.

Understanding the `B2buaHelper` Class

The `B2buaHelper` class contains all the necessary methods for creating B2BUA applications. It is retrieved by calling `SipServerRequest.getB2buaHelper`.

EXAMPLE 1–10 Example of Retrieving `B2buaHelper`

```
private void sendInfoToClient(SipServletResponse resp) {
    SipServletRequest req = resp.getRequest();
    B2buaHelper b2buaHelper = req.getB2buaHelper();
    ...
}
```

A typical B2BUA application has two SIP sessions, one for each client. The `B2buaHelper` class is typically used to create requests that are then forwarded to and from the SIP sessions. retrieve linked sessions.

For a complete list of `B2buaHelper`'s methods, see SIP Servlet 1.1 Javadocs.

Creating Requests with `B2buaHelper`

Once you've retrieved `B2buaHelper`, you can use it to link two SIP sessions by creating requests using the `createRequest` method.

EXAMPLE 1–11 Creating a Request Using `B2buaHelper`

```
SipServletRequest clientRequest =
        b2buaHelper.createRequest(serverReq, true, headerMap);
```

The `createRequest` method takes a `SipServletRequest` instance of the original request, an optional boolean indicating whether the sessions should be linked, and a `java.util.Map<String,java.util.Set>` map of headers that will be used instead of the headers in the original request. The `From` and `To` headers are the keys in the map. The only headers that can be set using this map are non-system headers and the `From`, `To`, and `Route` headers.

See Example 2–6 for the full method where `B2buaHelper` is used to create a request that links two sessions.

Retrieving Linked Sessions Using `B2buaHelper`

Once two client's sessions are linked you can then retrieve the sessions using `getLinkedSession`.

EXAMPLE 1–12 Retrieving Linked Sessions Using `B2buaHelper`

```
private void sendByeToServer(SipServletRequest clientBye)
        throws ServletException, IOException {
    B2buaHelper b2buaHelper = clientBye.getB2buaHelper();
    SipSession serverSession = b2buaHelper.getLinkedSession(clientBye.getSession());
    SipServletRequest serverBye = serverSession.createRequest("BYE");
    logger.info("Sending BYE request.\n" + serverBye);
    serverBye.send();
}
```

SIP Servlets and the SIP Servlet Container

The SIP servlet container manages the lifecycle of SIP servlets, enables network communication for SIP requests and responses by listening on a particular listening point, and provides optional services such as security and interaction with other server-side components.

Structure of a SIP Application

A typical SIP application consists of the following programming artifacts:

- One or more SIP servlets.
- Optional utility and helper classes such as SIP listeners.
- Static resources used by the classes.
- Metadata and optional configuration files.

The `sip.xml` Deployment Descriptor

The optional `sip.xml` deployment descriptor is used by the SIP servlet container to process deployed SIP applications and configure the runtime to properly respond to incoming SIP requests. It is similar in structure to `web.xml` deployment descriptor used by Java EE web applications. You may bypass the need for a fully defined `sip.xml` if you use SIP annotations in your application.

Packaging a SIP Application

SIP applications are packaged in either SAR (SIP archive) or WAR (web archive) files. These archives are standard Java archives (JAR). The SAR format is similar to and based on the WAR format, including the use of the presence of the `WEB-INF` folder that contains class files and deployment descriptors. SIP containers will recognize either the `.sar` or `.war` extensions when processing SIP applications.

Converged applications may be packaged in WAR files, or the SAR or WAR file may be itself packaged within an Enterprise archive (EAR), similar to a typical Java EE application. This

means a SIP application that has been packaged in a SAR or WAR may be packaged with enterprise bean components, Java Persistence API JARs, and any other Java EE component that is allowed to be packaged in EAR files.

◆ ◆ ◆ CHAPTER 2

Simple SIP Servlet Examples

This chapter describes several of the simple SIP servlet examples that are included with Communications Application Server 1.5.

Prerequisites for Running the Examples

You should have done the following before you can run the examples:

1. Downloaded and installed the example bundle as described in "Sample Applications" on page 6.
2. Installed NetBeans IDE as described in "NetBeans IDE" on page 6.
3. Downloaded and installed the SIP NetBeans IDE modules, including the SIP Test Agent module as described in "SIP Modules for NetBeans IDE" on page 6.

The `SipProxy` Example

This example is a simple SIP proxy servlet. The proxy servlet will forward all SIP messages from the caller client to the callee server.

Developing the SIP Servlet

The SIP servlet is called `SimpleProxyServlet`, and extends the base `SipServlet` class and implements the `SipErrorListener` and `Servlet` interfaces.

```
@SipListener
@SipServlet
public class SimpleProxyServlet
            extends SipServlet
```

```
        implements SipErrorListener,Servlet {

    /** Creates a new instance of SimpleProxyServlet */
    public SimpleProxyServlet() {
    }


    protected void doInvite(SipServletRequest request)
            throws ServletException, IOException {

            if (request.isInitial()) {
                Proxy proxy = request.getProxy();
                proxy.setRecordRoute(true);
                proxy.setSupervised(true);
                proxy.proxyTo(request.getRequestURI()); // bobs uri
            }
            System.out.println("SimpleProxyServlet: Got request:\n" + request);
        }

    protected void doBye(SipServletRequest request) throws
                ServletException, IOException {

        System.out.println("SimpleProxyServlet: Got BYE request:\n" + request);
        super.doBye(request);
    }


    protected void doResponse(SipServletResponse response)
        throws ServletException, IOException {

        System.out.println("SimpleProxyServlet: Got response:\n" + response);
            super.doResponse(response);
    }

    // SipErrorListener

    public void noAckReceived(SipErrorEvent ee) {
        System.out.println("SimpleProxyServlet: Error: noAckReceived.");
    }

    public void noPrackReceived(SipErrorEvent ee) {
            System.out.println("SimpleProxyServlet: Error: noPrackReceived.");
    }

}
```

SIP Methods

In SimpleProxyServlet, you override several methods to respond to the main SIP methods.

- doInvite- responds to INVITE requests.

  In SimpleProxyServlet, upon receiving an INVITE request the servlet will create a javax.servlet.sip.Proxy instance, set some options, and proxy the request to the target SIP server.

- doBye- responds to BYTE requests.

  In SimpleProxyServlet, the servlet logs a message upon receiving a BYE message, and calls the doBye method of the parent class (javax.servlet.sip.SipServlet).

SipErrorListener Methods

Because SimpleProxyServlet implements the SipErrorListener interface, it must implement the following methods:

- noAckReceived is invoked to notify the application that no ACK message was received for an INVITE transaction.
- noPrackReceived is invoked when no PRACK message was received for a previously sent response.

Deploying and Running SipProxy

Follow these instructions to deploy and run the example.

▼ Deploying and Running SipProxy in NetBeans IDE

1. **Click File→Open Project and navigate to the location where you downloaded and expanded the SimpeProxy example.**
2. **Select SipProxy and click Open Project.**
3. **Right-click on SipProxy in the Projects pane and click Run.**

▼ Testing SipProxy with the SIPp Application

Before You Begin Be sure you have installed the SIPp test application, as described in "SIPp" on page 6.

1. **In a terminal, enter the following command to start the SIPp server on port 5090:**

```
% sipp -sn uas -p 5090
```

2. **In a new terminal enter the following command to start the SIPp client on port 5080:**

```
% sipp -sn uac -rsa 127.0.0.1:5060 -p 5080 127.0.0.1:5090
```

You should now see the messages from the client get returned by the server, with the `SipProxy` application acting as a proxy between them.

The Click-To-Dial Example

The Click-To-Dial example demonstrates how to integrate a SIP servlet with a web application by allowing users to place calls to other users by using an HTTP servlet. The example demonstrates how SIP registration and invitation works, and how to share data between SIP servlets and HTTP servlets.

Architecture of the Click-To-Dial Example

The Click-To-Dial application allows users to call each other after registering their information using a web application. The example consists of two SIP servlets (`RegistrarServlet` and `CallSipServlet`) and two HTTP servlets (`LoginServlet` and `PlaceCallServlet`). The user data is stored in a database using the Java Persistence API.

The following scenario shows the procedure for using the Click-To-Dial example:

1. Users Alice and Bob login to the web application, using the `LoginServlet` HTTP servlet.
2. Alice and Bob register their SIP soft-phone with the web application. Registration is handled by the `RegistrarServlet` SIP servlet, which stores registration data in a database using the Java Persistence API.
3. Alice clicks on Bob's Call link from the web application to start a phone call to Bob. The `PlaceCallServlet`HTTP servlet passes the data to `CallSipServlet` in order to initiate the connection.
4. Alice's phone rings.
5. When Alice picks up her phone, a call is placed to Bob's phone, and Bob's phone rings.
6. When Bob picks up his phone, the connection is established, and Alice and Bob can have a conversation.
7. When Alice or Bob hangs up, the connection is terminated, and they are able to receive calls again.

Click-To-Dial's SIP Servlets

The SIP functionality in Click-To-Dial is split into two separate SIP servlets, `RegistrarServlet` and `CallSipServlet`.

SIP Application Annotations in `CllickToDial`

A `@SipApplication` annotation is used in `ClickToDial` to define a set of SIP servlets used together to provide SIP functionality. The `@SipApplication` annotation is set at the package level by putting it in the `package-info.java` file in the `clicktodial.sip` package.

EXAMPLE 2–1 Package-level `@SipApplicaton` Annotation in `ClickToDial`

```
@javax.servlet.sip.annotation.SipApplication(
    name="ClickToDial",
    mainServlet="RegistrarServlet")
package clicktodial.sip;
```

The `@SipApplication` annotation sets two elements: the name of the application, and the main servlet. The `name` element is required, and is set to the application name. The optional `mainServlet` element defines which SIP servlet will initially respond to SIP requests. In this case, the `RegistrarServlet`, which registers SIP clients so they can be later contacted for calls, is the main servlet for `ClickToDial`.

The `RegistrarServlet`

The `RegistrarServlet` allows users to register soft-phones with the application, and stores the user's data in a database using the Java Persistence API.

`RegistrarServlet` has three methods: `doRegister`, `handleRegister`, and `handleUnregister`.

The `doRegister` method responds to `REGISTER` messages and performs some checks on the incoming request, extracts the user name from the request, looks the user up in the database of users, and examines the `EXPIRES` header of the request to determine whether the request is a registration or unregistration request. If it is a registration request, the `handleRegister` private helper method is called. If it is an unregistration request, the `handleUnregister` private helper method is called. These methods will return a SIP response to send back to the client.

EXAMPLE 2–2 The `doResponse` Method

```
@Override
protected void doRegister(SipServletRequest req)
            throws ServletException, IOException {
        logger.info("Received register request: " + req.getTo());

        int response = SipServletResponse.SC_SERVER_INTERNAL_ERROR;
        ModelFacade mf = (ModelFacade) getServletContext().getAttribute("Model");

        // Figure out the name the user is registering with.  This is the
        // user portion of the SIP URI, e.g. "Bob" in "sip:Bob@x.y.z:port"
        String username = null;
```

EXAMPLE 2–2 The doResponse Method *(Continued)*

```
        if (req.getTo().getURI().isSipURI()) {
            username = ((SipURI) req.getTo().getURI()).getUser();
        }

        // get the Person object from the database
        Person p = mf.getPerson(username);
        if (p != null) {
            // the Expires header tells us if this is a registration or
            // unregistration attempt.  An expires value of 0 or no Expires
            // header means it is an unregistration.
            int expires = 0;
            String expStr = req.getHeader("Expires");
            if (expStr != null) {
                expires = Integer.parseInt(expStr);
            }

            if (expires == 0) {
                // unregister
                response = handleUnregister(req, p);
            } else {
                // register
                response = handleRegister(req, p);
            }
        } else {
            // no person found in the database
            response = SipServletResponse.SC_NOT_FOUND;
        }

        // send the response
        SipServletResponse resp = req.createResponse(response);
        resp.send();
    }
```

The `handleRegister` method extracts the user's SIP address from the request, stores it in the user database, and returns a SIP `OK` response. The user can now place and receive calls.

EXAMPLE 2–3 The handleRegister Method

```
private int handleRegister(SipServletRequest req, Person p)
        throws ServletException {

        // Get the contact address from the request.  Prefer the
        // "Contact" address if given, otherwise use the "To" address
        Address addr = req.getTo();
        String contact = req.getHeader("Contact");
```

EXAMPLE 2–3 The `handleRegister` Method *(Continued)*

```
        if (contact != null) {
            addr = sf.createAddress(contact);
        }

        logger.info("Register address: " + addr);

        // store the contact address in the database
        p.setTelephone(addr.getURI().toString());

        ModelFacade mf = (ModelFacade) getServletContext().getAttribute("Model");
        mf.updatePerson(p);

        return SipServletResponse.SC_OK;
    }
```

The `handleUnregister` method removes the user's SIP address from the database by setting it to `null`, then sends a SIP `OK` response back. The user cannot place or receive calls after being unregistered.

EXAMPLE 2–4 The `handleUnregister` Method

```
private int handleUnregister(SipServletRequest req, Person p) {
        // store the contact address in the database
        p.setTelephone(null);

        ModelFacade mf = (ModelFacade) getServletContext().getAttribute("Model");
        mf.updatePerson(p);

        return SipServletResponse.SC_OK;
    }
```

The `CallSipServlet`

The `CallSipServlet` SIP servlet connects registered SIP users to one another, allowing users to place calls to one another. There are 5 main SIP methods in `CallSipServlet`: `doSuccessResponse`, `sendInviteToClient`, `sendAckToClient`, `sendAckToServer`, and `sent200OKToClient`.

`CallSipServlet` is annotated at the class-level with a `@SipServlet` and `@SipListener` annotation.

```
@javax.servlet.sip.annotation.SipServlet
@SipListener
public class CallSipServlet extends SipServlet implements SipSessionListener {
```

```
    ...
}
```

The `doSuccessResponse` method connects a call between two registered users. When the first user Alice initiates a call to the second user Bob, first Alice's phone rings. If Alice answers her phone, a SIP `OK` message is sent. At that point, Bob's address is extracted from the request, a SIP `INVITE` message is sent to Bob's address by calling the `sendInviteToClient` private method, and Bob's phone rings. If Bob answers the phone, a SIP `OK` message is sent. The two SIP sessions, from Alice and Bob respectively, are linked, and a SIP `ACK` message is sent to both user's phones by calling the `sendAckToClient` and `sendAckToServer` private methods. Alice and Bob are now connected and can have a conversation. When the call is terminated, a `BYE` message is sent from the server, and the `send200OKToClient` private method is called.

EXAMPLE 2–5 The `doSuccessResponse` Method

```
@Override
protected void doSuccessResponse(SipServletResponse resp)
        throws ServletException, IOException {
    logger.info("Received a response.\n" + resp);

    if (resp.getMethod().equals("INVITE")) {
        List<SipSession> sipSessions = getSipSessions(resp.getApplicationSession());
        if (sipSessions.size() == 1) {
            sipSessions.get(0).setAttribute("ACK", resp.createAck());
            sendInviteToClient(resp);
        } else { // 200 OK from Client
            sendAckToClient(resp);
            sendAckToServer(resp);
        }
    } else if (resp.getMethod().equals("BYE")) {
        send200OKToClient(resp);
    }
}
```

EXAMPLE 2–6 The `sendInviteToClient` Method

```
private void sendInviteToClient(SipServletResponse serverResp)
        throws ServletException, IOException {
    SipServletRequest serverReq = serverResp.getRequest();
    B2buaHelper b2buaHelper = serverReq.getB2buaHelper();

    // Swap To & From headers.
    Map<String, List<String>> headerMap = new HashMap<String, List<String>>();
    List<String> from = new ArrayList<String>();
    from.add(serverResp.getHeader("From"));
    headerMap.put("To", from);
    List<String> to = new ArrayList<String>();
```

EXAMPLE 2–6 The `sendInviteToClient` Method *(Continued)*

```
    to.add(serverResp.getHeader("To"));
    headerMap.put("From", to);

    SipServletRequest clientRequest = b2buaHelper
                .createRequest(serverReq, true, headerMap);
    clientRequest.setRequestURI(clientRequest.getAddressHeader("To").getURI());
    if (serverResp.getContent() != null) { // set sdp1
        clientRequest.setContent(serverResp.getContent(),
                serverResp.getContentType());
    }
    logger.info("Sending INVITE to client.\n" + clientRequest);
    clientRequest.send();
}
```

EXAMPLE 2–7 The `sendAckToClient` Method

```
private void sendAckToClient(SipServletResponse clientResp)
        throws ServletException, IOException {
    SipServletRequest ack = clientResp.createAck();
    logger.info("Sending ACK to client.\n" + ack);
    ack.send();
}
```

EXAMPLE 2–8 The `sendAckToServer` Method

```
private void sendAckToServer(SipServletResponse clientResp)
        throws ServletException, IOException {
    B2buaHelper b2buaHelper = clientResp.getRequest().getB2buaHelper();
    SipSession clientSession = clientResp.getSession();
    SipSession serverSession = b2buaHelper.getLinkedSession(clientSession);
    SipServletRequest ack = (SipServletRequest) serverSession.getAttribute("ACK");
    serverSession.removeAttribute("ACK");
    if (clientResp.getContent() != null) { // set sdp2
        ack.setContent(clientResp.getContent(), clientResp.getContentType());
    }
    logger.info("Sending ACK to server.\n" + ack);
    ack.send();
}
```

EXAMPLE 2–9 The `send200OKToClient` Method

```
    protected void doBye(SipServletRequest request)
        throws ServletException, IOException
    {
```

EXAMPLE 2–9 The `send200OKToClient` Method *(Continued)*

```
        logger.info("Got bye");

        SipSession session = request.getSession();

        // end the linked call
        SipSession linkedSession = (SipSession) session.getAttribute("LinkedSession");
        if (linkedSession != null) {
            // create a BYE request to the linked session
            SipServletRequest bye = linkedSession.createRequest("BYE");

            logger.info("Sending bye to " + linkedSession.getRemoteParty());

            // send the BYE request
            bye.send();
        }

        // send an OK for the BYE
        SipServletResponse ok = request.createResponse(SipServletResponse.SC_OK);
        ok.send();
    }
```

There are three SIP session listener methods implemented in `CallSipServlet`, from the `SipSessionListener` interface: `sessionCreated`, `sessionDestroyed`, and `sessionReadyToInvalidate`. In `CallSipServlet`, the methods simply log the events.

EXAMPLE 2–10 `SipSessionListener` Methods Implemented in `CallSipServlet`

```
public void sessionCreated(SipSessionEvent sse) {
    logger.info("Session created");
}

public void sessionDestroyed(SipSessionEvent sse) {
    logger.info("Session destroyed");
}

public void sessionReadyToInvalidate(SipSessionEvent sse) {
    logger.info("Session ready to be invalidated");
}
```

Running the Click-To-Dial Example

This section describes how to deploy and run the Click-To-Dial Example in NetBeans IDE.

Deploying and Running Click-To-Dial in NetBeans IDE

1. **In NetBeans IDE, click Open Project and navigate to** `sip-tutorial/examples/ClickToDial`**.**
2. **Right-click on the** `ClickToDial` **project and select Run.**
   This will open a browser to `http://localhost:8080/ClickToDial`.

Registering Alice's SIP Phone

1. **In your web browser select** `Alice` **from the drop-down menu and click Login.**
2. **In X-Lite right-click on the phone and select SIP Account Settings.**
3. **Click Add.**
4. **Enter** `Alice` **under Display Name, User Name, and Authorization User Name.**
5. **Enter** `test.com` **under Domain.**
6. **Check Register With Domain and Receive Incoming Calls.**
7. **Under Send Outbound Via select Proxy and enter** *Communications Application Server 1.5 IP address*`:5060`**. For example,** `192.168.0.2:5060`**.**
8. **Click Ok.**

Registering Bob's SIP Phone

1. **On a different machine in your web browser go to** `http://`*Communications Application Server 1.5 IP Address*`:8080/ClickToDial`**. For example,** `http://192.168.0.2:8080/ClickToDial`**.**
2. **Select** `Bob` **from the drop-down menu and click Login.**
3. **In the second machine's X-Lite right-click on the phone and select SIP Account Settings.**
4. **Click Add.**
5. **Enter** `Bob` **under Display Name, User Name, and Authorization User Name.**
6. **Enter** `test.com` **under Domain.**
7. **Check Register With Domain and Receive Incoming Calls.**

8. **Under Send Outbound Via select Proxy and enter** *Communications Application Server 1.5 IP address*`:5060`**. For example,** `192.168.0.2:5060`**.**

9. **Click Ok.**

Placing a Call From Alice To Bob

1. **On Alice's machine, refresh the web browser to see that both Alice and Bob are registered.**

2. **Click Call next to Bob's SIP address to place a call to Bob.**

3. **In X-Lite click Answer to place the call to Bob.**

   X-Lite will initiate a call to Bob's X-Lite instance using Communications Application Server 1.5 as a proxy.

4. **On Bob's machine, click Answer to receive the call from Alice.**

   Alice and Bob are now connected and may talk.

◆ ◆ ◆ APPENDIX A

SIP Messages

This appendix describes the SIP requests and responses.

SIP Requests

The following table lists the SIP requests.

TABLE A–1 SIP Requests

SIP Request	Description
INVITE	A client is being invited to participate in a call.
ACK	The client has confirmed the INVITE request.
BYE	The call has been terminated by either the caller or callee.
CANCEL	Cancel any pending requests.
OPTIONS	Queries the server for its capabilities.
REGISTER	Registers the client with the server according to the address in the To header.
PRACK	Similar to ACK, but a provisional confirmation.
SUBSCRIBE	Subscribes the device for an event notification.
NOTIFY	Notifies all subscribers of an event.
PUBLISH	Publishes an event to a server.
INFO	Sends information in the middle of a session that doesn't modify the session's state.

TABLE A–1 SIP Requests *(Continued)*

SIP Request	Description
REFER	Asks the client to issue a SIP request, typically a call transfer.
MESSAGE	Sends an instant message using SIP.
UPDATE	Modifies a session's state without altering the dialog state.

For a list of all SIP requests and links to their definitions in their respective RFCs, see the SIP requests Wikipedia entry.

SIP Responses

TABLE A–2 SIP Responses

SIP Response	Description
`100-199`	Information responses.
`200-299`	Successful responses
`300-399`	Redirection responses
`400-499`	Client error responses
`500-599`	Server error responses
`600-699`	Global failure responses

For a list of all SIP responses, see the SIP responses Wikipedia entry.

Index

T

U

X