SIP: Creating next-generation telecom applications
The Session Initiation Protocol makes developing apps for telecommunications networks easier than ever

Level: Introductory
Developer, Ubiquity Software Corporation
First published by IBM developerWorks at http://www.ibm.com/developerWorks/.

Available from at UDN Developer Network at
https://developer.ubiquitysoftware.com/support/whitepapers/sip-creating-next-generation-telecom-applications/

30 Sep 2003

Developing applications to run on a telecommunications network has never been easier. Instead of yesterday's proprietary protocols and interfaces, you now can use open, Internet-based standards such as the Session Initiation Protocol (SIP). Combined with the power and simplicity of Java technology in the form of the SIP Servlet API, an application developer can create and deploy new services to users in a fraction of the time it previously took.

Taking advantage of this revolution means being conversant with SIP. In this article, you'll discover how SIP operates and, building on that knowledge, you will learn how to use the Java SIP Servlet API to build the exciting new applications of the future. The SIP tour concludes with code examples that demonstrate SIP application development in action.

The era of separate voice and data networks is gradually nearing an end. In the future, a single, converged network will provide the basis for all forms of communication. In addition to reducing costs, that convergence will enable a whole new range of services. The Session Initiation Protocol (SIP) will be the control mechanism at the center of this revolution.

SIP's momentum has grown, with network equipment vendors and telecommunication service providers increasingly accepting it as the protocol of choice. Additionally, the 3rd. Generation Partnership Project (3GPP) recently adopted SIP as the call control mechanism for next-generation networks. Moreover, the adoption of Voice Over IP in enterprises also continues at a quick pace. Considering all that, expect big things in the telecommunications application development space.

In this article, you'll discover the SIP protocol and the SIP Servlet API. You'll see the role played by User Agents (UA) and proxy servers in SIP, and find out how to establish communication between two devices. That provides a basis for the introduction of the SIP Servlet API and related concepts. SIP servlet behavior such as proxying, acting as an originating and terminating UA, and acting as a Back-to-Back User Agent (B2BUA) are also covered. Finally, sample SIP servlet code illustrates just how easy it is to program using the SIP Servlet API.

What is SIP?

The SIP application-layer protocol allows for the creation and management of multimedia communication sessions between devices. In 1999 the Internet Engineering Task Force (IETF) approved SIP as Request For Comment (RFC) 2543, which built on the Multiparty Multimedia Session Control (MMUSIC) Working Group's earlier development of multimedia on the Internet. Since then, SIP has evolved through several RFC revisions to the current RFC 3261.

Any media can be exchanged during a SIP session and any protocol can be used. SIP commonly works with:
Session Description Protocol (SDP), which is used to determine which media, will be used. See RFC 2327.
Real Time Protocol (RTP), which actually transports the media data during the session. See RFC 1889.

The protocol does not concern itself with the nature or content of a call, so it leaves users free to choose how they will use the session and exchange communication media. SIP sessions are therefore not limited to voice. Once a session has been established, users can exchange any type of media.

SIP concepts

The SIP RFC defines several key concepts and elements required in a SIP network:

- A *user agent* (UA), an end-point, lets users create and manage a communication session. A UA, either a SIP telephone or a software application, handles session setup and management tasks such as transfer, termination, and service invocation, to name a few. In addition, UAs can identify user availability and negotiate session capability.

- A *session* establishes when the UA (the *caller*) invites another UA (the *callee*) to join a communication session.
- A *SIP message*is a text-based entity. There are two types of messages: *requests* and *responses.* Requests are sent from one UA to another, which in turn sends a response back. (The following section discusses SIP messages in greater detail.) Any messages that pass from a caller to a callee move downstream; conversely, any messages that move in the opposite direction move upstream.
- A *SIP proxy server* typically handles registrations, implementing call-routing policies, and performing authentication and authorization. As its primary task, the SIP proxy server ensures that a request is sent to another entity closer to the targeted user. A proxy interprets, and, if necessary, rewrites specific parts of a request message before forwarding it. A SIP message might pass through several SIP proxy servers as it travels to the callee UA. A UA is usually configured to send any requests it originates to a specific SIP proxy server. The proxy server is known as an *outbound proxy* in this situation.
- A *SIP address* (also known as a *SIP URL*) uniquely identifies a user during the creation of a communication session. The address resembles an e-mail address except that it has a sip: prefix. For example, the telephone on your desk might have the following SIP address: sip:user@194.195.100.20.

Typically, you want people to call you using a SIP address with your company's domain name, for example, sip:user@ubiquity.net. To that end, SIP lets you perform a registration process to associate your SIP address with one or more UAs. Subsequently, when a call is made to the SIP address it goes to a SIP proxy server. The SIP proxy server performs a registration lookup that determines which UA the call should be directed to. You can also modify and delete registration information at any time.

The registration of several UAs against a single SIP address might cause several UAs to be notified and start ringing -- a process known as forking. There are two types of forking: sequential and parallel. With *sequential forking*, each UA is alerted in sequence after ringing for a certain time period. In contrast, with *parallel forking* all UAs are alerted simultaneously, and the session establishes with the first UA to answer or, depending on timing, more than one UA.

A *Call* represents all messages that pass between a caller UA and callee UA(s).

A Dialog is the SIP relationship between two UAs that persists for a period of time. It consists of the SIP messages that pass between the UAs, including those that pass through proxies.

A Transaction occurs between a client and a server and comprises all the messages from the first request sent from the client to the server, up to a final response sent from the server to the client.

SIP messages

SIP-based communication uses a text-based request/response model similar to the HTTP protocol. A client makes a request to a server, with the server returning the response to the client. The SIP specification defines six types of request messages and six types of response messages, as listed in Table 1.

SIP Request	**Description** Table 1
REGISTER	Creates and manages registrations.
INVITE	Initiates a communication session and is sent from the caller to the callee inviting them to join the session.
ACK	Sent by the caller after a final response has been received for an INVITE request.
BYE	Sent from caller or callee to terminate the session.
CANCEL	Used to CANCEL an INVITE request for which a final response has not yet been received.
OPTIONS	Request media information.

SIP	**Description**

Response	
1xx Information	Provides progress response to the caller, for example 100 Trying, 180 Ringing.
2xx Success	Confirms that a request has been accepted.
3xx Redirect	Notifies the caller of an alternative location to where the request should be sent.
4xx Request Failure	Indicates that a request has not been processed successfully; for example, 404 Not Found.
5xx Server Failure	Indicates that the server itself has erred. For example, a 503 response indicates that a server is temporarily unable to process the request due to overloading or maintenance of the server.
6xx Global Failure	Indicates a global failure regarding a particular user.

Response messages might be *provisional* or *final.* A provisional response indicates progress and does not terminate a SIP transaction. 1xx responses are provisional. Final responses terminate a SIP transaction. All 2xx, 3xx, 4xx, 5xx, and 6xx responses are final.
A request message is structured as: request-line, then headers, followed by a body. The *request-line,* the first line in the message, contains a method name, a request-URI, the protocol version separated by a single space (SP) character, and a Carriage Return Line Feed (CRLF). Listing 1 shows an example request (without a body):

Listing 1. INVITE request

```
INVITE sip:callee@143.145.52.13 SIP/2.0
Call-ID: 166106378111154808 4@143.145.52.134
Via: SIP/2.0/UDP 143.145.52.134:5070;branch=z9hG4bKC1C334860000F6D31DE9
Via: SIP/2.0/UDP app.ubiquity.net
From: sip:caller@ubiquity.net;tag=1738655730
To:sip:callee@143.145.52.13
CSeq:1 INVITE
contact:sip:143.145.52.134:5070
Accept:application/sdp
User-Agent:Ubiquity Third Party Call Control/-7671573430297227200
Max-Forwards: 70
Content-Length: 0
```

A response message is structured as: status-line, then headers, followed by body. The *status-line* consists of the protocol version followed by a numeric *status-code* and its associated textual phrase, with each element separated by a single SP character and terminated by a CRLF. Listing 2 shows the 100 TRYING response associated with the INVITE in Listing 1:

Listing 2. 100 TRYING response

```
SIP/2.0 100 Trying
Via: SIP/2.0/UDP 143.145.52.134:5070;branch=
```

```
z9hG4bKC1C334860000F6D31DE90,SIP/2.0/UDP app.ubiquity.net
From: sip:caller@ubiquity.net;tag=1738655730
To: sip:callee@143.145.52.13;tag=4E880060-97A
Date: Sun, 05 Mar 2000 23:01:39 GMT
Call-ID: 1661063781111548084@193.195.52.134
Server: Cisco-SIPGateway/IOS-12.x
CSeq: 1 INVITE
Allow-Events: telephone-event
Content-Length: 0
```

All SIP message headers include a field name followed by a field value. SIP headers can be categorized into five groups:

- Request headers
- Response headers
- General headers
- Entity headers
- User-defined headers

Furthermore, every SIP message requires several mandatory headers:

- **To:** A SIP address containing the request's destination.
- **From:** Indicates who has originated the request.
- **CSeq:** Contains a command sequence, which ensures that messages are dealt with in the order they were generated.
- **Call-ID:** The SIP proxy server uses the Call-ID header, a randomly generated string that uniquely identifies the session, to identify messages belonging to a session.
- **Via:** Contains information about what SIP devices the message has passed through as it moves between caller and callee. The Via header, moreover, routes a response in the reverse direction, through the same SIP devices as the request.
- **Contact:** Contains the actual location of the callee, which might be different from the address of the originator in the From header.

Other optional headers convey important information such as content type and content length.

A basic call flow

Table 2 illustrates an example SIP message flow that establishes a communication session between two UAs. A SIP proxy server is part of the network.

1	INVITE sent from ua1 to proxy
2	100 Trying response sent from proxy to ua1. The proxy will now forward the request

	downstream toward ua2.
3	INVITE sent from proxy to ua2.
4	100 Trying response sent from ua2 to proxy.
5	180 Ringing response sent from ua2 to proxy, which indicates that ua2 is alerting the user.
6	180 Ringing response sent from proxy to ua1
7	200 OK response sent from ua2 to proxy, which indicates that ua2 has accepted the call.
8	200 OK response sent from proxy to ua1.
9	ACK sent from ua1 directly to ua2, which represents the start of the session.
	...
	COMMUNICATION SESSION ESTABLISHED
	...
10	BYE sent from either UA to the other bypassing the proxy.
11	200 OK response from other party bypassing the proxy.

The example above shows only the simplest SIP call flow. Such call flows can quickly become more complex as the number of involved SIP entities grows. For example, if a single SIP address has two registered end-points, the proxy server will need to perform call forking.
A couple of other points should be noted. The original INVITE is known as an *initial request,* whereas the BYE request is known as a *subsequent request.* Also, the BYE request and any other subsequent requests pass directly between the two UAs rather than through the proxy. However, a proxy does have a method of ensuring it sees all subsequent requests as well.

Record routing

Typically, you'll employ a SIP proxy server only when you establish a communication session to perform registration lookups and forward the message. All subsequent requests then pass directly between caller and callee. A proxy, however, might want to see all subsequent SIP messages generated during the session. A user might want a proxy server to do this to generate billing records. To do so, the proxy server will need to see both the initial INVITE and terminating BYE requests. The proxy server will, in this case, enable record routing, which results in a Record-Route header being added to a request. Each proxy server that wants to remain on the signaling path will insert a Record-Route header into the initial INVITE as it passes from caller to callee.
The callee UA is obliged to retain the information contained within the Record-Route header and then return a response containing a copy of the Record-Route headers. The callee UA eventually receives the response and is obliged to retain the information contained within the Record-Route headers. At each UA, that information is stored as a

route set. The information in the route set adds Route headers to a subsequent request. A proxy server will then use the information contained within the Route headers to decide to which SIP device to send a request.

Advanced services

In addition to supporting basic functionality, a SIP server can present itself as a service creation platform -- letting a developer extend a SIP proxy server's basic functionality and create new applications and services.

Early attempts at providing service creation platforms resulted in proprietary application creation models and environments. Although meeting the requirements of the time, development communities often prefer a more open model. To address that problem, the SIP Servlet API has been proposed.

SIP servlets: Basic concepts

Conventional telephony applications are expensive to develop, both in terms of cost and time.

The SIP Servlet API, developed under the Java Community Process (JCP) as Java Specification Request (JSR) 116, provides a specification and a set of neutral interfaces that deliver a consistent, open platform on which to develop and deploy portable services. Because the API extends the HTTP Servlet model, they share common concepts such as the service method, a JAR-based file format, and deployment descriptors.

A standard SIP proxy server can act as the basis of a SIP application server. The *container,* a component of the application server (SIP A/S), provides the environment specified in the SIP Servlet API specification in which a SIP servlet can exist. The container loads and initializes a servlet and invokes the appropriate servlet methods when a SIP message arrives. When the SIP A/S stops, the container destroys each servlet. The container, therefore, performs many of the functions an HTTP servlet container performs.

A SIP servlet consists of a class that extends `javax.servlet.sip.Servlet` and implements the appropriate service method. The compiled class is then packaged in a Servlet Archive or SAR file together with a SIP deployment descriptor.

The XML-based deployment descriptor contains servlet configuration information and the message-matching rules, together with more general information such as the servlet name. The SAR file is deployed to the SIP application server.

The SIP Servlet API includes a set of objects and interfaces that provide high-level abstraction of many of the SIP concepts -- freeing you from worrying about SIP's fine details such as managing transactions. Some of the most important items are detailed in Table 3.

Class/Interface	**Description**
`SipServlet`	The base SIP servlet object that should be subclassed to create a SIP servlet. This class receives incoming messages through the service method, which calls `doRequest` or `doResponse` for incoming requests and responses, respectively. These two methods in turn dispatch a request method or status code to one of the following methods: • `doInvite`: For SIP INVITE requests • `doAck`: For SIP ACK requests • `doOptions`: For SIP OPTIONS requests • `doBye`: For SIP BYE requests • `doCancel`: For SIP CANCEL requests • `doRegister`: For SIP REGISTER requests • `doSubscribe`: For SIP SUBSCRIBE requests • `doNotify`: For SIP NOTIFY requests • `doMessage`: For SIP MESSAGE requests • `doInfo`: For SIP INFO requests • `doProvisionalResponse`: For SIP 1xx informational responses • `doSuccessResponse`: For SIP 2xx responses • `doRedirectResponse`: For SIP 3xx responses • `doErrorResponse`: For SIP 4xx, 5xx, and 6xx responses
`ServletConfig`	Used by a servlet container to pass configuration information to a servlet during initialization.
`ServletContext`	Used by a servlet to communicate with its servlet container; for example, to store attributes, dispatch requests, or write to a log file.
`SipServletMessage`	Defines common aspects of SIP requests and responses.
`SipServletRequest`	Provides high-level access to a SIP request message.
`SipServletResponse`	Provides high-level access to a SIP response message.
`SipFactory`	Factory interface for a variety of SIP Servlet API abstractions.
`SipAddress`	Represents SIP From and To header.
`SipSession`	Associates SIP messages belonging to the same SIP session. This is also known as a call log. Two messages belong to the same SipSession if they have identical Call-ID, From, and To headers as described in RFC 3261.
`SipApplicationSession`	Represents application instances, acts as a store for application data and provides access to contained protocol sessions.
`Proxy`	Represents the operation of proxying a SIP request and provides control over how that proxying is carried out; for example, record-routing and supervised mode.

While running, the SIP A/S receives various SIP messages. If the request is an initial request -- that is, a request that the container has no prior knowledge of -- the container then uses the message-matching rules contained in the deployment descriptor to determine whether to pass a message to a servlet. Depending on the rules, the servlet's service method is called. The default implementation of the service method attempts to identify the request type and call one of the doXXX methods, for example, `doInvite()`.

Processing a request involves either forwarding the message or returning a final response. In the former case, the appropriate response is received at some point in the future because a response always follows a reverse path to a request. Once a final response to an initial request has been received, an entity known as an *application path* is established. The application path ensures that any subsequent requests are routed correctly.

Create a simple servlet

To show how easy it is to work with the SIP servlet model, let's create a sample servlet. The servlet will listen for INVITE requests and always forward the INVITE on to the destination UA. Listing 3 shows a simple servlet:

Listing 3. Sample servlet

```
public class SampleServlet extends SipServlet
{
private static final String SERVLET_INFO = "SampleServlet,  "+
"1.0, Copyright Ubiquity Software Corporation, 2003"

  public void doInvite(SipServletRequest a_Request)
  throws ServletException, java.io.IOException
  {
   try
     {
     a_Request.send();
     }
   catch (IOException e)
     {
     throw new ServletException("Could not send request", e);
     }
  }


public String getServletInfo()
{
return "SampleServlet, 1.0, Copyright © Ubiquity, 2003";
}
}
```

Listing 4 shows the accompanying deployment descriptor. Note the rule matching that specifies this servlet should process INVITE requests only:

Listing 4. Deployment descriptor

```
<sip-app>
    <display-name>Sample Servlet</display-name>
    <servlet>
        <servlet-name>SampleServlet</servlet-name>
        <display-name>SampleServlet</display-name>
        <servlet-class>net.ubiquity.servlet.sample.SampleServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>SampleServlet</servlet-name>
        <pattern>
              <equal>
                <var>request.method</var>
                <value>INVITE</value>
              </equal>
        </pattern>
    </servlet-mapping>
</sip-app>
</code>
```

The simplest action a servlet can take is to send the message on without modification -- exactly what the sample servlet does. With only the sample servlet loaded, the request will be sent from the SIP A/S toward the specified callee UA. As stated earlier, any responses sent from the callee back to the caller will take the reverse path, although the SIP A/S will not present them to the servlet in this case.

Proxy behavior

As mentioned above, a servlet can simply send a message without modification. However, if the request must be sent downstream and requires further processing after the response is received, you'll need to create a proxy object. You'll also need a proxy object if a request must be sent to several locations, as happens with parallel or sequential forking. The proxy object also lets you specify that all subsequent requests pass to a servlet (the aforementioned record-routing process).

If a servlet must perform one or more of these activities, the `SipServletRequest`'s `getProxy()` method must be called to obtain a proxy object. You then call the proxy object's appropriate methods to obtain the desired behavior. For example, the code in Listing 5 proxies a request with record-routing enabled and in supervised mode:

Listing 5. Record routing

```
public void doInvite(SipServletRequest a_Request) throws ServletException,
IOException
{
    // Required by the Ubiquity Container to establish session state.
    SipApplicationSession appSession = a_Request.getApplicationSession();
    SipSession sipSession = a_Request.getSession();

    Proxy p = a_Request.getProxy(true);

    // Proxy in supervised mode.  This ensures that we see the response which will
    // be passed to us through the appropriate doResponse method.
    p.setSupervised(true);

    // If you want to see the ACK and the BYE, use RecordRouting.
    p.setRecordRoute(true);

    System.out.println("Proxying request: " + a_Request.getRequestURI());

    p.proxyTo(a_Request.getRequestURI());
}

public void doAck(SipServletRequest a_Request) throws ServletException, IOException
{
    System.out.println("ACK: " + a_Request);
}

public void doBye(SipServletRequest a_Request) throws ServletException, IOException
{
    System.out.println("BYE: " + a_Request);
}

public void doProvisionalResponse(SipServletResponse a_Response) throws
ServletException, IOException
{
    System.out.println("Provisional response: " + a_Response);
}

public void doSuccessResponse(SipServletResponse a_Response) throws
ServletException, IOException
{
    System.out.println("Success response: " + a_Response);
}
```

To reiterate, the above code will cause the servlet to receive any responses through the `doResponse()` method, which will normally send the response back to the originating UA. Any subsequent requests in the session will also invoke the servlet's `service()` method, for example the BYE request sent by either UA to terminate the session.

Originating and terminating behavior

At its simplest level, the SIP servlet can implement a terminating UA. In this case, a servlet simply performs some processing before returning a final response to the request, as seen in Listing 6:

Listing 6. Simple servlet

```
public void doInvite(SipServletRequest a_Request) throws ServletException,
IOException
{
    // Required by the Ubiquity Container to establish session state.
    SipApplicationSession appSession = a_Request.getApplicationSession();
    SipSession sipSession = a_Request.getSession();

    // Act as a terminating user agent and return a 403 Forbidden response to the
request.
    SipServletResponse response =
a_Request.createResponse(SipServletResponse.SC_FORBIDDEN);
    response.send();
}
```

The SIP Servlet API also lets a servlet act as an originating UA; that is, create initial and subsequent requests. A B2BUA would typically use such functionality. In this case, a servlet can use the `SipFactory` to create a new `SipAddress` and a new `SipServletRequest`, which can then be sent, as seen in Listing 7:

Listing 7. Creating a UAC request

```
public void doInvite(SipServletRequest a_Request) throws ServletException,
IOException
{
    // Required by the Ubiquity Container to establish session state.
    SipApplicationSession appSession = a_Request.getApplicationSession();
    SipSession sipSession = a_Request.getSession();

    // Create a new request and proxy it to a new destination.

    ServletContext sc = getServletContext();

    SipFactory factory =
(SipFactory)sc.getAttribute("javax.servlet.sip.SipFactory");

    Address to = factory.createAddress("sip:callee@company.net:5060");
    Address from = a_Request.getFrom();

    SipServletRequest newRequest = factory.createRequest(appSession, "INVITE", from,
to);

    Proxy p = a_Request.getProxy(true);

    p.setSupervised(true);
    p.setRecordRoute(true);

    System.out.println("Proxying request: " + a_Request.getRequestURI());

    p.proxyTo(newRequest.getRequestURI());
}
```

The SIP Servlet API does not currently let you create an initial request in response to a non-SIP event -- a problem highlighted by the need to obtain the `SipApplicationSession` from an existing request to create a new request. Until the SIP servlet community resolves this issue, SIP A/S vendors must implement their own functionality.

Acting as a B2BUA

A Back-to-Back User Agent acts as a middleman during a SIP call by receiving all requests and then forwarding them on to a callee. It also receives all responses before sending them back toward the caller. In essence, instead of one dialog between two UAs, there are two dialogs: one between the caller UA and the B2BUA, and the other between the B2BUA and the callee UA.

Not unsurprisingly, such behavior can break services. To minimize the risk, ensure all unknown headers are copied from an incoming request to the outgoing request. Another problem with B2BUAs centers on the session state they must keep to maintain the dialog. In the event of an SIP A/S failure, such session state will be lost and the call will no longer progress.

Despite these problems, a B2BUA proves useful in numerous scenarios, such as implementing billing applications, third-party call control, control of firewalls, and so on.

SIP: The future is actually now

This SIP introduction has only touched the surface of what SIP and the SIP Servlet API can achieve. Indeed, in the interest of brevity I skipped several important topics, including looping, spiraling, and servlet application composition.

This article, however, should make clear that SIP is a relatively simple protocol. And the SIP Servlet API provides a further level of abstraction that makes a developer's life even easier. Although the SIP Servlet model is still evolving, it does provide an excellent service creation environment. With SIP on the scene, creating telecommunications applications has never been easier.