SIP Servlet Specification, version 2.0

Final Release

JSR 359 Expert Group

Specification Lead:
Binod.P.G
Oracle Corporation

Please send comments to users@sipservlet-spec.java.net

Contents

Preface

This specification defines version 2.0 of the SIP Servlet API. The specification requires Java SE 6.0(or above) and support for SIP as defined in [RFC 3261].

The following IETF specifications referenced in this specification provide information relevant to the development and implementation of the SIP Servlet API and SIP Servlet containers: [RFC 6665], [RFC 3262], [RFC 2976], [RFC 3311], [RFC 3515], [RFC 3903], [RFC 3841], [RFC 3966], [RFC 3327], [RFC 3725], [RFC 3856], [RFC 4028], [RFC 3326], [RFC 4320], [RFC 3853], [RFC 4916], [RFC 5621], [RFC 5954], [RFC 5626], [RFC 5630], [RFC 5393], [RFC 6026]

Implementations of this specification should support the corrections to RFC 3261 specified in the RFCs above.

Online versions of these RFCs are available at http://www.ietf.org/rfc.

Who should read this Document

The intended audience for this specification includes the following groups:

- SIP application server vendors who want to provide servlet engines that conform to this standard.
- SIP servlet application developers.

Familiarity with SIP is assumed throughout.

Terminology

In this document, the key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and

"OPTIONAL" are to be interpreted as described in BCP 14, RFC 2119 [2] and indicate requirement levels for compliant JSR 359 implementations.

Typographic Conventions

Java classes, method names, parameters, literal values, and code fragments are printed in constant width as are XML elements and documents.

Providing Feedback

Feedback on this specification is welcome. Please e-mail your comments to users@sipservlet-spec.java.net

Acknowledgements for 2.0

This specification was developed under the Java Community Process 2.8 by JSR 359 expert group. The JSR 359 expert group consists of the following members:

Eric Cheung (AT&T)

Daniel Timoney(AT&T)

Nitzan Nissim (IBM)

Brian Pulito (IBM)

Tomas Ericson (Oracle)

Jean Deruelle(TeleStax, Inc.)

George Vagenas (TeleStax, Inc.)

Keith Lewis (Thrupoint)

Tom Strickland(Thrupoint)

Jonas Borjesson(Twilio Inc)

Wei Chen(Tropo Inc)

Subramanian Thamaraisamy (Cisco Systems)

Kristoffer Gronowski (Ericsson AB)

Rajnish Jain (IPC Systems)

Thomas Leseney(Nexcom Systems)

David Ferry (Open Cloud Limited)

We would like to thank the JSR 236 expert group for their contribution in the concurrency section of the specification.

Additionally, Cai Xia Liu, Kai Ellinger, Anurag Bahl, Lars Forsberg, Pu Xuan Song, Li Li, Fusheng Wang, David Gao, Amitha Puliajala, Changtai Liu and Wei Gan Ji (Oracle) made enormous contribution to the specification.

We would also like to thank Amanda Mills of Oracle for providing editorial changes to the specification.

Acknowledgements for 1.1

This specification was developed under the Java Community Process 2.6 as JSR 289. The JSR 289 expert group consists of the following members:

Hira Agrawal (Telcordia Technologies Inc.),

Bartosz Baranowski (Red Hat Middleware LLC),

Diego Besposvan (Mailvision LTD),

Chris Boulton (Avaya, Inc),

Matthias Britsch (T-Mobile International AG & Co. KG),

Erik Burckart (IBM),

Eric Cheung (AT&T),

David Connelly,

Christophe Dejouhanet (Orange France SA),

Jean Deruelle (Red Hat Middleware LLC),

Simon Dutkowski (Fraunhofer-Gesellschaft Institute FIRST),

Sreeram Duvur (Sun Microsystems, Inc.),

Tomas Ericson (Oracle),

Jarek Wilkiewicz (Hewlett-Packard),

Goran Eriksson (Ericsson AB),

Ian Evans (Avaya, Inc),

Anat Fradin (IBM),

Kristoffer Gronowski (Ericsson AB),

Robert Handl (Ericsson AB),

Avshalom Houri (IBM),

Ivelin Ivanov (Red Hat Middleware LLC),

Akinori Iwakawa (Fujitsu Limited),

Hakan Jonsson (Appium Technologies AB),

Nasir Khan,

Yueh Lee (Cisco Systems),

Thomas Leseney (Nexcom Systems),

Johan Liseborn (Oracle),

Stephane H. Maes (Oracle),

Roger N. Mahler (Cingular Wireless),

Gines Gomez Martinez (Voztelecom Sistemas S.L.),

Remy Maucherat (Apache Software Foundation),

Francesco Moggia (Telecom Italia),

Phelim O'Doherty (Oracle),

Amir Perlman (IBM),

Marc Petit-Huguenin (8x8),

Per Pettersson (Ericsson AB),

Jean-Pierre Pinal (Nexcom Systems),

Kermit Hal Purdy (AT&T),

Uri Segev (IBM),

James Steadman (Oracle),

Prasad Subramanian (Sun Microsystems, Inc.),

Atul Varshneya,

Lebing Xie (Fraunhofer-Gesellschaft Institute FIRST),

Sami Samandi (Netcentrex)

Damian O’Neill (Aepona Technologies AB)

We would like to thank Anders Kristensen (Cisco Systems Inc.) and the members of the JSR 116 Expert Group who developed the SIP Servlet Specification v1.0 that contributed immensely to the development of the v1.1 specification. This specification builds on the SIP Servlet Specification v1.0 and, for this reason, this specification copies text from the v1.0 specification in many places [SIP Servlet API].

Additionally, Rhys Ulerich of IBM, Paul Devine, Michael Palmeter, Vinod Mehra, and Anno Langen of BEA Systems made enormous contributions to this specification. We would like to thank Paul Sydell of Oracle for his help with Java EE schema alignment, Adrian Kalaveshi of Oracle for his help with administrating the collaborative site used by the JSR 289 expert group, and David Yozie of Oracle for his help in drafting this specification as a PDF.

An important goal of the SIP Servlet API is to build on the work of the well-established (HTTP) Servlet API. For this reason, this specification copies text from the Java Servlet Specification in many places [Servlet API]. The glossary includes a large number of entries copied from the SIP specification [RFC 3261].

This specification draws and adapts the principles of application independence, composition, and selection from Distributed Feature Composition [DFC1998], [CN2004], [SE2005]. This specification makes use of Java annotations introduced in J2SE 5.0 and re-uses common annotations for the Java platform [JSR250].

1 Overview

Session Initiation Protocol (SIP) is used to establish, modify, and tear down IP multimedia sessions, such as IP telephony, presence, and instant messaging sessions. An important aspect of any communication infrastructure is programmability and the purpose of the SIP Servlet API standardizes the platform for delivering SIP-based services. The term platform includes the Java API itself as well as standards covering the packaging and deployment of applications.

1.1 Goals of the SIP Servlet API

The following summarizes some important properties of the SIP Servlet API:

- SIP signaling: It allows applications to perform a fairly complete set of SIP signaling actions, including support for acting as user agent client (UAC), user agent server (UAS), and proxy.
- Simplicity: Containers handle nonessential complexity, such as managing network listen points, retransmissions, CSeq, Call-ID and Via headers, routes, etc.
- Converged applications: It is possible for containers to support converged applications, that is, applications that span multiple protocols and interfaces, such as Web, telephony, and other Java EE interfaces.
- Third party application development: The servlet model supports third party application development. An XML deployment descriptor is used to communicate application information from the application developer to deployers.
- Application composition: It is possible for several applications to execute on the same incoming or outgoing request or response. Each application has its own set of rules and executes independently of other applications in a well-defined and orderly fashion.

- Carrier grade: Servlets store application data in container managed session objects. Implementations may persist and/or replicate this data to achieve high availability.

1.2 What is a SIP Servlet?

A SIP servlet is a Java-based application component that is managed by a SIP servlet container and that performs SIP signaling. Like other Java-based components, servlets are platform independent Java classes that are compiled to platform neutral bytecode that can be loaded dynamically into and run by a Java-enabled SIP application server. Containers, sometimes called servlet engines, are server extensions that provide servlet functionality. Servlets interact with SIP clients by exchanging request and response messages through the servlet container.

1.3 What is a SIP Servlet Container?

The servlet container is a part of an application server that provides the network services over which requests and responses are received and sent. It decides which applications to invoke and in what order. A servlet container also contains and manages servlets through their lifecycle.

A servlet container can be built into a host SIP server, or installed as an add-on component to a SIP Server through that server's native extension API. Servlet containers can also be built into or installed on servlet-enabled application servers.

A SIP servlet container manages the network listen points on which it listens for incoming SIP traffic. (A listen point being a combination of transport protocol, IP address, and port number). The SIP specification requires all SIP elements to support both UDP and TCP, and, optionally, TLS, SCTP, and potentially other transports.

A servlet container may place security restrictions on the environment in which a servlet executes. In a Java Standard Edition (Java SE) or Java™ Platform, Enterprise Edition (Java EE) environment, these restrictions should be placed using the permission architecture defined by the Java Platform. For example, high-end application servers may limit the creation of a Thread object to ensure that other container components are not negatively impacted.

1.4 Relationship with the HTTP Servlet API

The Java Servlet API is defined in the *Java Servlet Specification* [Servlet API]. It consists of a generic part defined in package `javax.servlet` and an HTTP-specific part defined in package `javax.servlet.http`. This specification refers to the generic part by using the unqualified term *Servlet API* and to the HTTP-specific API as the *HTTP Servlet API*. The SIP Servlet API builds on the generic servlet API in much the same way as the HTTP Servlet API does, and is defined as package `javax.servlet.sip`. Therefore, a SIP servlet container MUST support the packages `javax.servlet` and `javax.servlet.sip`.

This specification is structured along the lines of the *Java Servlet Specification*, and like it, includes text not specific to the SIP Servlet API. Parts of this text has been copied from that other document, albeit modified to reflect the non-HTTP nature of SIP servlets, for example to use the broader term *servlet application* instead of *web application* when the context applies equally to SIP and HTTP applications.

1.4.1 Differences from the HTTP Servlet API

SIP was to some extent derived from HTTP, so the two protocols have much in common. Both are request-response protocols and both of their messages have a similar structure and format. However, in terms of providing services, there are important differences between the two protocols:

- HTTP services (including HTTP servlet applications) are almost exclusively hosted on HTTP origin servers. That is, they are hosted on the Web server generating the final response to requests (as opposed to a proxy server). In contrast, an important function of SIP applications is intelligent request routing and the ability to act as a proxy.
- HTTP is not a peer-to-peer protocol as is SIP and web applications never originate requests. SIP applications, on the other hand, need the ability to initiate requests of their own. An application that accepts an incoming call may have to terminate it by subsequently sending a BYE request towards the caller, a wakeup-call application may have to send the initial INVITE establishing a dialog, and a presence server application needs to be able to initiate NOTIFY requests. A back-to-back user agent (B2BUA) is a type of network based application that achieves a level of control over calls not attainable through proxying, and that requires client functionality, also. These examples demonstrate why client-side functionality is necessarily part of a SIP service infrastructure and explains the presence of such functionality in the SIP Servlet API.

Therefore, in addition to the ability inherited from the Servlet API of responding to incoming requests, the SIP Servlet API MUST also support the following capabilities:

- generate multiple responses (for example, one or more 1xx followed by a final response)
- proxy requests, possibly to multiple destinations
- initiate requests
- receive responses as well as requests

1.4.1.1 Use of Servlet.service()

To support these capabilities, the SIP Servlet API uses the original Servlet API interfaces differently than the HTTP Servlet API. SIP servlet applications are invoked when events occur in which they have registered an interest. These events can be either incoming requests or responses, and they are delivered to applications through the service method of the `javax.servlet.Servlet` interface:

```
void service(ServletRequest request, ServletResponse response);
```

This is the application entry point used by the HTTP Servlet API also, but it is used slightly differently here. When used to process SIP traffic, only one of the request and response objects is non-null. When invoked to handle incoming requests, the response argument will be null and vice versa, when invoked to handle incoming responses the request argument will be null.

Note: This caters for the fact that there is not necessarily a one-to-one correspondence between requests and responses in SIP applications.

1.4.1.2 Asynchronicity

The SIP Servlet API event model is also much more asynchronous than the largely synchronous HTTP API. This means that applications are not obliged to respond to incoming requests in the upcall reporting of the event. Instead, the applications may initiate some other action, return control to the container, and then respond to the request sometime later. The container relies on the timeout of the application instance to guarantee that resources are always recovered eventually, even if an application fails to respond to the request in a timely manner.

Asynchronicity simplifies the programming of event driven services and allows an application such as a B2BUAto not hog threads while waiting for a potentially long-lived transaction to complete.

1.4.1.3 Application Composition

Although the model of the HTTP Servlet API is to select a single application to process each incoming request, it is often desirable to apply multiple services to incoming SIP requests. This specification describes an application composition model that defines the conditions that a SIP servlet container must ensure are satisfied when it chooses to invoke multiple applications. This model is discussed in detail in chapter 18 Application Selection And Composition Model.

1.4.2 Converged Applications

Although the SIP Servlet API can be implemented independently of the HTTP Servlet API, many services may combine multiple modes of communication. For example, services may combine telephony, Web, email, instant messaging, and other server-side components like Web services and other Java EE interfaces. A converged SIP container enables the deployment of applications that use SIP, HTTP Servlet API, and other Java EE components like EJBs, Web services, and messaging.

1.5 Deployment Models

This section describes three major deployment models where the SIP Servlet container could be used. Each model has a use in different real life scenarios, and the SIP Servlet container implementations should allow the container to operate in all three deployment models.

1.5.1 Standalone SIP Servlet Container

A standalone SIP Servlet container provides only a SIP interface and hosts only SIP Servlets as its applications. This could be a lightweight container that is used in situations where the application only has a SIP interface. It could also be used as an embeddable SIP Servlet container in a Java application.

1.5.2 SIP and HTTP Converged Servlet Container

SIP Servlets and HTTP Servlets are both derived from the same base Servlet API. In a converged SIP and HTTP application, the SIP and HTTP Servlets share the same context. A combined SIP/HTTP servlet application contains deployment descriptors for both the SIP and HTTP parts and shares state through application session objects that represent application instances.(see 16 Converged Container and Applications). For this reason, it is important that it be possible to implement a servlet container that supports both the SIP and HTTP Servlet APIs simultaneously and in a manner that allows applications to include both SIP and HTTP components.

1.5.3 SIP and Java EE Convergence

A SIP Servlet Container is usually part of a larger application server providing not just a HTTP Servlet container but also other Java EE applications like EJBs and Web services. This specification facilitates the use of SIP Servlet technology in conjunction with the larger Java EE deployment model. This specification includes several features that allow the deployment of an application that has Java EE, HTTP and SIP components and that have seamless access from one context to the other. This functionality enables development of very powerful applications.

The following examples are only reference models of deployment. To be compliant with this specification, the container MUST implement all minimum features as specified in this specification.

1.6 Examples

1.6.1 A Location Service

Routing is an integral part of SIP and is a common function of SIP services. In this example, a Location Service SIP application performs a database lookup on the request URI of incoming requests and proxies the request to a set of destination addresses associated with that URI. The application and container perform the following steps:

1. The caller (Alice) makes a call to sip:bob@example.com. The servlet container receives the INVITE and then invokes the Location Service.
2. The Location Service determines, using non-SIP means, that the callee (Bob) is registered with two locations identified by two SIP URIs.
3. The Location Service proxies to those two destinations in parallel, without record-routing and in unsupervised mode. One of the destinations returns a 200 (OK) response and the other branch is canceled by the container.
4. The 200 response is forwarded upstream to Alice and the call setup is completed as usual.

In this example, the application (and the host application server) is involved only in establishing the SIP dialog and is not involved in subsequent signaling within that dialog.

1.6.2 A Multi-Protocol Servlet Application

In this example, a converged application consists of both SIP and HTTP components. SIP is used to establish media sessions and as a presence subscription and notification protocol [simple].

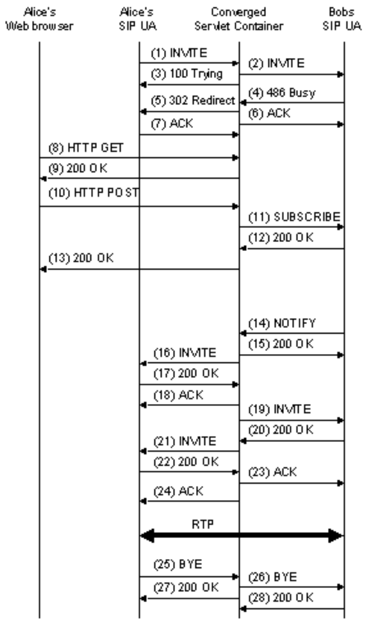
The service is called Call Schedule on Busy or No Answer (CSBNA). As in the previous example, it resides on a network server situated on the path between a caller and a callee. When a call setup attempt fails because the callee is busy or not available, the application returns a URL pointing to a Web page.The Web page prompts the caller to choose whether a phone call is set up the moment the callee becomes available. The application knows the callee’s availability by subscribing to his or her presence status. When the application receives a notification that the callee is available, it establishes a session between the original caller and callee using third party call control techniques [3pcc].

Figure 1-1 illustrates the call flow. The steps are as follows:

1. Alice makes a call to Bob. The INVITE is routed to the converged servlet container.
2. The CSBNA SIP servlet acts a B2BUA and sends an INVITE to Bob’s SIP phone.

3. The converged servlet container also sends a 100 (Trying) informational response upstream to stop retransmissions.
4. Bob’s phone returns a 486 (Busy Here) response as Bob is currently in a phone call.
5. The CSBNA servlet returns a 302 (Temporarily Unavailable) response containing a single Contact with an HTTP URL pointing back to an HTTP servlet that is part of the CSBNA application.
6. The SIP application server sends an ACK for the 486 error response to Bob's phone as per the SIP specification.
7. Alice’s SIP UA sends an ACK for the 302 error response to the application server.
8. Alice’s SIP UA launches a Web browser to retrieve the HTTP Contact URL from the 302 response.
9. The HTTP CSBNA servlet returns a Web page that asks Alice whether to have a call automatically setup based on Bob’s availability. The form is pre-populated by the servlet with all required information, in particular Alice and Bob’s SIP addresses.
10. Alice clicks the Submit button on the Web page, causing an HTTP request to be sent to the application server.
11. The service sends a SIP SUBSCRIBE request to Bob’s SIP UA. This allows the service to know when Bob becomes available again.
12. Bob’s UA accepts the subscription.
13. The HTTP CSBNA servlet returns a Web page to Alice which she can subsequently use to modify or cancel the scheduled call.
14. When Bob’s phone becomes available, his UA sends a notification of his new availability status to the CSBNA subscriber servlet in a SIP NOTIFY request.
15. The CSBNA application responds to the NOTIFY. (The application unsubscribes from Bob’s status at this point—this is not shown in the diagram.)

Figure 1-1 Call Flow for the CSBNA Application



In steps 16 through 24, the application establishes a call between Alice and Bob using the third party call control mechanisms described in [3pcc]. In this case, a call is established and media is exchanged over RTP.

In steps 25 through 28, the call is terminated.

In addition to illustrating the functioning of a converged container, this example also demonstrates the need for the concept of application sessions. Over the lifetime of an instance of the CSBNA application, some 5 point-to-point SIP signaling relationships are created. Each of these correspond to a SipSession instance. Additionally, one HttpSession is created—this also corresponds to a point-to-point "signaling" relationship between the container and the HTTP client. Obviously, all of these protocol sessions are related and need to share state. The application session represents an instance of the application and binds together the protocol sessions representing individual signaling relationships and allows for sharing of state between them. The session model is further discussed in chapter 8 Sessions.

1.6.3 A Java EE Converged Application

The following example demonstrates SIP and Java EE application convergence. It shows how a SIP UA (User-Agent) is notified of UA Profile changes for which it had subscribed previously. (The general mechanism of UA Profile propagation is described in the draft http://tools.ietf.org/wg/sipping/draft-ietf-sipping-config-framework/.)

This example uses a SIP UA that could be a SIP capable device like a SIP phone or SIP set-top box. This SIP UA is served by the SIP application server which is a SIP servlet container. The interaction uses the SIP Event Notification mechanism defined in [RFC 6665]. The SIP UA has its configuration data, also known as UA profile data which is maintained somewhere on the network. The SIP UA uses a converged application deployed on the SIP Servlet Container to access its profile data. Since the data can also be changed by mechanisms other than SIP (out of band mechanisms) and the SIP UA can further subscribe (using SIP SUBSCRIBE) for these changes, the converged application can notify the SIP UA of the profile changes (using SIP NOTIFY).

As the data is supposed to be stored outside of the application context, the application knows about these out of band profile data changes via JMS messages from an abstract profile database. (Where the profile actually resides is beyond the scope of this example).

This example is an enterprise application consisting of a SIP servlet application and an EJB. The EJB is actually a Message Driven Bean (MDB).

The SIP Servlet handles SUBSCRIBE and responses to the NOTIFY messages from the external SIP UA.

This application uses the `javax.servlet.sip.SipSessionsUtil` utility class described in 16 Converged Container and Applications. The `SipSessionsUtil` utility is a container provided lookup interface which can be used to access the reference to a particular `SipApplicationSession` given its ID.

The steps involved in the working of this example are as follows:

1. The SIP Servlet application using the annotation @EJB or ejb-ref element in the deployment descriptor gets the reference to the EJB.

2. The EJB gets the reference to the `SipSessionsUtil` utility through an annotation defined in 22.3.7 Annotation for SipSessionsUtil Injection.

3. The SIP UA sends a SIP SUBSCRIBE to the application. In the `doSubscribe()` processing, the `SipServlet` stores the sip-application-session-id and sip-session-id using a business method defined on the EJB. (This is the process of installing the subscription with the bean).

4. The SIP Servlet sends a 200 OK to the SUBSCRIBE and a NOTIFY according to the SIP procedures defined in [RFC 6665].
5. At some later time the user profile changes. The UA profile is managed by an abstract database which sends a JMS message to the application server whenever the profile is modified. The EJB (which is actually a MDB) in the application gets a JMS message on its queue. The message contains an identifier which tells us which UA this profile update is for.
6. The EJB maintains a map of UAs and the current sip-application-session-id. On getting the JMS message for a certain UA, it retrieves the sip-application-session-id from its map.
7. The MDB, which has access to the SipSessionsUtil utility, retrieves the `SipApplicationSession` reference using the sip-application-session-id.
8. Since the MDB also knows the `SipSession` id (previously set by the `SipServlet` and stored alongside the sip-application-session-id), it gets to the right SIP dialog from the `SipApplicationSession`.
9. Using normal in-session request creating mechanism the bean creates a new SIP NOTIFY request and sends the profile changes as payload on the dialog to the SIP UA
10. The SIP UA gets this SIP NOTIFY with profile data and sends a 200 OK. This 200 OK gets delivered to the SIP `Servlet` and the process completes.

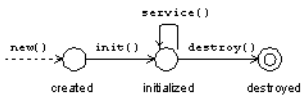
2 The Servlet Interface

The `Servlet` interface is the central abstraction of the Servlet API and hence of the SIP Servlet API. All servlets implement this interface either directly or, more commonly, by extending a class that implements the interface. The generic Servlet API defines a class `GenericServlet` that is an implementation of the `Servlet` interface. The SIP Servlet API defines a class `SipServlet` that extends the `GenericServlet` interface and performs dispatching based on the type of message received. For most purposes, developers extend `SipServlet` to implement their servlets.

2.1 Servlet Life Cycle

SIP servlets follow the lifecycle of servlets defined in [Servlet API, section 2.3]. All provisions in that section apply directly to SIP servlets. The lifecycle is illustrated in Figure 2-1 Servlet lifecycle.

Figure 2-1 Servlet lifecycle.



The servlet container loads the servlet class, instantiates it, and invokes the `init()` method on it, passing along configuration information in the form of a `ServletConfig` object. After it initializes successfully, the servlet is available for service. The container repeatedly invokes the service by calling the `service()` method with arguments representing incoming requests and responses. When the container decides to deactivate the servlet instance, it invokes the `destroy()` method – the servlet frees up resources allocated in `init()` and becomes garbage collected.

The failure to initialize any of the servlet within the application MUST be taken as a failure of the entire application and the application MUST be taken out of service. If the failed servlet was a `load-on-startup` servlet, the Application Router MUST NOT be notified of this application's deployment using `SipApplicationRouter.applicationDeployed()`. The Application Router component is described in detail in 18 Application Selection And Composition Model. If the servlet is not a `load-on-startup` servlet, but the initialization of it failed, the Application Router MUST be informed of this application's undeployment using `SipApplicationRouter.applicationUndeployed()`.

2.1.1 Servlet Life Cycle Listener

The container calls the Servlet `init()` method to allow the Servlet to initialize one time setup tasks that are not specific to any application instance. The Servlet is not usable until the `init()` method successfully returns. In addition to receiving requests, a SIP Servlet can also initiate requests (acting as a UAC) or create timers by using the `TimerService` interface. Until the initialization of the Servlet is complete, the SIP Servlet SHOULD NOT perform any of the signaling related tasks including sending SIP messages or setting timers. A listener for SIP Servlet lifecycle, if present, MUST be invoked by the container. The listener defines a single method indicating that the initialization of the Servlet is now complete and it can now receive messages as well as perform any other tasks.

```
void servletInitialized(javax.servlet.ServletContextEvent event);
```

The following initialization sequence MUST be followed for a `load-on-startup` servlet:

1. Deploy the application.
2. Invoke `Servlet.init(),` the initialization method on the servlet. Invoke the `init()` method on all of the `load-on-startup` servlets in the application.
3. Invoke `SipApplicationRouter.applicationDeployed()` for this application.
4. If present, invoke `SipServletListener.servletInitialized()` on each of initialized servlet's listeners.

For applications without any servlets declared as `load-on-startup`, the `SipApplicationRouter.applicationDeployed()` MUST be invoked immediately after the deployment completes successfully. The `init()` method on these servlets and `servletInitialized()` callback methods on their listener's are called just before getting the first request.

2.2 Processing SIP Messages

The basic `Servlet` interface defines a `service` method for handling client requests. This method is called for each message that the servlet container routes to an instance of a servlet. The handling of concurrent messages in a servlet application generally requires the developer to design servlets that can deal with multiple threads executing within the `service` method at a particular time. Generally, the servlet container handles concurrent requests to the same servlet by concurrent execution of the `service` method on different threads.

SIP servlets are invoked to handle both incoming requests and responses. In either case, the message is delivered through the `service` method of the `Servlet` interface:

```
void service(ServletRequest req, ServletResponse res)
   throws ServletException, java.io.IOException;
```

- For request messages, the response argument MUST be null.
- For response messages, the request argument MUST be null.
- When invoked to process a SIP event, the arguments must implement the `SipServletRequest` or `SipServletResponse` interfaces.

The `SipServlet` implementation of the `service` method dispatches incoming messages to methods `doRequest` and `doResponse` for requests and responses, respectively:

```
protected void doRequest(SipServletRequest req);
protected void doResponse(SipServletResponse resp);
```

These methods then dispatch further as described in the following sections.

Note: Section 22.3.2 @SipServlet Annotation describes an alternative mechanism of declaring the Servlet class by using annotations.

2.3 SIP Specific Request Handling Methods

The `SipServlet` abstract subclass defines a number of methods beyond what is available in the basic `Servlet` interface. These methods are automatically called by the `doRequest` method (and indirectly from the service) in the `SipServlet` class to aid in processing SIP-based requests. These methods are:

- `doInvite` for handling SIP INVITE requests
- `doAck` for handling SIP ACK requests

- `doOptions` for handling SIP OPTIONS requests
- `doBye` for handling SIP BYE requests
- `doCancel` for handling SIP CANCEL requests
- `doRegister` for handling SIP REGISTER requests
- `doPrack` for handling SIP PRACK requests
- `doSubscribe` for handling SIP SUBSCRIBE requests
- `doNotify` for handling SIP NOTIFY requests
- `doMessage` for handling SIP MESSAGE requests
- `doInfo` for handling SIP INFO requests
- `doUpdate` for handling SIP UPDATE requests
- `doRefer` for handling SIP REFER requests
- `doPublish` for handling SIP PUBLISH requests

The first six Java methods correspond to request methods defined in the baseline SIP specification [RFC 3261]. The following methods correspond to request methods defined in various SIP extensions:

- PRACK is defined in [RFC 3262] and is discussed in 6.8.1 Reliable Provisional Responses.
- SUBSCRIBE and NOTIFY are defined in the SIP event notification framework [RFC 6665] upon which the SIP presence framework is defined [simple].
- MESSAGE supports instant messaging [IM].
- INFO is a general-purpose mid-dialog signaling transport mechanism [RFC 2976]
- UPDATE updates SIP session parameters without affecting dialog state [RFC 3311].
- REFER transfers SIP calls [RFC 3515].
- PUBLISH, introduced in [RFC 3903], publishes SIP specific event state.

The `SipServlet` implementation of these methods is as follows. The `doAck` and `doCancel` methods do nothing. All other methods check whether the request is an initial request, as described in 12.2.9 Handling Subsequent Requests. If it is an initial request, the request is rejected with status code 501. Otherwise, the method does nothing (if the application proxied the initial request, the container proxies the subsequent request when the method call returns). A servlet typically overrides only those methods that are relevant for the service it is providing.

Note: The handling of incoming requests is asynchronous in the sense that servlets are not required to fully process incoming requests in the container's invocation of the service method. The request may be stored in a `SipSession` or `SipApplicationSession` object to be retrieved and responded to later—typically triggered by some other event. The container does not generate its own response if a servlet returns control to the container without having responded to an incoming request.

To handle SIP methods that are unknown to `SipServlet.doRequest`, an application can override the `doRequest` method and invoke super as follows:

```
protected void doRequest(SipServletRequest request)
      throws ServletException, IOException
  {
      if ("STORE".equals(request.getMethod())) {
          doStore(request);
      } else {
          super.doRequest(request);
      }
  }
```

2.4 Receiving Requests

Containers invoke SIP servlets to process incoming requests in the following cases:

- Initial requests.The container triggers an application based on the application selection process described in 18 Application Selection And Composition Model.
- Subsequent requests in a dialog for UA applications or in which the application is a proxy and is record-routed on the initial request.
- An ACK for a 2xx response to an INVITE which the application either responded to as a UAS or proxied with record-routing enabled.
- A CANCEL for an INVITE the application received but has not yet generated a final response for.

In all cases, the container invokes the servlet through the `Servlet.service` method with a `SipServletRequest` object and `null` for the response argument.

If a servlet throws an exception when invoked to process a request other than ACK and CANCEL, the servlet container MUST generate a 500 response to that request. The response's header or body may contain additional information for identifying the cause of the problem.

2.5 SIP Specific Response Handling Methods

The `doResponse` method dispatches to one of the following methods based on the class of the status code of the response:

- `doProvisionalResponse` for handling SIP 1xx informational responses other than 100
- `doSuccessResponse` for handling SIP 2xx responses
- `doRedirectResponse` for handling SIP 3xx responses
- `doErrorResponse` for handling SIP 4xx, 5xx, and 6xx responses

Chapters 10 and 11 describe, in detail, the rules for invoking these methods.

2.6 Number of Instances

Refer to [Servlet API, section 2.2].

In this specification, number of servlet instances are different from application instances.Application instance refers to an application session together with its contained protocol sessions as described in chapter 8 Sessions. Servlet instances are independent of any application instance and typically process requests belonging to many different application instances.

3 Servlet Context

`ServletContext` defines a servlet's view of the SIP application within which the servlet is running. Chapter 3 of the Java Servlet Specification describes `ServletContext` [Servlet API], and it also applies to the SIP servlet API. This chapter addresses issues specific to the SIP Servlet API.

3.1 SipServletContext

A SIP servlet container MUST make `SipServletContext`, an extension of the `ServletContext` instance, available to all applications. Using the `SipServletContext` object application can retrieve SIP specific utility objects, configuration etc. Applications may obtain the `SipServletContext` object from the container by using methods such as `SipSession.getServletContext()`, `SipServlet.getServletContext`(), and `ServletContextEvent.getServletContext()` or by using dependency injection as specified in 22.5.2 CDI Built-In Beans.

3.2 The SipFactory

Servlets use the `SipFactory` interface to create instances of various interfaces:

- **Requests**: the `createRequest` method creates instances of the `SipServletRequest` interface and is used by UAC applications when creating new requests that do not belong to existing `SipSession`s. To create subsequent requests in an existing dialog, `SipSession.createRequest` is used instead. 13.1.1 Creating Initial Requests discusses requirements of the `createRequest` methods.
- **Address objects**: ability to create `URI`, `SipURI`, `Address` and `Parameterable` instances.
- **Application sessions**: ability to create new application sessions.

All servlet containers MUST make an instance of the `javax.servlet.sip.SipFactory` interface available to servlets via the `SipServletContext.getSipFactory()` method and the context attribute `javax.servlet.sip.SipFactory`.

With this specification, the `SipFactory` instance can be injected by using the Java Metadata annotations in applications in addition to context lookup. Additionally, the `SipFactory` instance can be injected into Java EE applications that do not have access to `ServletContext`. Containers compliant with this specification MUST make the `SipFactory` instance available via this annotation as described in 16.2 Accessing SIP Factory.

3.3 Extensions Supported

SIP servlet containers MUST make an immutable instance of the `java.util.List` interface available by using the method `SipServletContext.getSupported()` and as a `ServletContext` parameter with the name `javax.servlet.sip.supported`. `List` contains the option tags of the SIP extensions registered with IANA, as supported by the container. Applications may use the List to determine whether the container supports a particular extension. For an example, see 6.8.1 Reliable Provisional Responses.

3.4 RFCs Supported

SIP servlet containers MUST make an immutable instance of the `java.util.List` interface available by using the method `SipServletContext.getSupportedRfcs()` and as a `ServletContext` parameter with the name `javax.servlet.sip.supportedRfcs`. `List` contains the RFC numbers of the SIP RFCs supported by the container. Applications can use the `List` to determine whether the container supports a particular RFC.The application can then adapt the call-flow or fail the application's deployment if a RFC which it relies on is not supported by the container. This context parameter helps in portability of applications across containers.

3.5 Context Path

The Servlet API defines the notion of a *context path*. This is an HTTP URL path prefix with which a web application is associated. All requests that include the context path of a web application are routed to the corresponding servlet context. Since SIP URIs do not have a notion of paths, the following `ServletContext` methods do not apply to SIP-only servlet applications/containers and must return null:

```
ServletContext getContext(String uripath);
String getRealPath(String path);
```

```
  RequestDispatcher getRequestDispatcher(String path);
```

As far as resource loading is concerned, the context path of a SIP-only servlet is always “/”. For a combined HTTP and SIP application executing in an HTTP Servlet capable container, the context path is defined by the HTTP Servlet API. Resource loading proceeds according to the Java Servlet Specification [Servlet API].

3.6 SipServletContext Methods and Context Parameters

Containers compliant with this specification MUST make the following references available to the applications by using `SipServletContext` methods and as `ServletContext` parameters.

Table 3-1 Context Parameters and Getter Methods

Parameter Name	Method	Description
`javax.servlet.sip.supported`	`getSupported()`	An immutable instance of the `java.util.List` interface containing the String names of SIP extensions supported by the container.
`javax.servlet.sip.supportedRfcs`	`getSupportedRfcs()`	An immutable instance of the `java.util.List` interface containing the RFC numbers represented as strings of SIP RFCs supported by the container.
`javax.servlet.sip.100rel`		Parameter whose value states whether the container supports the 100rel extension (i.e. RFC 3262). This parameter has been deprecated in this specification in favor of the `javax.servlet.sip.supported` parameter.
`javax.servlet.sip.SipSessionsUtil`	`getSipSessionsUtil()`	A container class `SipSessionsUtil` for looking up `SipApplicationSession` instances.
`javax.servlet.sip.SipFactory`	`getSipFactory()`	Instance of the applications `SipFactory`.
`javax.servlet.sip.outboundInterfaces`	`getOutboundInterfaces()`	An immutable instance of the `java.util.List` interface containing the `SipURI` representation of IP addresses that are used by the container to send out messages.

Table 3-1 Context Parameters and Getter Methods

Parameter Name	Method	Description
`javax.servlet.sip.outboundAddresses`	`getOutboundAddresses()`	An immutable instance of the `java.util.List` interface containing the `InetAddress` representation of IP addresses that are used by the container to send out the messages.
`javax.servlet.sip.TimerService`	`getTimerService()`	Instance of the `TimerService` class.
`javax.servlet.sip.DnsResolver`	`getDnsResolver()`	Instance of the `DnsResolver` class.

3.7 Programmatically adding and configuring SIP Servlets

Leveraging the work done in the servlet specification, this specification allows Servlets (and SIP servlet POJOs) and Listeners to be added programmatically. The specification also allows the application and Servlets (and SIP servlet POJOs) to be configured programmatically. These methods can be called only during the initialization of the application either from the `contexInitialized` method of a `ServletContextListener` implementation or from the `onStartup` method of a `ServletContainerInitializer` implementation. `ServletContext` and `SipServletContext` interfaces are added with the new methods. Usage of `ServletContainerInitializer` may further be filtered using `@HandlesTypes` annotation. More details on the interfaces and methods in this section can be found in the *Java Servlet Specification*.

3.7.1 Servlets

SIP servlets may be added programmatically by using the following methods.

- `addServlet(String servletName, String className)`
- `addServlet(String servletName, Servlet servlet)`
- `addServlet(String servletName, Class <? extends Servlet> servletClass)`
- `<T extends Servlet> T createServlet(Class<T> clazz)`

Note that the `createServlet` method supports all annotations applicable to the SIP servlets except `@SipServlet`.Applications can retrieve existing `ServletRegistrations` by using the following methods.

- `ServletRegistration getServletRegistration(String servletName)`
- `Map<String, ? extends ServletRegistration> getServletRegistrations()`

3.7.1.1 Servlet POJOs

SIP servlet POJOs may be added programmatically by using the following methods.

- `addServlet(String servletName, String className)`
- `addServletPOJO(String servletName, Object servlet)`
- `addServletPOJO(String servletName, Class <?> servletClass)`
- `<T> T createServletPOJO(Class<T> clazz)`

3.7.2 Listeners

The following types of Listeners may be added programmatically by the application.

- `javax.servlet.ServletContextAttributeListener`
- All listeners in section 10.1 SIP Servlet Event Types and Listener Interfaces except `javax.servlet.sip.AttributeStoreBindingListener and AutomaticProcessingListener.`

The following methods in `ServletContext` may be used by the application to add Listeners.

- `void addListener(String className)`
- `<T extends EventListener> void addListener(T t)`
- `void addListener(Class <? extends EventListener> listenerClass)`
- `<T extends EventListener> void createListener(Class<T> clazz)`

3.7.3 Configuring a SIP Application programmatically

An application may programmatically configure the SIP Application, apart from adding and configuring Servlets and Listeners. This may be done by using an instance of the `SipApplicationConfiguration` interface made available by the container via the `SipServletContext.getApplicationConfiguration()` method. Table 3-2 describes the methods of `SipApplicationConfiguration`.

Table 3-2 SipApplicationConfiguration

`getApplicationName()`	Gets the name of the SIP application.
`getMainServlet()`	Gets the name of the main servlet of the application.
`setMainServlet(String servletName)`	Sets the main servlet of the application.
`getProxyTimeout()`	Get the proxy time-out.
`getSessionTimeout()`	Gets the session time-out.
`getDistributable()`	Gets whether the application is distributable or not.

3.8 Unsupported ServletContext methods

`SipServletContext` is a child of `javax.servlet.ServletContext`. However, many methods in `javax.servlet.ServletContext` are specific to the HTTP protocol and do not apply to SIP servlets. The following lists the unsupported methods. These methods return null or throw `UnsupportedOperationException` if the return type is void.

- `addFilter(String filterName, Class<? extends Filter> filterClass)`
- `addFilter(String filterName, Filter filter)`
- `addFilter(String filterName, String className)`
- `createFilter(Class<T> clazz)`
- `getContext(String uripath)`
- `getContextPath()`
- `getDefaultSessionTrackingModes()`
- `getEffectiveSessionTrackingModes()`
- `getFilterRegistration(String filterName)`
- `getJspConfigDescriptor()`
- `getMimeType(String file)`
- `getRealPath(String path)`
- `getRequestDispatcher(String path)`

- `getResourcePaths(String path)`
- `getSessionCookieConfig()`
- `getVirtualServerName()`
- `setSessionTrackingModes(Set<SessionTrackingMode> sessionTrackingModes)`

4 SIP Servlet POJOs

SIP Servlet POJOs are SIP Servlets that do not extend from generic Servlets defined bythe `GenericServlet` interface. They are simple POJOs annotated with a `@SipServlet` annotation. These POJOs contain annotated methods invoked by the SIP Servlet Container when a SIP message arrives at the container.

Any Java class that is annotated with `@SipServlet` , but does not extend from `javax.servlet.sip.Servlet`, is a SIP Servlet POJO. SIP Servlet POJOs support all elements of the `@SipServlet` annotation. The SIP Container treats POJOs similar to a component class listed in Table EE.5-1 of Java EE specification. These classes act as a CDI managed bean and hence support all CDI capabilities explained in the CDI specification.

4.1 SIP Servlet POJO Life Cycle

The POJOs are instantiated by the procedure to create a non-contextual instance that is not managed by the CDI container. The exact procedure is explained at the end of *section* titled *"Support for Dependency Injection"* of Java EE specification.

The POJO life-cycle closely follows the procedure of other component classes in terms of instantiation and destruction. Thus, shortly after resource and CDI injection completes successfully, the `@PostConstruct` callback is invoked. Similarly, the `@PreDestroy` annotation may be applied to one method that is called when the class is taken out of service and is longer be used by the container. Otherwise, the load-on-startup behavior of SIP Servlets applies to SIP Servlet POJOs as well. The behavior can be specified either in the `@SipServlet` annotation or in the deployment descriptor. See section 22.3.2 @SipServlet Annotation for details about `SipServlet` annotation and how it maps to the deployment descriptor.

4.2 SIP Meta Annotations

A SIP Servlet POJO uses annotated methods to handle SIP Messages. Any Java method annotated with one or more annotations that carry the following SIP meta-annotations is used by the container to deliver messages to the application. SIP servlet containers do not depend on individual annotations that use these meta annotations enabling further extensibility as explained in 4.5 Extensibility.

4.2.1 @SipMethod

The `SipMethod` annotation associates the name of a SIP method with an annotation. A Java method annotated with a runtime annotation that is itself annotated with `SipMethod` handles SIP requests or responses with the indicated SIP method. The value of the annotation specifies the name of the SIP method (E.g., "INVITE"). If annotation is specified on a method whose first parameter is not a `SipServletRequest` or `SipServletResponse`, a deployment error occurs.The annotation is defined as follows:

```
@Target(value=ANNOTATION_TYPE)
@Retention(value=RUNTIME)
@Documented
public @interface SipMethod {
  String value();
}
```

4.2.2 @SipResponseCode

The `SipResponseCode` annotation associates a response code with an annotation. A Java method annotated with a runtime annotation that is itself annotated with `SipResponseCode` handles SIP responses with the specified code. The value of the annotation specifies the response code. If an annotation is specified on a method whose first parameter is not `SipServletResponse`, a deployment error occurs. The annotation is defined as follows:

```
@Target({ANNOTATION_TYPE})
@Retention(value=RUNTIME)
@Documented
public @interface SipResponseCode {
  int value();
}
```

4.2.3 @SipResponseRange

The `SipResponseRange` annotation associates a response filter with an annotation. A Java method annotated with a runtime annotation that is itself annotated with `SipResponseRange` handles SIP responses satisfying the filter. If an annotation is specified on a method whose first parameter is not `SipServletResponse`, a deployment error occurs. The specified range includes both `begin` and `end` values. The element `begin` specifies the beginning of the response range. The element `end` specifies the end of the response range. The annotation is defined as follows:

```
@Target({ANNOTATION_TYPE})
@Retention(value=RUNTIME)
@Documented
public @interface SipResponseRange {
  int begin() default 100;
  int end() default 999;
}
```

4.2.4 @SipPredicate

The `SipPredicate` annotation applies a predicate with a Java method in a SIP Servlet POJO. When a Java method is annotated with runtime annotations that are annotated with `SipPredicate`, then those predicates are evaluated by the SIP Servlet Container before executing the method. Thus, while other meta-annotations allow applications to define their own annotations, `SipPredicate` enables applications to provide further filtering of messages based on specific logic. For example, an application can filter decisions based on the value of the SIP header or the state of the SIP Session. The annotation is defined as follows:

```
@Target({ANNOTATION_TYPE})
@Retention(value=RUNTIME)
@Documented
public @interface SipPredicate {
Class<? extends Predicate> value() default Predicate.class;
}
```

The value of the annotation is a type of implementation of `javax.servlet.sip.Predicate`. The SIP Servlet Container instantiates the class and invokes the `Predicate.apply(SipServletMessage)` method to determine whether or not to invoke the method.

4.3 Method specific annotations

This specification defines an annotation for each known SIP method, such as INVITE, ACK, and REGISTER. Each runtime annotation is annotated with the SIP meta annotation `@SipMethod`. See the definition of `@Invite` below.

```
@Retention(RUNTIME)
@Target({METHOD})
@SipMethod("INVITE")
public @interface Invite {
}
```

A method annotated with `@Invite` is invoked by the SIP Container for SIP requests or SIP responses based on the type of the first parameter of the method. When the parameter is of type SipServletRequest.class, it is invoked for all SIP requests with the method INVITE. When the parameter is of type SipServletResponse.class, it is invoked for all SIP responses for the method INVITE.

Here is an example SIP Servlet POJO that uses the `@Invite` annotation.

```
@SipServlet
public class FooPOJO {

  @Invite
  public void handleInviteRequest(SipServletRequest request) {
    //...
  }

  @Invite
  public void handleInviteResponse(SipServletResponse response) {
    //...
  }
}
```

This specification defines the following method-specific annotations:

1. `@Invite`
2. `@Ack`
3. `@Options`
4. `@Bye`

5. `@Cancel`
6. `@Register`
7. `@Prack`
8. `@Subscribe`
9. `@Notify`
10. `@Message`
11. `@Info`
12. `@Update`
13. `@Refer`
14. `@Publish`

4.3.1 @AnyMethod Annotation

The `@AnyMethod` annotation can handle SIP messages corresponding to any SIP method. Applications may use `@AnyMethod` annotation to receive all SIP requests and SIP responses without checking the method. If annotation is specified on a method whose first parameter is not a `SipServletRequest` or `SipServletResponse`, a deployment error occurs. See below for the definition of the `@AnyMethod` annotation.

```
@Retention(RUNTIME)
@Target({METHOD})
public @interface AnyMethod {
}
```

4.4 Response Filtering

SIP Servlet 2.0 defines annotations for filtering responses. The defined annotations are `@ProvisionalResponse`, `@SuccessResponse`, `@RedirectResponse`, and `@ErrorResponse`. These annotations use the SIP meta-annotation `@SipResponseRange` to specify the response code range.

See below for the definition of `@ProvisionalResponse`.

```
@Retention(RUNTIME)
@Target({METHOD})
@SipResponseRange(begin = 101, end = 199)
public @interface ProvisionalResponse {
}
```

Table4-1 explains annotations and their response range.

Table 4-1 Annotations and their response range

Annotation	Response range begin	Response range end
@ProvisionalResponse	101	199
@SuccessResponse	200	299
@RedirectResponse	300	399
@ErrorResponse	400	699

Thefollowing example method uses a response range annotation:

```
@SuccessResponse
public void handleSuccessResponse(SipServletResponse response) {
  //...
}
```

To further filter responses, a POJO may combine both a Method specific annotation and a Response Filter annotation. The following example shows how to handle success responses for INVITE messages:

```
@Invite @SuccessResponse
public void handleInviteSuccessResponse(SipServletResponse response) {
  //...
}
```

It is also possible to specify multiple response filter annotations to a Java method. It allows application developers to handle multiple ranges using the same annotation. The following example shows how to handle multiple ranges:.

```
@Invite @ProvisionalResponse @SuccessResponse
public void handleInviteSuccessResponse(SipServletResponse response) {
  //...
}
```

4.4.1 @BranchResponse Annotation

An application may use the built-in `SipPredicate`,`@BranchResponse`, to associate an intermediate final response that arrived on a `ProxyBranch` with a Java method in a SIP Servlet POJO. See below for the definition of the `@BranchResponse` annotation.

```
@Retention(RUNTIME)
  @Target({METHOD})
  @SipPredicate(BranchResponse.Predicate.class)
public @interface BranchResponse {

    class Predicate implements
    javax.servlet.sip.Predicate<SipServletResponse> {
      @Override
      public boolean apply(final SipServletResponse response) {
        return response.isBranchResponse();
      }
    }
```

4.5 Extensibility

Meta-annotations provide a built-in extensibility. SIP meta-annotations allow application developers to define and use their own annotations. For example, an application can define an annotation called `foo.example.18xResponses` for handling only 18x responses. A single annotation can also contain more than one SIP meta-annotation. The following example shows a user defined annotation:

```
@Retention(RUNTIME)
@Target({METHOD})
@SipResponseRange(begin = 200, end = 299)
@SipMethod("INVITE")
public @interface MySucessfulInviteResponse {
}
```

The following example shows an annotation that combines a `SipPredicate` with other meta annotations:

```
@Retention(RUNTIME)
  @Target({METHOD})
  @SipPredicate(MyInitialInvite.Predicate.class)
  @SipMethod("INVITE")
  public @interface MyInitialInvite {
    class Predicate implements
    javax.servlet.sip.Predicate<SipServletRequest> {
      @Override
      public boolean apply(final SipServletRequest request) {
        return request.isInitial();
      }
    }

  }
```

The application can then use the `@MyInitialInvite` annotation to select a method for handling an initial invite.

```
@MyInitialInvite
  public void handleInviteRequest(SipServletRequest request) {
    //...
  }
```

4.6 Method Selection Procedure

Containers must ensure that only one java method is invoked for a particular SIP message arriving at the container. When more than one Java method has matching annotations, the container selects the Java method with most specific annotations.

For example, given a success response to an INVITE that could go to either of two methods, one of which matches any response to an INVITE and the other which matches only success responses to an INVITE, the container chooses the second method. These rules are applied in a particular order.For example, assume that two methods each take a `SipServletResponse`. If one method only matches the INVITE method, and another method only matches 200 OK responses, a 200 OK response to an INVITE goes to the INVITE-matching method because the SIP Method matching has higher priority than response-code matching.

The following sections outline the procedure for finding the most specific Java method.

4.6.1 Notations

Jm = A Java method annotated by annotations with SIP meta annotations.

Sm = A `@SipMethod` annotation.

Sr = A `@SipResponseRange` annotation.

Sc = A `@SipResponseCode` annotation.

Sp **=** A `@SipPredicate` annotation.

ESm = (Method of the `SipServletMessage`) eq (value of ***Sm)***

ESr = (Status code of the `SipServletResponse`) in (range of ***Sr)***

Esc = (Status code of the `SipServletResponse`) eq (value of ***Sc)***

ESp **=** Result of evaluation of `@SipPredicate`.

4.6.2 Basic Rule

SIP Servlet containers determine that a *Jm* meets the selection criteria if the annotations evaluate the following equation to true. If an annotation is not present in a sub-expression, the equation evaluates to true.

(ESm-1|| ESm-2 ||..||ESm-n) && ((ESc-1 || ESc-2 ||..||ESc-n) ||(ESr-1 || ESr-2 || ..||ESr-n)) && (ESp-1 || ESp-2||..||ESp-n)

4.6.3 Conflict resolution

If more than one *Jm* meets the selection criteria, the container selects the *Jm* with the smallest count of *Sm*. If the number of *Sms* is equal, the container selects the *Jm* with the smallest count of *Sc*. If the number of Scs are equal, the container selects the one with the shortest span of *Srs*.

4.6.4 Requests

- ***Sc*** (`SipResponseCode`) and ***Sr*** (`SipResponseRange`) are not applicable to Requests.
- If none of the methods satisfy the basic rule, the container selects a Java method annotated with `@AnyMethod`.
- If none of the above is true, it results in the default behavior as specified by the section 2.3 of the SIP Servlet specification.

4.6.5 Responses

- If none of the methods satisfy the basic rule, the container selects a Java method annotated with `@AnyMethod`.

4.6.6 SipPredicate and Method Selection

If an application includes one or more `@SipPredicate` annotations, the application must ensure that the predicates are not written to cause conflict during selection of methods.

4.6.7 Deployment

During deployment, the SIP servlet container scans the annotations present.If more than one Java method has the same SIP meta-annotation specificity using `@SipMethod, @SipResponseCode`, or `@SipResponseRange`, the container fails the deployment.

4.6.8 Examples

4.6.8.1 Conflict resolution examples

Following examples explain which method is selected when there are more than one Java method that match the selection criteria.

Listing 4-1 Example 1

```
@Invite
@SuccessResponse
public void handleResponse01(SipServletResponse response) {
}

@SuccessResponse
public void handleResponse02(SipServletResponse response) {
}
```

For a 200/INVITE response, the container will select `handleResponse01`, since that is more specific of the two methods for that message. For a message with another SIP method, the container will select `handleResponse02.`

Listing 4-2 Example 2

```
@Invite
@SuccessResponse
public void handleResponse01(SipServletResponse response) {
}

@InviteOkResponse
public void handleResponse02(SipServletResponse response) {
}

@Retention(RetentionPolicy.RUNTIME)
```

```
@Target(METHOD)
@SipResponseCode(200)
@SipMethod("INVITE")
@interface InviteOkResponse { }
```

For a 200/INVITE response, the container will select `handleResponse02`, since that is more specific of the two methods for that message. For 201/INVITE message, the container will select `handleResponse01.`

Listing 4-3 Example 3

```
@SuccessResponse
public void handleResponse01(SipServletResponse resp ) {
}

@NonFailureResponses
public void handleResponse02(SipServletResponse resp) {
}

@OkResponse
public void handleResponse03(SipServletResponse resp ) {
}

@Retention(RUNTIME)
@Target(METHOD)
@SipResponseRange(begin = 100, end = 399)
@interface NonFailureResponses { }

@Retention(RUNTIME)
@Target(METHOD)
@SipResponseCode(200)
@interface OkResponse { }
```

In this example, for a 200 response `handleResponse03` is selected since that is the only method that matches the response codes (Sc). There is no conflict. For a 201 response, both `handleResponse01 and handleResponse02` matches the criteria. However since the `handleResponse01` has the shorter span of response range than the `handleResponse02`, container will choose that. For a 302 response, container will select `handleResponse02` since that is the only method that match the criteria.

4.6.8.2 Examples of container failing deployment

Following examples explain the situations in which container fail the deployment since it cannot decide which method will be more specific for a message.

Listing 4-4 Example 4

```
@Invite
@Message
public void handleRequest01(SipServletRequest req) {
}

@Invite
@Register
public void handleRequest02(SipServletRequest req) {
}
```

In this example, container will fail the deployment, since both the methods contain same number of requests and for an INVITE request, container will find two methods with same specificity.

Listing 4-5 Example 5

```
@OptionsOkResponse
@SubscribeAccept
public void handleResponse01(SipServletResponse resp) {
}

@InfoOkResponse
@SubscribeNoNotification
public void handleResponse02(SipServletResponse resp) {
}

@Retention(RUNTIME)
@Target(METHOD)
@SipResponseCode(200)
@SipMethod("INFO")
@interface InfoOkResponse { }

@Retention(RUNTIME)
@Target(METHOD)
@SipResponseCode(200)
```

```
@SipMethod("OPTIONS")
@interface OptionsOkResponse { }

@Retention(RUNTIME)
@Target(METHOD)
@SipResponseCode(204)
@SipMethod("SUBSCRIBE")
@interface SubscribeNoNotification { }


@Retention(RUNTIME)
@Target(METHOD)
@SipResponseCode(201)
@SipMethod("SUBSCRIBE")
@interface SubscribeAccept { }
```

In the above listing, the container will fail the deployment because of the following. The `handleResponse02` supports both SUBSCRIBE and INFO methods and 200 and 204 responses. The `handleResponse01` supports SUBSCRIBE and OPTIONS methods and 200 and 201 responses. Hence, for 200/SUBSCRIBE responses, the specificity of SIP meta annotations is the same.

Listing 4-6 Example 6

```
@InviteNonFailureResponses
public void handleResponse01(SipServletResponse resp) {
}

@Invite
@NonFailureFinalResponses
public void handleResponse02(SipServletResponse resp) {
}

@Retention(RUNTIME)
@Target(METHOD)
@SipResponseRange(begin = 100, end = 299)
@SipMethod("INVITE")
@interface InviteNonFailureResponses { }

@Retention(RUNTIME)
@Target(METHOD)
@SipResponseRange(begin = 200, end = 399)
@interface NonFailureFinalResponses { }
```

In the above listing, the span of the response ranges are exactly of the same size and all other SIP meta annotations are the same in both the methods. For overlapping responses, SIP meta annotations in the two methods have the same specificity. Container will fail the deployment.

Listing 4-7 Example 7

```
@Invite
@MySpecialResponses
public void handleResponse01(SipServletResponse resp) {
}

@InviteSpecialResponses
public void handleResponse02(SipServletResponse resp) {
}

@Retention(RUNTIME)
@Target(METHOD)
@SipMethod("INVITE")
@SipPredicate(MyPredicate.class)
@interface InviteSpecialResponses { }

@Retention(RUNTIME)
@Target(METHOD)
@SipPredicate(MyPredicate.class)
@interface MySpecialResponses { }

class MyPredicate implements Predicate<SipServletResponse> {
  @Override public boolean apply(SipServletResponse response) {
    //logic.
  }
}
```

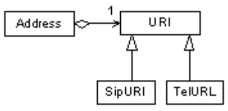
Since all the sip meta annotations of both methods are the same and the `@SipPredicate` annotation use the same class, the container will fail the deployment.

5 Addressing

Addressing plays a role in many SIP functions. Therefore, in addition to the request URI, many header fields are defined to carry one or more addresses, such as From, To, and Contact. The address format is generally the same. They consist of a URI with an optional display name and an optional set of parameters.

Figure 5-1 shows the addressing abstractions of the SIP Servlet API and the relationship between them.

Figure 5-1 Addressing Abstractions



The `SipFactory` interface described in 3.2 The SipFactory constructs instances of these interfaces.

5.1 The Address Interface

The Address interface represents values of header fields that conform to the following address BNF rule:

```
address= (name-addr | addr-spec) *(SEMI generic-param)
```

The constituent non-terminals are defined in RFC 3261, chapter 25, and are (incompletely) given below for ease of reference.

```
        name-addr=  [ display-name ] LAQUOT addr-spec RAQUOT
        addr-spec            =  SIP-URI / SIPS-URI / absoluteURI
        display-name  =  *(token LWS)/ quoted-string
```

The baseline SIP specification defines the following set of header fields that conform to this grammar: From, To, Contact, Route, Record-Route, Reply-To, Alert-Info, Call-Info, and Error-Info [RFC 3261]. The `SipServletMessage` interface defines a set of methods that operate on any address header field (see 6.4.1 Parameterable and Address Header Fields). This includes the RFC 3261 defined header fields listed above as well as extension headers such as Refer-To [refer] and P-Asserted-Identity [privacy].

Future SIP extensions will likely define more header fields.Thus, it is useful to have a set of generic methods that can be used by applications to access header fields as `Address` objects rather than as `String`s.

SIP Servlet containers typically have built-in knowledge of certain headers but must be able to handle unknown headers as well. When an application attempts to access an unknown header as an address header (by calling one of the methods discussed in 6.4.1 Parameterable and Address Header Fields), the container MUST try to parse all values of that header field as `Address` objects according to the address rule.

The actual definition of address-based header fields differs in small ways:

- some define specific parameters and possibly limit the set of values they can take. For example, the From and To headers define a tag parameter which, when present, must be a token.
- some may not have parameters at all, such as Reply-To.
- some may not have a display name

The address BNF rule can be thought of as defining a superset of all legal values for a large number of header fields. If the SIP Servlet container knows the actual, possibly more restrictive, definition for a particular address-based header, it should enforce any such constraint when serializing a message.

5.1.1 The Parameterable Interface

Some SIP headers follow the following form described in RFC 3261, section 7.3.1:

```
field-name: field-value *(;parameter-name[=parameter-value])
```

where the field-value may be in `name-addr` or `addr-spec` format as defined in RFC 3261 or may be any sequence of tokens until the first semicolon.

The list of headers in RFC 3261 that follow this form are Accept, Accept-Encoding, Accept-Language, Alert-Info, Call-Info, Content-Disposition, Content-Type, Error-Info, Retry-After, and Via. The `Address` API manifests a specialized form of this BNF for Address headers like Contact, From, To, Route, Record-Route, and Reply-To. Besides these, there are a number of other extensions where this form is applicable like Accept-Contact and Reject-Contact from RFC 3841. A generic `Parameterable` interface caters to all the SIP headers of this form.

The `Address` interface extends this interface.

Methods are available in `SipFactory` to create a `Parameterable` object from a `String` object and `SipServletMessage` to read/write `Parameterable` objects. In general, any SIP header having parameters in the form described above MUST be available as `Parameterable` to the application.

5.1.2 The From and To Header Fields

The From and To header fields contain the addresses of the UAC and UAS respectively. 13.1.1.1 Copying From and To Addresses discusses how the container guarantees the integrity of From and To headers through cloning and immutability.

RFC 4916 extends the use of the From header by allowing the From header field URI to change during a dialog to reflect the connected identity. It also deprecates the mandatory reflection of the original To and From URIs in mid-dialog requests and their responses. This extension is compatible for containers compliant with [RFC 3261] as it requires the use of only tags from the From and To headers for dialog identification [RFC 3261 Section 12.2.1.1] and not the entire URIs as in the case for [RFC 2543].

This specification allows modifications to all parts of the From and To headers except the tag parameter. From and To headers are still considered system headers with respect to the tags (tag parameters). Containers should preserve any changes to the From and To headers in mid-dialog and use the updated ones for any subsequent requests in the dialog.

From and To header modification is supported through the following API methods:

```
SipServletMessage.getAddressHeader("From");
SipServletMessage.getAddressHeader("To");
SipServletMessage.getFrom();
SipServletMessage.getTo();
```

Applications can change all parts of the `Address` returned by the above methods except the tag parameter. The tag parameter MUST NOT be modified by applications, and containers MUST throw an `IllegalStateException` if an attempt is made to set the tag parameter for these headers. The following methods operating on the From and To headers MUST continue to throw `IllegalArgumentException`:

```
SipServletMessage.setAddressHeader("From", fromHdr)
SipServletMessage.addAddressHeader("To", toHdr)
SipServletMessage.removeAddressHeader("From", fromHdr)
```

Note: Containers that need to support [RFC 2543] MUST NOT allow modification of the From and To headers as that RFC requires the entire URI for dialog identification. Container support for Connected Identity [RFC 4916] is optional in this specification and is indicated by the presence of the "from-change" option tag in the `javax.servlet.sip.supported` list. (See 3.3 Extensions Supported).

5.1.3 The Contact Header Field

The Contact header field specifies one or more locations (URIs) where a user is reachable. This header field plays two different roles in SIP. One role is as a mechanism for a UA to specify, for example in INVITE and 2xx responses to INVITE, to its peer UA the exact address to which the peer should address subsequent messages within that dialog. In this case, the Contact header field always has a single value. Servlets must not set the Contact header in these cases. Containers know which network interfaces they listen on and are responsible for choosing and adding the Contact header. Containers should throw an `IllegalArgumentException` on application attempts to set the Contact header field in these cases.

The second role of Contact is in REGISTER requests and responses, as well as 3xx and 485 responses. The value of Contact header fields in these messages specify alternate addresses for a user, and there may be more than one. In some cases, SIP servlets may legitimately set Contact addresses. The Contact header field defines two “well-known” parameters, `q` and `expires`, and the `Address` interface includes methods for those parameters.

The special Contact value “*” is used in REGISTER requests when the UAC wishes to remove all bindings associated with an address-of-record without knowing their precise values. The `isWildcard` method returns true for `Address` objects representing the wild-card Contact value and `SipFactory.createAddress` returns a wild-card `Address` given a value of “*”. Wild-card `Address` objects are legal only in Contact header fields.

The Contact header is a system header, which means that it is managed by the container and cannot be modified or set by the applications except for the following messages:

1. REGISTER requests and responses
2. 3xx responses
3. 485 responses

4. 200/OPTIONS responses

For any other messages, Contact header constituents are modifiable by applications as described below:

1. A UA application MUST be able to set parameters in the Contact header.
2. A UA application MUST be able to set the user part of the Contact header.
3. A UA application MUST be able to modify the display name in the Contact header.
4. Per Sec 11.2 of RFC 3261, "*Contact header fields MAY be present in a 200 (OK) response and have the same semantics as in a 3xx response. That is, they may list a set of alternative names and methods of reaching the user.*" Thus,UAs responding to OPTIONS must be capable of setting Contact header(s) like in a 3xx response.

UA applications can perform these three operations on the Contact URI as retrieved using the following API:

```
SipServletMessage.getAddressHeader("Contact");
```

The `Address` returned should be good only for specifying a set of parameters that the application can set on `Address` and/or `URI` and set the user part in the URI. The host component of the URI and URI Scheme are irrelevant and cannot be trusted to reflect the actual values that the container will be using when inserting a Contact header into Request or Response. The container MUST ignore or overwrite any host/port set on the Contact URI accessed as above. This is because the container is responsible for managing the actual network listen points and uses these to create the Contact headers' host/port.

Applications MUST NOT modify the following list of URI parameters, and containers MUST ignore any of these parameter values set by applications.

- method
- ttl
- maddr
- lr

Applications may set any other SIP URI parameter or Contact header parameter relevant for the message.

The following method MUST continue to throw `IllegalArgumentException` wherever the Contact header is defined as a system header by this specification:

```
SipServletMessage.setAddressHeader("Contact", contactHdr);
```

In case the application modifies the Contact as specified above but then proxies the request, the containers MUST ignore any modification made to the Contact header.

5.1.3.1 Transport Parameters in Contact Header

When setting the Contact header for an outgoing message, containers MUST set the transport parameter to the transport that sent the message.

5.2 URIs

SIP entities are addressed by URIs. When initiating or proxying a request, SIP servlets identify the destination by specifying a URI. SIP defines its own URI scheme that SIP containers are required to support. Some implementations may know how to handle other URI schemes, e.g., tel URLs [RFC 3966]. Implementations must be able to represent URIs of any scheme, so that if, for example, a 3xx response contains a Contact header with a mailto or http URL, the container can construct `Address` objects containing `URI`s representing those Contact URIs. Also, `SipFactory.createURI` should return a `URI` instance given any valid URI string. The container may not be able to route SIP requests based on such URIs but must be able to present them to applications.

5.2.1 SipURI

This interface represents SIP and SIPS URIs. Implementations must be able to route requests based on SIP URIs.

SIP and SIPS URIs are similar to email addresses in that they are of the form user@host, where user is either a user name or telephone number and host is a host name, a domain name, or a numeric IP address. Additionally, SIP/SIPS URIs may contain parameters and headers. See RFC 3261, section 19.1.1 for restrictions on the contexts in which various parameters are allowed. Headers are allowed only in SIP/SIPS URIs appearing in Contact headers or in external URIs, for example when being used as a link on a Web page.

For example, the following SIP URI contains a transport parameter with value “tcp” and a Subject header with value “SIP Servlets”:

```
sip:alice@example.com;transport=tcp?Subject=SIP%20Servlets
```

The string form of SIP/SIPS URIs may contain escaped characters. The SIP servlet container is responsible for unescaping those characters before presenting URIs to servlets. Likewise, string values passed to setters for various SIP/SIPS URI components may contain reserved or excluded characters that need escaping before being used. The container is responsible for escaping those values as necessary. Syntactically, SIP and SIPS URIs are identical except for the name of the URI scheme. The semantics differ in that the SIPS scheme implies that the identified resource is to be contacted using TLS. Quoting from RFC 3261:

> "A SIPS URI specifies that the resource be contacted securely. This means, in particular, that TLS is to be used between the UAC and the domain that owns the URI. From there, secure communications are used to reach the user, where the specific security mechanism depends on the policy of the domain. Any resource described by a SIP URI can be "upgraded" to a SIPS URI by just changing the scheme, if it is desired to communicate with that resource securely."

SIP and SIPS URIs are both represented by the `SipURI` interface as they are syntactically identical and are used the same way. The `isSecure` method can be used to test whether a `SipURI` represents a SIP or a SIPS URI, and the `setSecure` method can be used to change the scheme.

5.2.2 TelURL

This interface represents tel URLs as defined in [RFC 3966]. The tel URL scheme represents addresses of terminals in the telephone network, such as telephone numbers. SIP servlet containers may or may not be able to route SIP requests based on tel URLs, but must be able to parse and represent them. `SipFactory.createURI` must return an instance of the `TelURL` interface when presented with a valid tel URL.

5.2.3 Resolving Telephone Numbers to SipURI

ENUM (E.164 Number Mapping as specified RFC 6116) is a system that uses DNS to translate telephone numbers, like '+12025332600', into URIs. A `TelURL` or `SipURI` with `user=phone` parameter constitutes a telephone number. Often, SIP servlet applications need to resolve this telephone number using ENUM as specified in RFC 3824. The SIP servlet API provides a `DnsResolver` interface to resolve telephone numbers in a `TelURL` or `SipURI` that has a `user=phone` parameter. `DnsResolver` is implemented by the container and is made available to applications as a `ServletContext` parameter with the name `javax.servlet.sip.DnsResolver`. It can also be accessed using resource injection as specified in section 22.3.8 Annotation for DnsResolver Injection. After a SIP servlet obtains the `DnsResolver`, it can use the following methods to resolve the telephone numbers to `SipURI`:

- `SipURI resolveToSipURI(URI uri)`
- `List<SipURI> resolveToSipURIs(URI uri)`
- `List<String> resolveToStrings(URI uri, String enumService)`

`DnsResolver` also contains utility methods that help applications while resolving the URIs. The `toEnum(URI uri)` method helps applications to get the representation of a URI in ENUM format. The `resolvesInternally(SipURI uri)` method helps applications decide whether the `SipURI` resolves to the internal container.

6 Requests and Responses

This chapter describes the structure of SIP servlet message objects. Chapters 10 and 11 describe operational aspects of message processing.

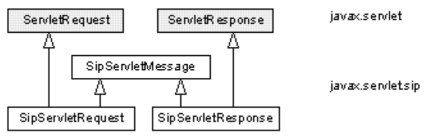
6.1 The SipServletMessage Interface

The generic Servlet API is defined with an implicit assumption that servlets receive requests from clients, inspect various aspects of the corresponding `ServletRequest` object, and generate a response by setting various attributes of a `ServletResponse` object. Because HTTP servlets reside only on origin servers and always generate responses to incoming requests, this model is a good fit for HTTP.

The requirement that SIP services must be able to initiate and proxy requests implies that SIP request and response classes must be more symmetric. That is, requests must be writable as well as readable. Likewise, responses must be readable as well as writable.

The `SipServletMessage` interface defines a number of methods that are common to `SipServletRequest` and `SipServletResponse`, such as setters and getters for message headers and content. Figure 6-1 illustrates how the SIP request and response interfaces extend the generic `javax.servlet` interfaces the same way that the HTTP request and response interfaces do, but additionally implement the `SipServletMessage` interface.

Figure 6-1 Request-response Hierarchy



6.2 Implicit Transaction State

`SipServletRequest` and `SipServletResponse` objects always implicitly belong to a SIP transaction. The transaction state machine (as defined by the SIP specification) constrains the messages that can legally be sent at various points of processing. If a servlet attempts to send a message that violates the transaction state machine, the container throws an `IllegalStateException`.

A `SipServletMessage` is called committed when one of the following conditions is true:

- the message is an incoming request for which a final response has been generated
- the message is an outgoing request that was sent
- the message is an incoming non-reliable provisional response received by a servlet acting as a UAC
- the message is an incoming reliable provisional response for which PRACK was already generated
- the message is an incoming final response received by a servlet acting as a UAC for a Non-INVITE transaction
- the message is a response that has been forwarded upstream
- the message is an incoming final response to an INVITE transaction and an ACK was generated
- the message is an outgoing request, the client transaction has timed out, and no response was received from the UAS and the container generates a 408 response locally

The semantics of the committed message is that it cannot be further modified or sent in any way. Also see 6.9 Accessibility of SIP Servlet Messages.

6.3 Access to Message Content

The HTTP Servlet API provides access to message content through a stream metaphor. Applications have access to posted data through an `InputStream` or a `Reader` and can generate content through either an `OutputStream` or a `Writer`.

SIP is more like email than HTTP with regard to message content. Messages are generally smaller.Also, using chunked encoding for generated content is not practical due to the SIP requirement that a Content-Length header be included in all messages. SIP applications often need access to parsed representations of message content.

For these reasons, there are no stream-based content accessors defined for SIP messages, and the following `ServletRequest` and `ServletResponse` methods must return null:

- `getInputStream`
- `getReader`
- `getOutputStream`
- `getWriter`

The `SipServletMessage` interface defines the following set and get methods for message content (exceptions omitted for clarity):

```
int getContentLength();
void setContentLength(int len);

String getContentType();
void setContentType(String type);

Object getContent();
byte[] getRawContent();
void setContent(Object obj, String type);
```

The interface parallels how the JavaMail API defines access to message content in the `javax.mail.Part` interface [JavaMail]. The `getRawContent` method is not present in JavaMail but is useful when writing back-to-back user agents (B2BUA) as these may copy content from one message to another without incurring the overhead of parsing it (as they may not actually care what is in the body). The type of `Object` returned by `getContent` depends on the message's Content-Type. It is required to return a `String` object for MIME type text/plain as well as for other text MIME media types for which the container does not have specific knowledge. It is encouraged that the object returned for multipart MIME content is a `javax.mail.Multipart` object. For unknown content types other than text, the container must return a `byte[]`. Likewise, `setContent` is required to accept `byte[]` content with any MIME type, and `String` content when used with a text content type. When invoked with non-`String` objects and a text content type, containers should invoke `toString()` on the content `Object` to obtain the body's character data. Again, it is recommended that implementations know how to handle `javax.mail.Multipart` content when used together with multipart MIME types.

6.3.1 Character Encoding

Several of the message content accessors discussed above need to be able to convert between raw eight-bit bytes and sixteen-bit Unicode characters. The character encoding attribute of `SipServletMessage` specifies which mapping to use:

```
String getCharacterEncoding();
void setCharacterEncoding(String enc)
        throws UnsupportedEncodingException;
```

Character encodings are identified by strings and generally follow the conventions documented in [RFC 2278]. The character encoding may affect the behavior of methods `getContent` and `setContent`. A message's character encoding may be changed by calls to `setCharacterEncoding` and `setContentType`. For incoming messages, the character encoding is specified by the charset parameter of the Content-Type header field, if such a parameter is present.

Note: Because of the evolution of servlet spec 2.4, `SipServletResponse.setCharacterEncoding()`, which extends both `SipServletMessage` and `ServletResponse`, now does not throw the `UnsupportedEncodingException` because it inherits a more generic method from `ServletResponse`. This is a binary compatible change, meaning that the compiled applications shall run on the v1.1 containers without change.However, it is not a source compatible change if the application is explicitly catching the `UnsupportedEncodingException` on `SipServletResponse.setCharacterEncoding()`.

6.4 Headers

A servlet can access SIP message headers through the following methods of the `SipServletMessage` interface (see also [Servlet API, sections 4.3 and 5.2]):

```
String getHeader(String name);
ListIterator getHeaders(String name);
List getHeaderList(String name);
Iterator getHeaderNames();
```

```
  List getHeaderNameList();
  void setHeader(String name, String value);
  void addHeader(String name, String value);
```

The `getHeader` method allows access to the value of a named header field. Some header fields, for example Warning, may have multiple values in a SIP message. In this case, `getHeader` returns the first value of the header field. The `getHeaders` and `getHeaderList` methods allow access to all values of a specified header field by returning an iterator over `String` objects and a `Set` of `String` objects representing those values respectively.

The `setHeader` method sets a header with a given name and value. If a previous header exists, it is replaced by the new header. In the case where a set of header values exist for the given name, all values are cleared and replaced with the new value.

The `addHeader` method adds a header with the specified name and value to the message. If one or more headers with the given name already exists, the new value is appended to the existing list.

6.4.1 Parameterable and Address Header Fields

The methods listed above treat SIP header field values as `Strings`. As discussed in 5.1 The Address Interface, many SIP header fields carry `Parameterables` (and hence `Addresses`). Having the ability to access `Parameterables` (and hence `Addresses`) in a parsed form is more convenient and allows for better performance than accessing those header field values as `Strings`. The following methods on the `SipServletMessage` interface are defined in terms of the `Parameterable` and `Address` interfaces:

```
  Parameterable getParameterableHeader(java.lang.String name);
  java.util.ListIterator getParameterableHeaders(java.lang.String name)
  java.util.List getParameterableHeaderList(java.lang.String name)
  void setParameterableHeader(java.lang.String name, Parameterable param)
  void addParameterableHeader(java.lang.String name, Parameterable param,
boolean first)

  Address getAddressHeader(String name);
  ListIterator getAddressHeaders(String name);
  List getAddressHeaderList(String name);
  void setAddressHeader(String name, Address addr);
  void addAddressHeader(String name, Address addr, boolean first);
```

If a header field has multiple `Parameterable` values in a message, the `getParameterableHeader` method returns the first value. The `getParameterableHeaders` and `getParameterableHeaderList` methods allow access to all values of the specified header field, returning an iterator over `Parameterable` objects and a `Set` of `Parameterable` objects respectively.

The same explanation applies to the `getAddressHeader`, `getAddressHeaders`, and `getAddressHeaderList` methods if a header field has multiple `Address` values.

`Parameterable` and `Address` objects obtained from or added to messages are live in the sense that modifications made to them cause the corresponding header field value of the underlying message or messages to be modified accordingly.

`Parameterable` and `Address` objects may belong to more than one `SipServletMessage` at a time. In this case, modification of any `Parameterable` or `Address` object results in modification of the value of the underlying header field for all messages. This sort of aliasing can result in bugs when not intended. Application writers may practice defensive programming by deep cloning the `Parameterable` and `Address` objects to avoid sharing.

6.4.2 System Headers

The term system header refers to headers that are managed by the servlet container and which servlets must not attempt to modify directly via calls to `setHeader` or `addHeader`. This includes the headers Call-ID, From, To, CSeq, Via, Record-Route, Route, Path, and Contact when used to confer a session signaling address. That is, in messages other than REGISTER requests and responses, 200 response to OPTIONS as described by section 11.2 [RFC3261], and 3xx and 485 responses.

The following rules apply when applications try to modify system headers:

- The Call-ID and CSeq headers cannot be modified by the application
- The Via header can only be modified by adding or removing non-protected parameters
- An Address system header can be modified by the application (as long as the modification generates a valid SIP header) in all of its parts except for:
  - The host/port and scheme part in the Address URI. This restriction does not apply to the To and From headers.
  - Protected parameters and their value.
- Contact header can be modified/added by the application as described in 5.1.3 The Contact Header Field.
- For the From and To headers, all parts of the headers except the tags (tag parameter) can be modified as described in 5.1.2 The From and To Header Fields.

The protected parameters defined by this spec are: method, ttl, maddr, transport, user, branch, received, and tag. These parameters are protected when used either as a URI parameter or as a Header parameter.

There is no need for services to set the system headers directly and disallowing it makes it easier for containers to ensure correct protocol behavior. SIP servlet containers throw an `IllegalArgumentException` on attempts to modify system headers against the rules specified above.

Containers are required to enforce this immutability requirement also when system headers are accessed through `Address` objects or through the `ListIterator` returned by the `getAddressHeaders` method.

6.4.3 TLS Attributes

If an incoming request or response is transmitted over a secure protocol, such as TLS, this information must be exposed via the `isSecure` method of the `SipServletMessage` interface.

Note: The `isSecure` method indicates whether a message was received over a secure transport. However, since secure protocols are frequently used in a hop-by-hop manner in SIP, it does not follow that the message was sent over a secure transport in all hops before reaching the container. It may have passed over an insecure link at some point. If there is a TLS certificate associated with the message, it MUST be exposed to the servlet programmer as an array of objects of type `java.security.cert.X509Certificate` and accessible via a `SipServletRequest` or `SipServletResponse` attribute of `javax.servlet.request.X509Certificate` for requests and `javax.servlet.response.X509Certificate` for responses. Since the TLS connection is hop-by-hop, this TLS certificate is not available beyond the first application in case of an application composition chain as described in 18.6 Transport Information.

6.5 Transport Level Information

The following `SipServletMessage` methods allow applications to obtain transport level information from incoming messages:

```
String getLocalAddr();
int getLocalPort();
String getRemoteAddr();
int getRemotePort();
String getTransport();
String getInitialRemoteAddr();
int getInitialRemotePort();
String getInitialTransport();
```

These methods return the IP address/port number of the SIP interface on which the message was received (`getLocalAddr, getLocalPort`), the IP address/port number of the sender of the message (`getRemoteAddr, getRemotePort, getInitialRemoteAddr, getInitialRemotePort`), as well as the transport protocol used, e.g., UDP, TCP, TLS, or SCTP (`getTransport`). Note that these methods have meaning for only incoming SIP messages. For all other messages such as application created outgoing messages and container generated responses, these methods return null (or -1 for ports). See section 18.6 Transport Information for the impact of application composition on these methods.

The method `SipServletRequest.isRequestUriInternal()` helps to decide if the request-uri of the SIP request is a SIP or SIPS URI and if that URI is addressed internally to the container. Similarly, the method `SipServletMessage.isInternallyRouted()` identifies whether the message was sent by the an application in the container.

6.6 Attributes

A servlet can bind an object attribute into a `SipServletRequest` or `SipServletResponse` by name. Hence, `SipServletRequest` and `SipServletResponse` are attribute stores. For more information about attributes and the attribute store, see Chapter 7, “Attributes”.

6.7 Requests

The `SipServletRequest` object encapsulates information of a SIP request, including headers and the message body.

6.7.1 Accessing Active Requests

Every request has a string identifier which is unique within the `SipSession` it belongs to. Applications may obtain the request identifier by using the `SipServletRequest.getId()` method. Applications can access the active request from the session by using the `SipSession.getActiveRequest(String requestId)` method.

A request can be accessed until a final response is sent or received using this method. A request created by the application is not considered active until it is sent. Hence, such requests are not available using this method.

6.7.2 Parameters

Request parameters are strings sent by the client to a servlet container as part of a request. The parameters are made available to applications as a set of name-value pairs. Multiple parameter values can exist for any given parameter name. The following `ServletRequest` methods are available to access parameters:

```
String getParameter(String name);
Enumeration getParameterNames();
String[] getParameterValues(String name);
```

The `getParameterValues` method returns an array of `String` objects containing all of the parameter values associated with a parameter name. The value returned from the `getParameter` method must always equal the first value in the array of `String` objects returned by `getParameterValues`.

Note: Support for multi-valued parameters is defined mainly for HTTP because HTML forms may contain multi-valued parameters in form submissions.

When passing an incoming `SipServletRequest` to an application, the container populates the parameter set in a manner that depends on the nature of the request:

- For initial requests where an application is invoked, the parameters are those present on the request URI, if this is a SIP or a SIPS URI. For other URI schemes, the parameter set is undefined.
- For initial requests where a preloaded Route header specifies the application to invoke, the parameters are those of the SIP or SIPS URI in the Route header.
- For subsequent requests in a dialog, the parameters sent to the application are those that the application itself set on the Record-Route header for the initial request or response (see 12.4 Record-Route Parameters). These are typically the URI parameters of the top Route header field. However, if the upstream SIP element is a "strict router", the parameters may be returned in the request URI (see RFC 3261). It is the container's responsibility to recognize whether the upstream element is a strict router and to determine the right parameter set. This specification introduces access to the popped Route header, which the container pops when the Route header points towards the container. This mechanism allows the application to have complete access to the popped route header and all parameters. See 6.7.3 Popped Route Header. It is recommended that applications access the Route parameters from the popped route rather than from the request parameters. In the future, this mechanism may be deprecated.

6.7.3 Popped Route Header

On receiving an initial request that contains a SIP Route header (preloaded) or receiving a subsequent request with a Route header (converted from a Record-Route header), a SIP Servlet container determines if the request is intended for itself (this is based on local policy, for example IP addresses of interfaces or representative DNS entries). If an initial request is for the SIP Servlet container, the container MUST remove the Route header before passing it to any application or the Application Router.

A side effect of removing a SIP Route message header before presenting the request to applications (and Application Router) is that applications do not have access to the SIP Route message header and its associated information. Certain architectures utilize the SIP Route header for transporting application and other related information. The following methods return the Route header popped by the container:

```
Address getPoppedRoute();
Address getInitialPoppedRoute();
```

If application composition is being used, the values returned by these methods may differ. The `getPoppedRoute` method returns the route popped before the current application invocation in the composition chain. The `getInitialPoppedRoute` method returns the route popped by the container when it first received the request.

If no header is popped by the SIP Servlet container on an initial request, both methods return null.

Both methods return the Route header as an `Address`. So, parameters added to the Record-Route header using the `Proxy.getRecordRouteURI()` API should be retrieved not from the popped route `Address` directly but from the URI of the popped route `Address`.

6.8 Responses

Response objects encapsulate headers and content sent from UA servers upstream to the client. Unlike the case for HTTP, a single request may result in multiple responses. The `SipServletResponse.getRequest()` method returns the request object associated with a response. For UAC and UAS applications, this is the original `SipServletRequest`. For proxying applications, it is an object for the request sent on the branch on which the response was received. The original request object can be obtained by using the `getProxy().getOriginalRequest()` method (see 12.2.4.2 Correlating responses to proxy branches and 8.2.3 Creation of SipSessions).

6.8.1 Reliable Provisional Responses

Provisional responses (1xx's) are not sent reliably in baseline SIP. However, a SIP extension allows transmission of 1xx's other than 100 with guarantees concerning reliability and ordering [RFC 3262]. This is useful in cases where a provisional response carries information critical to providing a desired service, often related to PSTN inter-working.

Support for the 100rel extension is mandatory in SIP Servlet containers ("100rel" is the name of the option tag defined for the provisional response extension). The container must include the string "100rel" in the list of supported extensions available to applications through the `ServletContext`. See 3.3 Extensions Supported.

Applications can determine at runtime whether the container supports the 100rel extension by testing whether `String` is in the supported list.

To send a 1xx reliably, the application invokes `SipServletResponse.sendReliably()`. If this method call is successful, the container sends the response reliably. A `Rel100Exception` is thrown if:

- the response is not within 101-199 range
- the request is not an INVITE
- the UAC did not indicate support for the 100rel extension in a Supported or Required header
- the container does not support the extension

For containers that do support the 100rel extension, the RSeq and RAck headers are system headers (see 6.4.2 System Headers). That is, they are handled by the container and may not be added, modified, or deleted by applications. PRACK requests are treated as subsequent requests, meaning they are associated with the same `SipSession` as the corresponding INVITE and are delivered to `Servlet.service()` whose SIP Servlet implementation dispatches to the `doPrack()` method.

The following method is defined on `SipServletResponse` to create PRACK requests:

```
SipServletRequest createPrack();
```

Since different PRACKs can be generated on different reliable responses and since RAck is system header, this method must be used to correctly create a PRACK request.

If no PRACK is received for a reliable provisional response within the time specified by RFC 3262, the container notifies the application through the `noPrackReceived` method of the `SipErrorListener` interface if this is implemented by the application. It is then up to the application to generate the 5xx response recommended by RFC 3262 for the INVITE transaction. The original INVITE request as well as the unacknowledged reliable response is available from the `SipErrorEvent` passed to the `SipErrorListener`.

When a container supporting 100rel receives a retransmission of a reliable provisional response, it does not invoke the application(s) again.

Also, containers supporting 100rel are responsible for guaranteeing that UAC applications receive incoming reliable provisional responses in the order defined by the RSeq header field.

Applications can obtain an identifier for each reliable provisional response by using the `SipServletResponse.getProvisionalResponseId()` method. Applications can check if the response is a reliable provisional response by using the `SipServletResponse.isReliableProvisional()` method.

6.8.2 Obtaining responses

An application can access a response being acknowledged by an ACK or PRACK request by using the `SipServletRequest.getAcknowledgedResponse()` method. An application can look up an unacknowledged provisional response by using the `SipSession.getUnacknowledgedProvisionalResponse(String provisionalResponseId) method`. It can also access all unacknowledged provisional responses that belong to a session by using the `SipSession.getUnacknowledgedProvisionalResponses(UAMode mode)` method.
An application can access the final response of `SipServletRequest` by using the `SipServletRequest.getFinalResponse` method.

6.8.3 Buffering

The Servlet API defines a number of `ServletResponse` methods related to buffering content returned by a server in responses [Servlet API, section 5.1]:

- `getBufferSize`
- `setBufferSize`
- `reset`
- `flushBuffer`

These methods can improve performance of HTTP Servlets significantly. However, unlike HTTP, SIP is not intended as a content transfer protocol and buffering is not usually a concern. Therefore, it is not expected that these methods will yield any useful result and implementations may simply do nothing. It is recommended that `getBufferSize` return 0.

The `isCommitted()` method returns true if the message is committed in the sense of 6.2 Implicit Transaction State.

6.9 Accessibility of SIP Servlet Messages

Access to SIP Servlet messages is not limited to the scope of the Servlet's `service` method. Applications can handle a SIP message within the service method as well as from other threads (either timer events or any other thread in the system) outside the scope of the Servlet's `service` method. For example, an application can respond to a request based on an event delivered via JMS.

A `SipServletMessage` cannot be sent again after it is committed as defined in 6.2 Implicit Transaction State. Further, since the committed message belongs to a context that completed its state machine or life-cycle, any modification of the message is meaningless. Containers SHOULD throw an `IllegalStateException` for any mutation attempt on a committed message. However, the URI and other headers of committed messages can be used to construct a new message, subject to following restriction: In case the URI or any other header is modified for use, it MUST be cloned by the application as it is not guaranteed that containers will return a deep copy on access. Even if a message is committed, the container may still access them for handling a retransmission.

6.10 Servlet 3.0 Asynchronous Processing

Servlet 3.0 introduced the ability for asynchronous processing of requests so the thread may return to the container and perform other tasks before generating a response. However, SIP Servlets always remain asynchronous for both requests and responses, and this new asynchronous processing capability is not supported by SIP Servlets. Hence, the `isAsyncSupported()` method of `ServletRequest` always returns false for SIP Servlets. The behavior of other associated methods are defined to depend on `isAsyncSupported` and behave accordingly.

6.11 Internationalization

Language identification tags are used in several places in SIP, notably Accept-Language and Content-Language headers and the charset parameter of various Content-Type header values.

In the SIP Servlet API, languages are identified by instances of `java.util.Locale`.

Note: While the `javax.servlet` interfaces `ServletRequest` and `ServletResponse` contain methods related to internationalization, these assume that servlets only respond to incoming requests and are insufficient for the SIP Servlet API.

6.11.1 Indicating Preferred Language

User agents may optionally indicate to proxies and peer UAs in which natural language(s) it prefers to receive content, reason phrases, and warnings. This information can be communicated from the UA by using the Accept-Language header.

The following methods are provided in the `SipServletMessage` interface to allow the sender of a message to indicate preferred locale(s):

```
void setAcceptLanguage(Locale locale);
void addAcceptLanguage(Locale locale);
```

The `setAcceptLanguage` method sets the preferred language of the Accept-Language header and removes any existing Accept-Language header.The `addAcceptLanguage` method adds another (least preferred) locale to the list of acceptable locales.

The following `SipServletMessage` methods are used to determine the preferred locale of the message sender:

```
  Locale getAcceptLanguage();
  Iterator getAcceptLanguages();
  Set getAcceptLanguageSet();
```

The `getAcceptLanguage` method returns the preferred locale that the client accepts for content. See section 14.4 of RFC 2616 (HTTP/1.1) for more information about how the Accept-Language header must be interpreted to determine the preferred language of the client. If no preferred locale is specified by the client, `getAcceptLanguage()` must return null, `getAcceptLanguages()` must return an empty `Iterator,` and `getAcceptLanguageSet()` must return en empty `Set`. (Note that this behavior has changed from JSR 116 and is noted in Appendix A.)

The `getLocales` method returns an `Iterator` over the set of `Locale` objects indicating, in decreasing order starting with the preferred locale, the locales that are acceptable to the UA originating the message. If no preferred locale is specified by the client, `getLocale` must return the default locale for the servlet container and `getLocales` must return an `Iterator` over a single `Locale` element of the default locale.

6.11.2 Indicating Language of Message Content

When sending a message containing a body, SIP Servlets may indicate the language of the body by calling the `setContentLanguage` method of the `SipServletMessage` interface:

```
  void setContentLanguage(Locale locale);
```

This method must correctly set the Content-Language header (along with other mechanisms described in the SIP specification) to accurately communicate the `Locale` to the client.

Note that a call to the `setContent` or `setContentType` methods with a charset component for a particular content type sets the message's character encoding.

The default encoding of message content is “UTF-8” if none has been specified by the servlet programmer. Upon receiving messages, servlets can obtain information regarding the locale of the message content by using the following `SipServletMessage` method:

```
Locale getContentLanguage();
```

7 Attributes

This chapter explains attributes used in SIP servlet applications and how they are stored in attribute stores.

7.1 Attributes

Attributes are objects identified by a specified name. Attributes are usually set by the application to store application-specific information in objects that are exposed by the container. The container may also set attributes to express information that could not otherwise be expressed via the API.

Attributes are visible only in the same application. The application boundary is determined by EAR files. If an application accesses the attributes across different modules of the same EAR file, the class visibility rules apply for the class of the attribute objects. For example, if an EAR file contains two SIP modules (SAR1 and SAR2), SAR1 can access the attributes stored in attributes stores of SAR2 (section 7.2 Attribute Stores) only if the class of the attribute is visible to SAR1. For more information about classloading and class visibility pertaining to SIP servlet applications, see section 9.10 Servlet Application Classloader.

Attributes are not shared between different applications in an application composition chain. For example, attributes set on `SipServletMessage` are not available to other applications in a composition chain.

7.1.1 Attribute Names

Only one attribute object may be associated with an attribute name in an attribute store. Attribute stores are explained in the following sections of this chapter. All attributes placed in the attribute store should be named in accordance with the reverse package name convention suggested by the Java Programming Language Specification [JLS].

Attribute names beginning with the prefix `javax.servlet.sip.` are reserved for definition by this specification and should not be used by applications.

7.2 Attribute Stores

Many interfaces exposed by the SIP servlet API are capable of storing attributes in their instances. All such interfaces extend the `AttributeStore` interface and are called Attribute Stores. `SipSession`, `SipApplicationSession`, `ForkingContext`, `SipServletRequest`, `SipServletResponse`, and `InviteBranch` are the attribute stores at the time of writing of this specification.

Table 7-1 explains the `AttributeStore` interface.`AttributeStore` provide *happens-before*

Table 7-1 AttributeStore Interface

`Object getAttribute(String name)`	Retrieves an attribute from attribute store
`void removeAttribute(String name)`	Removes an attribute from the attribute store
`void setAttribute(String name, Object attribute)`	Places or updates an attribute in the attribute store
`Set<String> getAttributeNameSet()`	Returns a set of attribute names
`void clearAttributes()`	Removes all of the attributes that the application placed on the attribute store

concurrency guarantee that retrieval operations (`getAttribute`) reflects the most recently completed update operation (`setAttribute` and `removeAttribute`).

7.3 AttributeStoreListener

Implementations of the `AttributeStoreListener<T extends AttributeStore>` interface receive notifications when attributes are added, removed, or replaced from the attribute store. Containers should examine the generic type of `AttributeStoreListener` to invoke the correct listeners. For example, the container must make sure that a listener that implements `AttributeStoreListener<SipSession>` is invoked only for attributes of the `SipSession` attribute store. However, a listener that implements `AttributeStoreListener` without any generic type is invoked for all types of attribute stores.

`AttributeStoreBindingEvent<T extends AttributeStore>` is the type of event object used for all events of `AttributeStoreListener`.

7.4 AttributeStoreBindingListener

Attributes that implement the `AttributeStoreBindingListener<T extends AttributeStore>` interface receive notifications when attributes are bound or unbound from the attribute store. This interface defines the following methods that signal an object being bound into, or being unbound from, an attribute store.

- `valueBound`
- `valueUnbound`

The `valueBound` method must be called before the object is made available via the `getAttribute` method of the `AttributeStore` interface. The `valueUnbound` method must be called after the object is no longer available via the `getAttribute` method of the `AttributeStore` interface.

Containers must examine the generic type of `AttributeStoreBindingListener` to invoke the correct listeners. For example, the container must ensure that a listener that implements `AttributeStoreBindingListener<SipSession>` is invoked only for attributes of `SipSession` attribute store. However, a listener that implements `AttributeStoreBindingListener` without any generic type can be invoked for all types of attribute stores.

`AttributeStoreBindingEvent<T extends AttributeStore>` is the type of event object used for all events of `AttributeStoreListener`.

8 Sessions

SIP applications typically must process multiple messages in order to provide the intended service. As servlets themselves are stateless (or rather, contain no per-dialog or per-transaction data), the API provides a mechanism that allows messages to be correlated and specify how containers associate application data with subsequent messages processed in an “application instance”.

The HTTP Servlet API provides such a mechanism in the form of HTTP sessions. The HttpSession interface allows servlets to correlate a series of HTTP requests from a particular client and also acts as a store for application data.

The `SipSession` interface is the SIP Servlet API equivalent of `HttpSession`. It represents a point-to-point relationship between two user agents and roughly corresponds to a SIP dialog [RFC 3261]. However, SIP applications are typically more complicated than Web applications:

- many services involve multiple dialogs, for example conferencing applications and applications acting as back-to-back user agents and third-party controllers
- converged applications communicate with other network elements using multiple protocols, for example SIP, HTTP, email, etc.
- application composition allows for many applications active on one call.

This implies that there may be more than one application invoked on a single call and any one application instance may consist of multiple point-to-point relationships, and that these relationships may employ different protocols. This is reflected in the SIP Servlet API through the notions of protocol sessions and application sessions. A protocol session is a protocol specific session object typically representing a point-to-point relationship. The `SipSession` and `HttpSession` interfaces are both examples of protocol sessions.

An application session in a sense represents an application instance. It contains a number of protocol sessions and is also used as a container for application data. All protocol sessions belonging to the same application instance belong to the same `SipApplicationSession`. For example, a SIP servlet acting as a back-to-back user agent will consist of two SipSessions and one `SipApplicationSession` for each application instance.

Containers may, as an optimization, create application session and SIP session objects lazily, for example postpone creation until requested by the application. The result should be indistinguishable from an implementation that always creates the session objects.

8.1 SipApplicationSession

The application session serves two purposes: it provides a storage for application data and correlates a number of protocol sessions. Note: The application session is not SIP specific, but for practical reasons, this version of the SIP Servlet API defines the application session as `javax.servlet.sip.SipApplicationSession`, whereas the more logical choice would be `javax.servlet.ApplicationSession`. It is our hope that a future version of the Servlet API will adopt the notion of application sessions, in which case the `SipApplicationSession` will be deprecated and/or refactored to extend `ApplicationSession`.

8.1.1 Protocol Sessions

The following `SipApplicationSession` methods allow servlets to obtain contained protocol sessions:

- `getSessions()` iterates over all valid child protocol session objects
- `getSessions(String protocol)` iterates over all valid child session objects that are of the specified protocol, for example “SIP” to get all `SipSession`s, and “HTTP” to get all `HttpSession`s.
- `getSipSession(java.lang.String id)` returns a certain `SipSession` by its id (since v1.1)
- `getSession(java.lang.String id, Protocol protocol)` returns a child session associated with the specified protocol by its id (since v1.1)

8.1.2 SipApplicationSession Lifetime

Containers manage session data and so need a mechanism for knowing when `SipApplicationSession` objects are no longer in use and are therefore eligible for garbage collection. The mechanism provided is known as `SipApplicationSession` invalidation. An application session becomes invalidated in one of three ways:

1. The `SipApplicationSession` expires and the container subsequently invalidates it.
2. A servlet explicitly invalidates it by invoking the `invalidate()` method.
3. A servlet marks the `SipApplicationSession` to be invalidated and the container invalidates it when the `SipApplicationSession` is in the ready-to-invalidate state as described in 8.1.2.2.2 Invalidate When Ready Mechanism.

For reasons of performance, it is recommended that applications explicitly invalidate `SipApplicationSession` and `SipSession` objects as soon as possible. Note: It would be unfortunate if applications were to force creation of an application session just so that they can invalidate it. The `getApplicationSession(boolean create)` method can be used with a false argument to avoid forcing creation of the session object.

When used, an application session expiration timer ensures that application sessions will eventually become eligible for container invalidation regardless of whether an application explicitly invalidates them. Servlets can register for application session timeout notifications using the `SipApplicationSessionListener` interface. In the `sessionExpired()` callback method, the application may request an extension of the application session lifetime by invoking `setExpires()` on the timed out `SipApplicationSession` giving as an argument the number of minutes until the session expires again. The container may grant the extension of session lifetime, grant it with a modified timeout value, or reject it. The ability to accept with a different timeout value allow containers to apply their own policies on application session lifetime and timeouts. A container might for example choose to enforce a maximum total lifetime for application sessions. A `SipApplicationSession` object is said to be expired when its expiration timer is no longer active. An expired `SipApplicationSession` is not invalid until the container explicitly invalidates it.

The ability to extend session lifetime is useful to applications because it allows them to not use an unrealistically high expiration timer value in cases where application lifetime depends on some “external” event, that is, an event unknown to the servlet container.

8.1.2.1 SipApplicationSession Timer Operation and SipApplicationSession Expiration

When a `SipApplicationSession` is created, the `SipApplicationSession` timer starts if the the timeout value specified by the session-timeout parameter in the deployment descriptor or `@SipApplication(sessionTimeout)` annotation is set to a positive number. If not specified, the default timeout value is set to 3 minutes.Explicit invalidation of the `SipApplicationSession` leads to cancellation of the application session timer. If the session timeout value is 0 or less, then an application session timer never starts for the `SipApplicationSession` object and the container does not consider the object to ever have expired. However, if a the session timeout value was set to 0 or less, and a servlet subsequently calls the `setExpires()` method on a `SipApplicationSession` object, it is left up to container implementation whether to accept that request and start an expiration timer for the `SipApplicationSession` or to reject the `setExpires()` call by returning 0. If the container accepts the `setExpires()` request thereby starting an expiration timer, then it becomes the container's responsibility to invalidate the `SipApplicationSession` when it expires if the application neglects to do so.

A `SipApplicationSession` expires in one of two ways:

1. Container rejects an application session lifetime extension by returning 0 to a `setExpires()` call.

2. The application’s `SipApplicationSessionListener` implementation chooses not to call `setExpires()` on the `SipApplicationSession` object inside the `sessionExpired()` callback.

If the application session timer is active and the expiration timeout is reached, then the `sessionExpired()` method of any `SipApplicationSessionListener` implementation is called.

8.1.2.2 SipApplicationSession Invalidation

There are two ways in which a `SipApplicationSession` can be invalidated:

1. Explicit invalidation Mechanism

2. Invalidate When Ready Mechanism

Once a `SipApplicationSession` object is invalidated by either the application or the container, it may no longer be used. All references to the object should be removed by the container and applications as soon as possible thus enabling invalidated `SipApplicationSession` objects to be garbage collected. The container MUST invoke the listener callback sessionDestroyed() if a listener exists for both `SipSessions` and `SipApplicationSessions` when they are destroyed.

8.1.2.2.1 Explicit Invalidation Mechanism

An application may invalidate a `SipApplicationSession` at any time using the invalidate() method. On explicit invalidation, the container MUST purge all state for that `SipApplicationSession` from its memory. This includes the application state stored in the `SipApplicationSession` as well as all the contained protocol session objects. The container will also invoke `Future.cancel(false)` for all asynchronously submitted tasks as specified in section 17.3.2 Concurrency Utilities.

The `invalidate()` method will throw an `IllegalStateException` if the `SipApplicationSession` object has already been invalidated. Invalidating a `SipApplicationSession` using the `invalidate()` method causes all the protocol sessions contained within it to be explicitly invalidated by the container. Explicit invalidation of `SipSession` objects is described in 8.2.4.1 SipSession Invalidation.

8.1.2.2.2 Invalidate When Ready Mechanism

The explicit invalidation mechanism described above causes containers to invalidate `SipSessions` immediately, which could result in partially invalidated application paths and orphaned sessions in other network entities participating in the dialog.

This specification provides a mechanism for invalidation that applications can use to circumvent the above drawbacks called the "Invalidate When Ready" mechanism.

A `SipApplicationSession` is said to be in a ready-to-invalidate state if the following conditions are met:

1. All the contained `SipSessions` are in the ready-to-invalidate state.
2. None of the `ServletTimers` associated with the `SipApplicationSession` are active.
3. None of the asynchronous tasks submitted as specified in 17.3.2 Concurrency Utilities is active.

A `SipApplicationSession` transitions into the ready-to-invalidate state when the following conditions are met:

1. The last protocol session belonging to the `SipApplicationSession` is invalidated
2. The last ServletTimer associated with the `SipApplicationSession` expires.

3. The last active asynchronous task as specified in the section 17.3.2 Concurrency Utilities is completed

A `SipSession` is in the ready-to-invalidate state if it can be explicitly invalidated such that the SIP state is terminated cleanly across all the SIP network entities participating in the dialog. Refer to 8.2.4.2 Important Semantics for details on when container determines a `SipSession` is in the ready-to-invalidate state.

This specification provides methods to help applications invalidate `SipApplicationSessions` cleanly. The methods introduced are:

1. `isReadyToInvalidate()` - returns true if the `SipApplicationSession` is in the ready-to-invalidate state and false otherwise.

2. `setInvalidateWhenReady(boolean flag)` - allows applications to indicate to the container to notify it when the `SipApplicationSession` is in the ready-to-invalidate state. The container notifies the application using the `SipApplicationSessionListener.sessionReadyToInvalidate(SipApplicationSessionEvent se)` callback method.

3. `getInvalidateWhenReady()` - returns true if the container will notify the application when the `SipApplicationSession` is in the ready-to-invalidate state.

An application willing to invalidate a `SipApplicationSession` cleanly could use the callback mechanism to perform any application clean up before the `SipApplicationSession` gets invalidated by the container.

Servlets can register for `sessionReadyToInvalidate` notifications on the `SipApplicationSessionListener` interface. In the `sessionReadyToInvalidate` callback method, an application may choose to invalidate the `SipApplicationSession` or perform any other cleanup activities. If the application does not explicitly invalidate the `SipApplicationSession` in the callback or has not defined a listener, the container will invalidate the `SipApplicationSession`.

Applications may also use the callback to call `setInvalidateWhenReady(false)` to indicate to the container to not observe this `SipApplicationSession` anymore. In this case, the containers MUST not invalidate the `SipApplicationSession` after the callback. Applications could then either rely on explicit invalidation mechanism or again call `setInvalidateWhenReady(true)`. This parallels the expiry callback mechanism defined above in 8.1.2.1 SipApplicationSession Timer Operation and SipApplicationSession Expiration.

The firing of the `SipApplicationSession` expiry timer influences the lifetime of a `SipApplicationSession` and overrides the behavior of a `SipApplicationSession` marked with `invalidateWhenReady(true)`. If the `SipApplicationSession` times out when it is not yet ready to be invalidated state, an application could detect it in the `sessionExpired` callback of the `SipApplicationSessionListener` and extend the lifetime of the `SipApplicationSession` using `setExpires`. Failing to do so will cause the `SipApplicationSession` expiry to explicitly invalidate all the contained sessions and itself. Conversely, a `SipApplicationSession` that transitions to the ready-to-invalidate state may have an active expiry timer. The container MUST cancel the expiry timer before it invalidates the `SipApplicationSession`.

8.1.3 Binding Attributes into a SipApplicationSession

A servlet can bind an object attribute into a `SipApplicationSession` by name. Hence `SipApplicationSession` is an attribute store. See more information on attributes and attribute store at Chapter 7, “Attributes”.

8.2 SipSession

`SipSession` objects represent point-to-point SIP relationships, either as established dialogs or in the stage before a dialog is actually established. The `SipSession` can be obtained from a `SipServletMessage` by calling the `getSession` method.

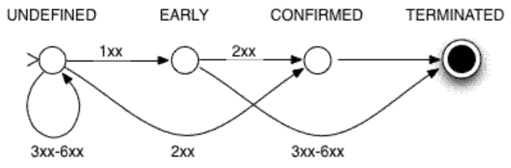
8.2.1 Relationship to SIP Dialogs

A `SipSession` represents either an actual SIP dialog in its early, confirmed, or terminated state defined in RFC 3261, or else represents a pseudo dialog. The notion of pseudo dialogs extend the definition of dialogs to have a certain well-defined meaning before a dialog is established in the RFC 3261 sense and after it has transitioned away from the early state because of a non-2xx final response being received. The `SipSession` interface embodies this notion of pseudo dialogs and because of this, some `SipSession` instances do not correspond to SIP dialogs.

Containers enforce SIP protocol restraints based on the dialog and transaction state. If it is illegal to send a message in a given state, an `IllegalStateException` is thrown by the container.

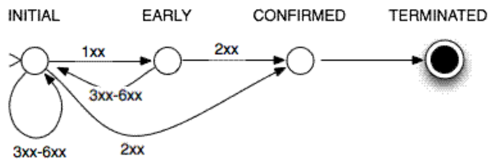
The SIP dialog state machine is shown in Figure 8-1.

Figure 8-1 The SIP Dialog State Machine



The “undefined” state in Figure 6-1 is not a real state. For dialogs created as the result of an INVITE, the dialog springs into existence on receipt of a 1xx or 2xx with a To tag.

Figure 8-2 The SipSession State Machine



`SipSession` objects are in one of the four states: INITIAL, EARLY, CONFIRMED, or TERMINATED. These states represent the state of a dialog that may be associated with the `SipSession` object. A method on the `SipSession` class is defined that allows the application to have access to the dialog state. The `SipSession` state depends not only the state of an underlying SIP dialog but also on whether the servlet has acted as a UAC, UAS or a proxy. The rules governing the state of a `SipSession` object are given below. Note that for any state transition caused by the receipt of a SIP message, the state change must be accomplished by the container before calling the `service()` method of any `SipServlet` to handle the incoming message.

1. Before a dialog creating request is received or sent on a `SipSession`, the `SipSession` state is defined to be INITIAL.
2. In general, whenever a non-dialog creating request is sent or received, the `SipSession` state remains unchanged. Similarly, a response received for a non-dialog creating request also leaves the `SipSession` state unchanged. The exception to the general rule is that it does not apply to requests (e.g. BYE, CANCEL) that are dialog terminating according to the appropriate RFC rules relating to the kind of dialog.

3. If the servlet acts as a UAC and sends a dialog creating request, then the `SipSession` state tracks directly the SIP dialog state except that non-2XX final responses received in the EARLY or INITIAL states cause the `SipSession` state to return to the INITIAL state rather than going to TERMINATED.

   a. Further, when non-2XX final responses are received, `SipSession` state of all sessions in the EARLY or INITIAL state and in the same forking context (as specified in section 8.2.6 Forking and SipSessions) also return to INITIAL state.

   b. Also, if 2xx final response for INVITE is received and subsequently when the transaction timer M fires as per RFC 6026, `SipSession` state of all sessions in the EARLY or INITIAL state and in the same forking context (as specified in section 8.2.6 Forking and SipSessions) transitions to TERMINATED state.

4. If the servlet acts as a UAS and receives a dialog creating request, then the `SipSession` state directly tracks the SIP dialog state. Unlike a UAC, a non-2XX final response sent by the UAS in the EARLY or INITIAL states causes the SipSession state to go directly to the TERMINATED state.

5. If the servlet acts as a proxy for a dialog creating request then the `SipSession` state tracks the SIP dialog state except that non-2XX final responses received from downstream in the EARLY or INITIAL states cause the `SipSession` state to return to the INITIAL state rather than going to the TERMINATED state. This enables proxy servlets to proxy requests to additional destinations when called by the container in the `doResponse()` method for a tentative non-2XX best response. After all such additional proxy branches have been responded to and after considering any servlet created responses, the container eventually arrives at the overall best response and forwards this response upstream. If this best response is a non-2XX final response, then when the forwarding takes place, the state of the `SipSession` object becomes TERMINATED. If this best response is a 2XX final response, then the `SipSession` state becomes CONFIRMED.

   a. Further when a non-2xx final response is forwarded as the best response, `SipSession` state of all sessions in the EARLY or INITIAL state and in the same forking context (as specified in section 8.2.6 Forking and SipSessions) transitions to TERMINATED state.

   b. Also, if a 2xx final response for INVITE request is forwarded as best response and subsequently when the transaction timer M fires as per RFC 6026, `SipSession` state of all sessions in the EARLY or INITIAL state and in the same forking context (as specified in section 8.2.6 Forking and SipSessions) transitions to TERMINATED state.

6. Because setting the supervised flag to false affects only whether responses are seen for the transaction associated with the current request, the value of the supervised flag has no effect on the `SipSession` state.

An enum that defines the possible SIP dialog states is defined for use with the `SipSession` interface:

```
public enum SipSession.State {INITIAL, EARLY, CONFIRMED, TERMINATED }
```

The following method is introduced on the `SipSession` interface to return the current SIP dialog state:

```
public SipSession.State getState()
```

This method returns one of the `SipSession.State` enum values - INITIAL, EARLY, CONFIRMED or TERMINATED. These values represent the SIP dialog related state of the SipSession when the method is called.

The standard SIP rules governing when a second or subsequent response cause a single request to establish multiple dialogs hold unmodified for `SipSession`s. If, for example, two 200 responses are received for an initial INVITE, the container will create a second SipSession on receipt of the second 200 response. This second `SipSession` will be derived from the one in which the INVITE was generated as described in 8.2.3.2 Derived SipSessions below. Both `SipSession`s will then represent dialogs in the CONFIRMED state.

The INITIAL state is introduced to allow a UAC to generate multiple requests with the same Call-ID, From (including tag), and To (excluding tag), and within the same CSeq space. This is useful, for example, in the following situations:

- When a UAC receives a 3xx for a request initiated outside of a dialog and decides to retry with the Contact addresses received in the 3xx, it is recommended to reuse the same To, From and Call-ID for the new request [RFC 3261, section 8.1.3.4].
- When a UAC receives certain “non-failure” 4xx responses indicating that the request can be retried, e.g. 401, 407, 413, 415, 416, and 420 [RFC 3261, section 8.1.3.5].
- REGISTER requests sent from a UAC to the same registrar should all use the same Call-ID header field value and should increment the CSeq by one for each request sent [RFC 3261, section 10.2].
- When a UAC using the session timer extension receives a 422 response to an initial INVITE it retries with the same Call-ID and a higher Min-SE value [timer].

These examples have in common, a need to create similar requests without an established dialog being in place. There may well be other scenarios where it’s desirable to correlate non-dialog requests by Call-ID and ensuring proper sequencing by using the CSeq header field. A new request can be created from the INITIAL state only when there is no ongoing transaction. The request URI should be changed if the request is to be sent to different target than previous request.

Note that the "pseudo dialog" semantics presented here is defined for use in UAC applications only. Containers treat incoming requests as subsequent requests, i.e., routes to existing sessions, only if those requests belong to an actual established SIP dialog. There is no expectation, for example, that a container treat an incoming INVITE as a subsequent request after it has previously sent a 3xx response to another INVITE with the same Call-ID, CSeq, and From (including tag).

8.2.1.1 CANCEL Message Processing

Whether a servlet acts as a proxy or as a UAS, receiving a CANCEL request does not in itself cause a `SipSession` state change. However, since receiving a CANCEL request causes the UAS to respond to an ongoing INVITE transaction with a non-2XX (specifically, 487) response, the `SipSession` state normally becomes TERMINATED as a result of the non-2XX final response sent back to the UAC. Note that the default CANCEL processing behavior of the container may be turned off by the application using `SipApplication` annotation as specified in section 22.3.3 @SipApplication Annotation.

8.2.2 Maintaining Dialog State in the SipSession

UAs use the `SipSession` method `createRequest` to create subsequent requests in a dialog. This implies that containers must associate SIP dialog state with `SipSession`s when the application acts as a UA. The dialog state is defined in RFC 3261 and consists of the following pieces of data: local/remote URIs, local/remote tags, local/remote sequence numbers, route set, remote target URI, and the secure flag.

8.2.2.1 When in a Dialog

For applications acting as UAs, the `SipSession` must track dialog state according to RFC 3261. Proxies cannot generate new requests of their own and so `SipSession.createRequest` results in an `IllegalStateException`. Similarly, requests created by applications acting as UAs cannot be proxied. However, the methods `getCallId`, `getLocalParty` and `getRemoteParty` must be implemented. `getLocalParty` returns the address of the caller (value of the From header of the initial request in the dialog) and `getRemoteParty` returns the address of the callee. This implies that Call-ID, From, and To must be associated with "proxy" `SipSessions` along with any application state.

8.2.2.2 When in the INITIAL SipSession State

When a UAC or UAS transitions from the early state to the initial state in Figure 6-2, the dialog state maintained in the `SipSession` is updated as follows (see RFC 3261 for the definition of these dialog state components):

- the remote target is reset to the remote URI
- the remote tag component is cleared
- the remote sequence number is cleared (e.g. set to -1)
- the route set is cleared
- the "secure" flag is set to false

As a consequence of these rules, requests generated in the same `SipSession` have the following characteristics:

- they all have the same Call-ID
- they all have the same From header field value including the same non-empty tag
- the CSeq of requests generated in the same `SipSession` will monotonically increase by one for each request created, regardless of the state of the `SipSession`
- all requests generated in the same `SipSession` in the initial state have the same To header field value which will not have a tag parameter
- `SipSession` objects in the initial state have no route set, the remote sequence number is undefined, the "secure" flag is false, and the remote target URI is the URI of the To header field value.

8.2.3 Creation of SipSessions

A big difference between dialogs and `SipSession`s is that whereas SIP requests may exist outside of a dialog (for example, OPTIONS and REGISTER), in the SIP Servlet API all messages belong to a `SipSession`.

SIP dialogs get created as a result of non-failure responses being received to requests with methods capable of setting up dialogs. The baseline SIP specification defines one such method, namely INVITE. Dialog handling is complicated by the fact that proxies may fork. This means a single request, for example an INVITE, can be accepted at multiple UASs, thus causing multiple dialogs to be created. In such cases, the UAC SIP servlet will see multiple `SipSession`s.

The relationship between `SipSessions` and SIP dialogs can be summed up as follows:

When an initial request results in one dialog being set up, the `SipSession` of the initial `SipServletRequest` will correspond to that dialog. When more than one dialog is established, the first one will correspond to the existing SipSession and for each subsequent dialog a new `SipSession` is created in the manner defined in 8.2.3.2 Derived SipSessions.

The rules for when `SipSessions` are created are:

- Locally generated initial requests created via `SipFactory.createRequest` belong to a new `SipSession`.
- Incoming initial requests belong to a new `SipSession`.
- Responses to an initial request that do not establish a dialog, belong to the “original” `SipSession` of the request.
- The first message that establishes a SIP dialog (for example, a 2xx to an initial INVITE request) is associated with the original `SipSession` as are all other messages belonging to that dialog.
- Subsequent messages that establish dialogs are associated with new `SipSessions` derived from the original `SipSession`, see 8.2.3.2 Derived SipSessions below.

When a proxying application receives a response that is associated with a new `SipSession` (say, because it is a second 2xx response to an INVITE request), the `getSession` method of that response returns a new derived `SipSession`. The original `SipSession` continues to be available through the original request object — itself available through the `getOriginalRequest` method on the `Proxy` interface.

8.2.3.1 Extensions Creating Dialogs

Extensions to the baseline SIP specification may define additional methods capable of establishing dialogs. The SIP event framework is one such extension [RFC 6665]. In the event framework, a matching NOTIFY request establishes the dialog instead of 2xx response. As per RFC 6665 a subscriber would wait until timer N expires for such a matching NOTIFY request. If timer N expires without receiving any matching NOTIFY, it is communicated to the application through the `SipErrorListener.noNotifyReceived` method. Note that since SIP session state directly track the transaction state, it will become “confirmed” on NOTIFY transaction rather than SUBSCRIBE. Similarly container will not establish the route set until NOTIFY request as specified in RFC 6665. If any application want to establish the dialog on SUBSCRIBE transaction as well (as per RFC 3265), they can choose to use `establishDialogOnSubscribeTransaction` element of `SipApplication` annotation.

The other SIP extension introducing a dialog creating method as of this writing is [RFC 3515] SIP REFER.

A SIP servlet container that "understands" an extension capable of establishing dialogs should create new derived `SipSession` objects for the second and subsequent dialogs created as a result of sending a single request capable of establishing multiple dialogs.

8.2.3.2 Derived SipSessions

A derived `SipSession` is essentially a copy of the `SipSession` associated with the original request. It is constructed at the time the message creating the new dialog is passed to the application. The new `SipSession` differs only in the values for the tag parameter of the address of the callee (this is the value used for the To header in subsequent outgoing requests) and possibly the route set. These values are derived from the dialog-establishing message as defined by the SIP specification. The set of attributes in the cloned `SipSession` will be the same as that of the original, if the `enableDerivedSessionAttributeCloning` element of the `SipApplication` annotation is set to true by the application, however the values are not cloned. Note that SIP servlet containers are required to enable this attribute cloning by default, if the application is a SIP Servlet 1.1 application. More information on Forking Context can be found in section 8.2.6 Forking and SipSessions.

New `SipSessions` corresponding to the second and subsequent 2xx responses (or 1xx responses with To tags) are available through the getSession method on the `SipServletResponse`. The "original" `SipSession` of the request continues to be available through the original request object.

In case of invite requests, the invite transaction that creates a new dialog is represented by the `InviteBranch` as explained in section 8.5 InviteBranch.

8.2.4 SipSession Lifetime

`SipSessions` do not have a timeout value of their own separate from that of the parent `SipApplicationSession`. Rather, when the parent session times out, all child protocol sessions time out with it.

In general, the lifetime of a `SipSession` can be controlled in the following ways:

1. The parent `SipApplicationSession` times out or is invalidated explicitly and this invalidates all child protocol sessions.
2. Application explicitly invalidates the `SipSession` using the invalidate() API.

3. Application marks the `SipSession` to be invalidated and the container invalidates it when the session is in the ready-to-invalidate state.

Any attempt to retrieve or store data on an invalidated `SipSession` causes an `IllegalStateException` to be thrown by the container as does any call to `createRequest`.

When a `SipSession` terminates, either because the parent application session timed out or because the `SipSession` was explicitly invalidated, the container MUST purge all state of that `SipSession` from its memory. In such a case, if a subsequent request or response belonging to the corresponding dialog is received, the container will reject the requests with 481 response and discard responses and ACK messages.

It is quite possible that different application instances on the same application path have different lifetimes. Containers handling subsequent requests and responses on a dialog corresponding to a partially invalidated application path would invoke all applications up to the first invalidated session instance and reject them if the session acts as an UA or proxy them statelessly if the session acts as a Proxy.

8.2.4.1 SipSession Invalidation

There are two ways in which a `SipSession` can be invalidated:

1. Explicit Invalidation Mechanism

2. Invalidate When Ready Mechanism

Once a `SipSession` object is invalidated either by the application or the container, it may no longer be used. On invalidation, the container MUST invoke the `sessionDestroyed()` callback on implementations of the `SipSessionListener` interface, if any exist.

8.2.4.1.1 Explicit Invalidation Mechanism

An application may invalidate a `SipSession` at any time using the `invalidate()` method. On explicit invalidation, the container MUST purge all state of that `SipSession` from its memory. The state would include both application state as well as container state such as the underlying transactions.

The `invalidate()` method called on a `SipSession` object will throw an `IllegalStateException` if that `SipSession` object has already been invalidated. Invalidating a `SipSession` using the `invalidate()` method causes the parent `SipApplicationSession` to have one less session.

8.2.4.1.2 Invalidate When Ready Mechanism

The explicit invalidation mechanism described above causes containers to clean up session state immediately and may suffer from the following drawbacks:

1. A broken or partially invalidated application path.
2. Orphaned sessions in network entities participating in the dialog.

This specification provides a mechanism for invalidation that applications can use to circumvent the above drawbacks called the "Invalidate When Ready" mechanism.

A `SipSession` is said to be in a ready-to-invalidate state only when it can be explicitly invalidated such that the SIP state is terminated cleanly across all the SIP network entities participating in the dialog. A `SipSession` is invalidated cleanly in this state, therefore application developers invalidating `SipSession`s in the ready-to-invalidate state will not cause sessions in the application path to be orphaned.

This specification introduces methods to help applications invalidate `SipSession`s cleanly across all network entities participating in a dialog. The methods introduced are:

1. `isReadyToInvalidate()` - returns true if the `SipSession` is in the ready-to-invalidate state and false otherwise. Applications can use this method to extend the lifetime of the parent `SipApplicationSession` if it times out before the `SipSession` is in the ready-to-invalidate state.
2. `setInvalidateWhenReady(boolean flag)` - allows applications to indicate to the container to notify it when the `SipSession` is in the ready-to-invalidate state. The container notifies the application using the `SipSessionListener.sessionReadyToInvalidate(SipSessionEvent se)` callback.
3. `getInvalidateWhenReady()` - returns true if the container will notify the application when the `SipSession` is in the ready-to-invalidate state.

An application willing to invalidate a `SipSession` cleanly could use the callback mechanism to perform any other application clean up before the `SipSession` gets invalidated by the container.

Servlets can register for `sessionReadyToInvalidate` notifications on the `SipSessionListener` interface. In the `sessionReadyToInvalidate` callback method, an application may choose to invalidate the session and perform any other cleanup activities. If the application does not explicitly invalidate the session in the callback or has not defined a listener, the container will invalidate the session.

Applications may call `setInvalidateWhenReady(false)` at any time to indicate to the container to not observe this session anymore. In such a case, the containers MUST not invalidate the session after the callback. Applications could then either rely on explicit invalidation mechanism or again call `setInvalidateWhenReady(true)`. This parallels the expiry callback mechanism defined above in 8.1.2.1 SipApplicationSession Timer Operation and SipApplicationSession Expiration.

For example: An application registers a `SipSession` for a callback using `session.setInvalidateWhenReady(true)` and handles the callback in the `SipSessionListener` as follows:

```
@SipListener
public class MySessionListener implements SipSessionListener {
 ...
 sessionReadyToInvalidate(SipSessionEvent se) {
   Session sess = se.getSession();

   // directs the container to stop observing this session and
   // not invalidate it when this callback returns.
   sess.setInvalidateWhenReady(false);
 }
...
}
```

The container determines the `SipSession` to be in the ready-to-invalidate state under any of the following conditions:

1. A dialog corresponding to a `SipSession` terminates when the `SipSession` transitions to the TERMINATED state. If derived sessions (sub-dialogs) were created, it is up to container implementations to declare the original `SipSession` to be in ready-to-invalidate state either when the original `SipSession` (primary dialog) terminates or when all the derived sessions (sub-dialogs) terminate.

2. When there are multiple usages, a dialog terminates only when all the usages in the dialog are terminated. Only when all the usages are terminated, SipSession will be in ready-to-invalidate state.

3. A `SipSession` transitions to the CONFIRMED state when it is acting as a non-record-routing proxy.

4. A `SipSession` acting as a UAC transitions from the EARLY state back to the INITIAL state on account of receiving a non-2xx final response (8.2.1 Relationship to SIP Dialogs, point 4) and has not initiated any new requests (does not have any pending transactions).

The container MUST NOT treat `SipSession` objects to be in the ready-to-invalidate state if the session has any underlying transactions in progress.

The firing of the parent `SipApplicationSession` timer influences the lifetime of a child `SipSession` and overrides the behavior of a `SipSession` marked with `invalidateWithReady(true)`. If the parent `SipApplicationSession` times out when it contains a child `SipSession` that is not yet ready to be invalidated, an application could detect it in the `sessionExpired` callback of the `SipApplicationSessionListener` and extend the lifetime of the `SipApplicationSession` using setExpires. Failing to do so will cause the `SipApplicationSession` expiry to forcefully invalidate the child `SipSession` even if it is not yet ready to be invalidated.

For example, to extend the lifetime of a `SipApplicationSession` when one of its child sessions is not yet in the ready-to-invalidate state, one could handle the callback in the `SipApplicationSessionListener` as follows:

```
@SipListener
public class MySipApplicationSessionListener
             implements SipApplicationSessionListener {
 ...
 sessionExpired(SipApplicationSessionEvent sase) {
   SipApplicationSession sas = sase.getApplicationSession();
   Iterator sessions = sas.getSessions("SIP");
   while (sessions.hasNext()) {
     SipSession ss = (SipSession) sessions.next();
     if (! ss.isReadyToInvalidate()) {
       sas.setExpires(extensionTime);
       return;
     }
   }
 }
 ...
}
```

If an application invokes any method on an invalidated `SipSession` object, the container SHOULD throw an `IllegalStateException`. Note that the methods introduced on the `SipSession` and `SipApplicationSession` classes in this specification throw `IllegalStateException` when invoked against invalid session objects. However, some of the existing methods (from v1.0) which did not throw this exception will continue to not throw it in this specification for backwards-compatibility reasons. `IsValid()` method is provided on both `SipApplicationSession` and `SipSession` interfaces to check whether the session object has been invalidated.

8.2.4.2 Important Semantics

This specification introduces the Invalidate When Ready mechanism for session invalidation. The invalidateWhenReady flag is true by default for both `SipApplicationSession`s and `SipSession`s for applications written compliant to v1.1 (or later) of this specification. This means that by default, all sessions are observed by the container and are invalidated automatically after they transition to the ready-to-invalidate state. This automatic cleanup helps in releasing resources as soon as possible and helps containers in managing their resources efficiently.

The `invalidateWhenReady` flag is false by default for both `SipApplicationSession`s and `SipSession`s for applications written compliant to v1.0 of this specification. Hence, sessions belonging to v1.0 applications will not be subject to the automatic session cleanup. By default, such v1.0 applications should see no change in session lifetime behavior on containers compliant with this specification. v1.0 applications that want the automatic session cleanup must explicitly call `setInvalidateWhenReady(true)` on each session. These applications may also implement the `sessionReadyToInvalidate()` callback on the session listeners.

If applications use the explicit invalidation mechanism for session invalidation, it is recommended that applications implemented a SIP dialog cleanup mechanism using the application composition function (18 Application Selection And Composition Model). The deployer may choose to always deploy a B2BUA application in the application path which implements the desired deployer policy as regards the duration of SIP dialogs. For example, in the simplest case, such an application could set a timer when a SIP call is first established through it. On expiration of this timer, this B2BUA application would tear down the call by appropriately sending termination SIP messages (BYE) on both of its dialogs.

8.2.5 Dialog Termination

An application may terminate a SIP dialog using `SipSession.terminateDialog()` method at any time. If the application acts as SIP user agent on this `SipSession`, and if this `SipSession` corresponds to an established SIP dialog, or a dialog-establishing transaction is pending, this method call instructs the container to send the appropriate SIP messages to terminate the dialog. Otherwise, this is a no-op. For this specification, SIP dialogs are only created by the INVITE, REFER, and SUBSCRIBE methods, and therefore `terminateDialog` only affects dialogs created by these methods and for any other session, the method call will result in an `IllegalStateException`.

Once this method is called, the application cannot send any more messages in this SipSession. Doing so will cause `IllegalStateException` to be thrown by the `SipServletMessage.send()` method. Furthermore, the container will not invoke the application's `SipServlet service()` or `doXXX()` methods from this point on.When the dialog is terminated, the app will be notified by the current `SipSessionListener.sessionReadyToInvalidate()` method.

An application may terminate all the dialogs belong to a forking context (see section 8.2.6 Forking and SipSessions) using `ForkingContext.terminateDialogs(AutomaticProcessingListener listener, SipSession... exclude)` method. This method will not terminate dialogs that belong to a proxy application. The container will also terminate all derived sessions that belong to the same `ForkingContext`, created after executing this method.

In certain cases, the application may wish to modify the outgoing SIP message that the container is about to send for the purpose of terminating a dialog. For example, the application may want to add a Reason header to a BYE, or a message content to a NOTIFY. There are also cases where the application may wish to be notified of incoming SIP messages. The application may provide a listener using the following method signature.
`SipSession.terminateDialog(AutomaticProcessingListener listener)`

`AutomaticProcessingListener` should not make changes to message that subvert the RFCs and container behavior in terminating the dialog. It is also recommended that applications do not throw any exception during the execution of `AutomaticProcessingListener`. Any exception thrown by the application will be ignored by the SIP servlet container.

Following is the definition of the `AutomaticProcessingListener` interface.

```
javax.servlet.sip.AutomaticProcessingListener extends
java.util.EventListener
```

Methods:

`void outgoingRequest(SipServletRequest request)` : This method is invoked before the container sends a SIP request. The application may modify the request in-place.

`void outgoingResponse(SipServletResponse response)`: This method is invoked before the container sends a SIP response. The application may modify the response in-place.

`void incomingRequest(SipServletRequest request)`: This method is invoked when the container receives a SIP request.

`void incomingResponse(SipServletResponse response)` : This method is invoked when the container receives a SIP response.

8.2.5.1 Terminating Proxy dialogs

As per RFC 3261, a SIP proxy MUST NOT create and send requests down an established dialog. However, some 3GPP applications need this ability (see 3GPP TS 24.229 section 5.2.8.1.2 Release of an existing session). To support this use-case, a separate method is provided so that application can terminate proxy dialogs. Given the way that this method breaks RFC 3261, this method should be used with caution and any application that uses it should be carefully tested. 3GPP specifications also require a way where the termination messages are sent for either one side of the proxy or both.

If the `SipSession` corresponds to an established SIP dialog, or a dialog-establishing transaction is pending, the following methods instructs the container to send the appropriate SIP messages to terminate the dialog.

`void terminateProxiedDialog(UAMode direction)`: Terminate the proxied dialog by sending appropriate messages in the direction specified. For example, if the direction is UAC, the container will send BYE message to terminate an INVITE dialog in the direction of UAC.

`void terminateProxiedDialog(UAMode direction, AutomaticProcessingListener listener)`: Terminate the proxied dialog by sending appropriate messages in the direction specified. Applications may intercept messages during termination.

`void terminateProxiedDialog()`: Terminate the proxied dialog by sending appropriate messages in both the directions.

`void terminateProxiedDialog(AutomaticProcessingListener listener)`: Terminate the proxied dialog by sending appropriate messages in both the directions. Applications may intercept messages during termination.

Note: An application should invoke the first two `terminateProxiedDialog` methods only when it find using some other means that other side of the specified `direction` will not respond. Any message on the other side may be ignored by the container.

8.2.5.2 Notes on Container behavior

The container implementations should use the following RFCs as guidelines for correct dialog termination behavior: RFC 3261, RFC 6665 (for SUBSCRIBE), RFC 3515 (for REFER), RFC 5057 (multiple dialog usages), RFC 5407 (races) and RFC 6026 (update to RFC 3261). The following sections provide examples of correct SIP servlet container behavior in terminating SIP dialogs based on these RFCs. In these examples, the terms UA, UAC and UAS should be understood to mean the SIP servlet container acting on behalf of an UA application that has called `SipSession.terminateDialog()`. Similarly, when a proxy dialog is being terminated using

`SipSession.terminateProxiedDialog()`, these terms corresponds to the direction(s) in which the messages are being sent.

8.2.5.2.1 INVITE dialog

- When a dialog is in early state, caller UA must send CANCEL to terminate the dialog. (While RFC3261 allows sending CANCEL or BYE , RFC 5407 sec 2 says UAC MAY send BYE, but not recommended. Hence this specification mandates the CANCEL request.) Callee UA can send error final response.
- Callee's UA MUST NOT send a BYE on a confirmed dialog until it has received an ACK for its 2xx response or until the server transaction times out. (See RFC3261 sec 15, and RFC6026)
- When a UA receives BYE, it MUST respond to any pending requests received for that dialog(RECOMMENDED 487) (RFC3261 sec 15.1.2). In this specification, the SIP Servlet container MUST use 487 response code.
- Not specified explicitly, but a UA should respond to any pending requests before sending BYE. In this specification, the SIP Servlet container MUST use 487 response code while responding to pending requests.
- Various races as described by RFC5407 e.g. requests can arrive after UA has sent BYE. They must be responded to.

8.2.5.2.2 SUBSCRIBE dialog

- SUBSCRIBE dialog can be created explicitly by subscriber sending SUBSCRIBE
- Can also be implicitly created by REFER
- A subscription is destroyed when a UA acting as notifier sends a NOTIFY request with a "Subscription-State" of "terminated". If there are no other dialog usage, the SIP dialog is terminated.
- A UA acting as a subscriber cannot explicitly terminate a subscription. A UA acting as subscriber may send a SUBSCRIBE request with an "Expires" header of 0 in order to trigger the notifier to send a NOTIFY request that destroys the subscription as above.

8.2.5.2.3 Multiple dialog usage

- It is also possible for a dialog to have multiple usages (RFC 5057). For the purpose of dialog termination, in order to terminate the dialog the UA must terminate each of the usages independently to terminate the entire dialog.

- The order in which messages are sent to terminate the usages is not important. For example, the UA may need to send BYE and NOTIFY to terminate the INVITE and SUBSCRIBE usages in a dialog respectively, and the UA may send these messages in either order.

Note: For Instant Messages, MESSAGE method does not create a dialog. For IM within a dialog, MSRP is used, and the dialog is established with INVITE.

8.2.6 Forking and SipSessions

The forking of SIP requests means that multiple dialogs can be established from a single request. When ever multiple dialogs are created due to forking a derived session as explained in section 8.2.3.2 Derived SipSessions is created from the original session. Such derived sessions that effectively represent such sibling dialogs belong to a forking context. `ForkingContext` interface help the application to navigate to all such derived sessions. Thus `SipSession` and its derived siblings belong to the same forking context.

A `ForkingContext` is created whenever the first session on a new dialog is created and it remains valid as long as at least one `SipSession` that belong to the forking context remains valid. Application can obtain an instance of `ForkingContext` using `SipSession.getForkingContext()` method.

When acting as a UA, it is possible that the application might want to send a new request on the same forking context, but on a different dialog. In this case such a request establish a new dialog. The only example of such a request, at the time of writing this specification, is the NOTIFY request as specified in RFC 6665. Sip servlet application can create a new request with the same forking Context using the method `ForkingContext.createRequest(String method)`.

8.2.6.1 Binding Attributes to a ForkingContext

A servlet can bind an object attribute into a `ForkingContext` by name. Hence `ForkingContext` is an attribute store. See more information on attributes and attribute store at Chapter 7, “Attributes”.

8.2.7 Session Keep Alive

SIP user agents and proxies depend on session terminating messages to end the session and clean up resources. When such a terminating message does not arrive, SIP servlet containers support a mechanism to determine whether the session should be kept alive or not as explained in this section. For instance, when a user agent fails to send a BYE message at the end of a session, or when the BYE message gets lost due to network problems, a call stateful proxy need to know when the session has ended.

RFC 4028 defines an extension that defines a keepalive mechanism for SIP sessions. UAs send periodic re-INVITE or UPDATE requests to keep the session alive. The interval for the session refresh requests is determined through a negotiation mechanism defined in RFC 4028. If a session refresh request is not received before the interval passes, the session is considered terminated. Both UAs are supposed to send a BYE, and call stateful proxies can remove any state for the call. SIP Servlet containers are required to support this keepalive mechanism as defined in this section.

8.2.7.1 Enabling Session Keep Alive

A SIP servlet can enable the session keep alive by setting appropriate keep alive preference to generate an initial session refresh request. SIP servlet can retrieve a `SessionKeepAlive.Preference` object using the method `SipServletMessage.getSessionKeepAlivePreference()`. It can then enable the keep alive by invoking `SessionKeepAlive.Preference.setEnabled(true)`. SIP servlet containers are required to set the headers (Session-Expires, Supported, Min-SE etc) as specified in RFC 4028 by using the values specified in `SessionKeepAlive.Preference` if the session keep alive is enabled.

If a UAC want the session timer to be applied to the session, UAC is required to enable the session keep alive by invoking `SessionKeepAlive.Preference.setEnabled(true)` before sending the initial session refresh request. The SIP servlet container is expected to set the Session-Expires to default value of 1800 seconds once the keep alive is enabled, if it has not been set previously.

For Proxy and UAS, session keep alive will be enabled, if the initial session refresh request contains Session-Expires header. If there is no Session-Expires header in the request, then UAS or Proxy may apply session timer to the session by enabling the session keep alive.

Once session keep alive is enabled, the SIP servlet container is required to follow the procedures specified in RFC 4028 for UAC, Proxy and UAS for activating session timer for the session.

8.2.7.2 Disabling Keep Alive

As mentioned in the previous section, the Proxy and UAS may receive a session refresh request with a Session-Expires header. In this case, by default, session keep alive is enabled. If a proxy or UAS does not want to take part in the session keep alive activity, then it may choose to disable the session keep alive by invoking `SessionKeepAlive.Preference.setEnabled(false)`. Note that disabling session keep alive does not mean that UAC will not send session refresh requests any more. It may continue to send session refresh requests if the session keep alive remains enabled.

8.2.7.3 Refreshing Sessions

When the UA that enabled session keep alive assumes the role of *refresher*, the SIP servlet container will schedule a keep alive timer. However container will not send a refresh request on its own. A SIP servlet can provide a refresh callback that will be executed by the container at the appropriate time for sending the session refresh request. The refresh callback need to implement `SessionKeepAlive.Callback` interface. Following is an example of such a callback.

```
request.getSessionKeepAlivePreference().setEnabled(true);
SessionKeepAlive skl = request.getSession().getKeepAlive();
skl.setRefreshCallback(new SessionKeepAlive.Callback() {
    @Override
    public void handle(SipSession session) {
        try {
            session.createRequest("UPDATE").send();
        } catch (IOException e) {
        }
    }
});
```

It is recommended that containers invoke the refresh callback once half the session expiration interval has elapsed. If the application sends a refresh request on its own by that time, the container will re-calculate the next session expiration time and invoke the refresh callback accordingly.

Application can remove the refresh callback by setting a null refresh task by invoking `SessionKeepAlive.setRefreshCallback(null);`

8.2.7.4 Expiring Sessions

When a Proxy or UA does not receive a session refresh request before the expiration interval, the SIP servlet container invokes the expiry callback. The expiry callback need to implement `SessionKeepAlive.Callback` interface as shown below.

```
request.getSessionKeepAlivePreference().setEnabled(true);
SessionKeepAlive skl = request.getSession().getKeepAlive();
```

```
        skl.setExpiryCallback(new SessionKeepAlive.Callback() {
            @Override
            public void handle(SipSession session) {
                try {
                    session.createRequest("BYE").send();
                } catch (IOException e) {
                }
            }
        });
```

8.2.7.5 422 Response

UAS and Proxy applications are expected to generate 422 response if it finds that the session expiration interval is too small. Similarly UACs are expected to handle the 422 response and retry the request as specified in RFC 4028.

8.2.8 The RequestDispatcher Interface

There is a need for a structuring mechanism, that allows applications to consist of multiple servlets that handle various parts of the application. In the case of HTTP, servlets may use a `RequestDispatcher` to forward a request to another servlet or to include the output of another servlet in its own response. SIP servlets can use the `RequestDispatcher` interface the same way—as a mechanism for passing a request or response to another servlet in the same application using the forward method and can be thought of as an “indirect method call”. (In the case of SIP, the name “`RequestDispatcher`” is a bit of a misnomer as it applies equally well to responses and requests). The include method has no meaning for SIP servlets.

8.2.9 The SipSession Handler

The `RequestDispatcher` interface provides one structuring mechanism for SIP servlets. Additionally, the `SipSession` interface has a notion of a handler—a reference to a servlet that is effectively the callback interface through which the container dispatches all incoming events related to that `SipSession` to the application.

When a servlet application creates a new initial request, a `SipSession` springs into existence. At this point, the container assigns the applications one servlet to be the handler for the newly created `SipSession`. This servlet selection is described in the 19 Mapping Requests To Servlets. The `SipSession` handler will be invoked to handle responses for all requests as well as new incoming requests. When a `SipSession` is created as part of processing an initial incoming request, the servlet object invoked becomes the initial handler for that `SipSession`.

Regardless of how a `SipSession` was created, the application may subsequently specify that a different servlet within the same application should become the handler for a particular `SipSession`. This is done by calling the `SipSession` method:

```
void setHandler(String name);
```

The argument is the same as in the call to `ServletContext.getNamedDispatcher`, i.e., it is a servlet name. Servlet names are typically associated with servlets through the deployment descriptor or using the `@SipServlet(name)` annotation.

8.2.10 Binding Attributes into a SipSession

A servlet can bind an object attribute into a `SipSession` by name. Hence `SipSession` is an attribute store. See more information on attributes and attribute store at Chapter 7, “Attributes”.

8.3 Last Accessed Times

The `getLastAccessedTime` method of the `SipApplicationSession` and `SipSession` interfaces allows a servlet to determine the last time a session was accessed before the current message was handled. A session is considered to be accessed when a message that is part of the session is handled by the servlet container. The last accessed time of a `SipApplicationSession` will thus always be the most recent last accessed time of any of its contained protocol sessions.

Whenever the last accessed time for a `SipApplicationSession` is updated, it is considered refreshed i.e., the expiry timer for that `SipApplicationSession` starts anew.

8.4 Accessing Messages from Session

It is possible to retrieve active requests from a `SipSession` as mentioned in section 6.7.1 Accessing Active Requests. It is also possible to access unacknowledged reliable provisional responses from the session as explained in section 6.8.2 Obtaining responses. Unacknowledged reliable provisional responses will be available from the session either until they are acknowledged or until the transaction times out.
Applications are often required to access the INVITE messages until

- At a UAC: an ACK has been sent or an error response received.
- At a UAS: an ACK has been received or an error response sent.

- At a proxy: a final response has been processed.

Application can access INVITE requests using `SipSession.getActiveInvite()` method according to the above time frame.

8.5 InviteBranch

An invite branch represents a branch of transaction resulting from an initial INVITE request. Since there is a session (or derived session) that represents each branch of the forked dialog, it represents the initial INVITE transaction of a session. An application may obtain the request that initiated the transaction and the final response that corresponds to the branch from `InviteBranch`.

Application may obtain an `InviteBranch` using `SipSession.getActiveInviteBranch` method. This method will return the `InviteBranch` as long as it is active. The `InviteBranch` is active under the following conditions.

- At a UAC: until ACK has been sent or an error response received
- At a UAS: until ACK has been received or an error response sent.
- At a proxy: until a final response has been processed

Application might send or receive responses on different forked branches of the initial INVITE request. It is possible to access the `InviteBranch` object from the responses using `SipServletResponse.getInviteBranch()` method as long as the invite branch is active.

Application may access the original request that created the transaction and the final response in the transaction branch from the `InviteBranch` object. It can also create new responses on a specific branch using `InviteBranch` interface.

When acting as UAS, an application might want to send a response on a new dialog. In this case, application may create a new `InviteBranch` on the request using `SipServletRequest.createInviteBranch()` method.

8.6 Important Session Semantics

8.6.1 Message Context

When passed to applications, SIP servlet request and response objects are always associated with a `SipSession` object which, in turn, belong to a `SipApplicationSession`. This specification refers to the combination of application, `SipApplicationSession`, and `SipSession` as the message context. When routing subsequent requests, containers must determine which applications to invoke and also the context in which to invoke them. When multiple applications execute on the same request, they will do so in different message contexts—each application has its own session objects which are independent of those of other applications. Application data placed into a session object by application A is not accessible to application B.

When applications are invoked to process subsequent requests, the message context will be identical to that of the initial request. Suppose, for example, that an application is invoked to handle an INVITE and proxies it with record-routing enabled. When a BYE for the same dialog is received, the same application is invoked and in the same context, that is, `getApplicationSession` and `getSession` are required to return the same session objects when invoked on the INVITE and BYE requests as well as on corresponding response objects.

Note that while logically all SIP messages passed to applications have `SipApplicationSession` and `SipSession` objects associated with them, for performance reasons, containers may choose to defer creation of session objects when applications cannot observe the difference.

8.6.2 Threading Issues

Containers may invoke a single SIP servlet application in more than one concurrent thread, and these threads may have active access to the same `SipSession` and `SipApplicationSession` objects at the same time. 17.3 SIP Servlet Concurrency explains ways to write portable applications without concurrency issues. If an application uses an unspecified concurrency control and container implementation does not provide thread safety, the application developer has the responsibility for synchronizing access (possibly with container-specific means) to these session objects and any attributes and objects held by these session objects, as appropriate.

Alternatively, container implementations may provide a specific threading model (pertaining to access to session objects), an explicit locking API or implicit serialized access semantics. In such cases, the container behavior should be documented in sufficient detail to facilitate application thread safety and porting of applications between different vendors.

8.6.3 Distributed Environments

For an application marked as distributable the container must be able to handle all objects placed into `SipSession,ForkingContext` and `SipApplicationSession` (generically called sessions in this section) using the `setAttribute` method according to the following rules.

- The container MUST accept objects that implement the `Serializable` interface.

- The container MUST accept object of the type `SipServletMessage`.
- The container MAY choose to support storage of other objects in the `SipSession`, such as references to Enterprise Java Beans and transactions.
- Migration of sessions will be handled by container-specific facilities.

The servlet container MAY throw an `IllegalArgumentException` if an object is placed into the sessions that is not `Serializable` or for which specific support has not been made available. The `IllegalArgumentException` MUST be thrown for objects where the container cannot support the mechanism necessary for migration of a session storing them.

These restrictions mean that the developer is ensured that there are no additional concurrency issues beyond those encountered in a non-distributed container.

The container provider can ensure scalability and quality of service features like load-balancing and failover by having the ability to move a session object and its contents from any active node of the distributed system to a different node of the system.

If distributed containers persist or migrate sessions to provide quality of service features, they are not restricted to using the native JVM serialization mechanism for serializing sesssion and their attributes, however the containers MUST provide consistent environment after de-serialization as one would expect Java serialization mechanism to provide. Developers are not guaranteed that containers will call `readObject()` and `writeObject()` methods on session attributes if they implement them, but are guaranteed that the `Serializable` closure of each attribute will be preserved. Defining closure around individual attributes mean that each individual attribute can be saved independently without having to save the entire `SipSession` or `SipApplicationSession` on an attribute addition or update, of course the shared objects accross the attributes shall be referenced as in normal Java serialization mechanism.

Containers MUST notify any session attributes implementing the `SipSessionActivationListener` and `SipApplicationSessionActivationListener` during migration of a session. They MUST notify listeners of passivation prior to serialization of a session, and of activation after deserialization of a session when such a migration takes place.

Developers writing distributed applications should be aware that since the container may run in more than one Java VM, the developer cannot depend on static or instance variables for storing application states. They should store such states using an EJB or a database.

In addition to this the containers MUST also ensure that the `Serializable` "info" object associated with the `ServletTimer` is also migrated or persisted alongside the sessions if the `ServletTimer` is chosen to be persistent.

9 SIP Servlet Applications

A SIP servlet application is a collection of servlets, class files, and other resources that comprise a complete application on a SIP server. The SIP application can be bundled and run on multiple containers from multiple vendors.

By default, an instance of a SIP application must run on one VM at any one time. This behavior can be overridden if the application is marked as "`distributable`" via its deployment descriptor or via the `@SipApplication(distributable)` annotation. An application marked as distributable must obey a more restrictive set of rules than is required of a normal servlet application. These rules are set out throughout this specification.

9.1 Relationship with HTTP Servlet Applications

The directory structure used for SIP servlet applications is very similar to the one defined for HTTP servlets. In particular, the deployment descriptor, class files, and libraries of SIP servlet applications reside in the WEB-INF/ directory. This allows converged HTTP and SIP servlet applications to be packaged in a single archive file and deployed as a unit without having any part of the SIP application be interpreted as content to be published by the Web server.

9.2 Relationship to ServletContext

The servlet container must enforce a one-to-one correspondence between a servlet application and a `ServletContext`. A `ServletContext` object provides a servlet with its view of the application. This is true for converged SIP and HTTP applications. In addition, all servlets within an application see the same `ServletContext` instance.

9.3 Elements of a SIP Application

A SIP application may consist of the following items:

- Servlets
- Utility classes
- Static resources and content (text and speech announcements, configuration files, etc.)
- Descriptive meta information that ties all of the above elements together

9.4 Deployment Hierarchies

This specification defines a hierarchical structure used for deployment and packaging purposes that can exist in an open file system, in an archive file, or in some other form. It is recommended, but not required, that servlet containers support this structure as a runtime representation.

9.5 Directory Structure

A SIP application exists as a structured hierarchy of directories. As mentioned above, for compatibility with the HTTP servlet archive file format, the root of this hierarchy serves as the document root for files intended to be published from a Web server. This feature is only used for combined HTTP and SIP servlet applications.

A special directory exists within the application hierarchy named “WEB-INF”. This directory contains all things related to the application that are not in the document root of the application. For SIP-only applications, everything typically resides under WEB-INF. The WEB-INF directory is not part of the application’s public document tree. No file contained in the WEB-INF directory may be served directly to a client by the container. However, the contents of the WEB-INF directory are visible to servlet code by using the `getResource()` and `getResourceAsStream()` method calls on the `ServletContext`. Any application-specific configuration information that the developer needs to access from servlet code but does not wish to expose to a web client may be placed under this directory. Since requests are matched to resource mappings case-sensitively, client requests for ‘/WEB-INF/foo’, ‘/WEb-iNf/foo’, for example, should not result in contents of the web application located under /WEB-INF being returned, nor any form of the directory listing thereof.

The contents of the WEB-INF directory are:

- The /WEB-INF/sip.xml deployment descriptor.
- The /WEB-INF/classes/* directory for servlet and utility classes. The classes in this directory are available to the application classloader.
- The /WEB-INF/lib/*.jar area for Java Archive files. These files contain servlets, beans, and other utility classes useful to the web application. The web application classloader can load class from any of these archive files.

The application classloader must load classes from the WEB-INF/ classes directory first, and then from library JARs in the WEB-INF/ lib directory.

9.6 Application Names

A SIP servlet application referred to in this specification is analogous to a Java EE module. The Java EE platform specification mandates that each Java EE module has a name. Unless specified in the deployment descriptor, for a standalone module, module name is the base name of the module file (.war,.sar, etc.) with any extension removed and with any directory names removed. Similarly, each application (e.g., EAR file) in a Java EE application server has a unique application name, which again can be overridden in the application deployment descriptor. For a standalone module, application name is the same as the module name. Refer to section 8.1.1 of the Java EE specification for details about how the application name and module names are determined.

SIP Servlet specification requires that a SIP servlet application is identified by the application name for standalone modules. SIP servlet applications packaged in an EAR file are identified by `application-name/module-name`.

If an application specifies `<app-name>`, under the `<sip-app>` element in the sip.xml deployment descriptor file of an application, or using `name` element of `@SipApplicationName` annotation as specified in SIP servlet 1.1, containers continue to use that for identifying the SIP servlet application. This ensures backward compatibility.

Further, a SIP servlet application may access the name of the current application by using the pre-defined JNDI name `java:app/AppName`. A component may access the name of the current module by using the pre-defined JNDI name `java:module/ModuleName`. Both of these names are represented by String objects.

For more details about Application Names and Module Names, refer to *Java EE Specification*.

Listing 9-1 Example of sip.xml file overriding module-name

```
<sip-app>
```

```
    <module-name>voicemail</module-name>
    ...
</sip-app>
```

If the application is specified by annotation, chapter 22 Java Enterprise Edition Container provides the procedures for determining the name of such applications.

9.6.1 Example Application Directory Structure

The following is a listing of all the files in a sample SIP servlet application:

```
/WEB-INF/sip.xml
/WEB-INF/wakeup.wav
/WEB-INF/lib/foo.jar
/WEB-INF/classes/WakeupServlet.class
```

The following is the same example but with an HTTP servlet component included:

```
/wakeup.html
/register.html
/WEB-INF/sip.xml
/WEB-INF/web.xml
/WEB-INF/wakeup.wav
/WEB-INF/lib/foo.jar
/WEB-INF/classes/RegisterWakeupCall.class
/WEB-INF/classes/WakeupServlet.class
```

The second example includes separate SIP and HTTP deployment descriptors: sip.xml and web.xml, respectively. Resources not under WEB-INF/ are part of the content served from the HTTP part of the converged application.

9.7 Servlet Application Archive File

Servlet applications can be packaged and signed into a Servlet Archive format (SAR) file using the standard Java Archive tools. For example, a click-to-dial application might be distributed in an archive file called click2dial.sar.

When packaged this way, a META-INF directory is present that contains information useful to Java Archive tools. This directory cannot be directly served as content by the container in response to a web client's request, though its contents are visible to servlet code via the `getResource()` and `getResourceAsStream()` calls on `ServletContext`.

Such an archive deployed to a Java EE compliant servlet container may reference a JAR file or directory by naming the referenced JAR file or directory in a Class-Path header in the referencing JAR file's Manifest file as per chapter 8*(Application Assembly and Deployment)*, of the Java EE Specification. The referenced JAR file or directory is named using a URL relative to the URL of the referencing JAR file.

Note that the SAR archive format is modeled after the WAR archive format. Since SAR archives can contain the web application components, a WAR archive should also be able to contain the SIP application components. It should not matter to the converged container since the contents of the archive determine what components are in them. This specification establishes the equivalence of SAR and WAR archive formats in the context of SIP Servlet containers. For the purposes of discussion in this specification, the archive file is called Servlet Archive or SAR for short.

9.8 SIP Application Configuration Descriptor

The following types of configuration and deployment information are in the SIP application deployment descriptor (the Deployment Descriptor schema is presented in 21 Deployment Descriptor):

- ServletContext init parameters
- Session configuration
- Servlet definitions
- Security

9.9 Dependencies On Extensions

When several applications use the same code or resources, they are typically installed as library files in the servlet container. These files are often common or standard API that can be used without portability being sacrificed. Files used by only one or a few applications are made part of the servlet application to be available for access. Application developers need to know what extensions are installed on a servlet container, and containers need to know what dependencies on libraries that servlets in a SAR/WAR may have in order to preserve portability.

Servlet containers should have a mechanism by which servlet applications can learn what JAR files containing resources and code are available, and for making them available to the application. Containers should provide a convenient procedure for editing and configuring library files or extensions.

It is recommended that application developers provide a META-INF/MANIFEST.MF entry in the SAR/WAR file listing extensions, if any, needed by the SAR/WAR. The format of the manifest entry should follow standard JAR manifest format. In expressing dependencies on extensions installed on the servlet container, the manifest entry should follow the specification for standard extensions defined at http://docs.oracle.com/javase/7/docs/technotes/guides/extensions/versioning.html.

Servlet containers should be able to recognize declared dependencies expressed in the optional manifest entry in a SAR/WAR file, or in the manifest entry of any of the library JARs under the WEB-INF/lib entry in a SAR/WAR. If a servlet container is not able to satisfy the dependencies declared in this manner, it should reject the application with an informative error message.

9.10 Servlet Application Classloader

The classloader that a container uses to load a servlet in an application archive must not allow the archive to override J2SE or Java servlet API classes. It is further recommended that the loader not allow servlets in the archive access to the servlet container's implementation classes.

It is recommended, however, that the application classloader be implemented so that classes and resources packaged within the SAR/WAR are loaded in preference to classes and resources residing in container-wide library JARs.

Section 8.3.1*(Web Container Class Loading Requirements)* of the Java EE specification explain the classloading requirements of the web container. These requirements are applicable to SIP Servlet applications also, irrespective of whether the application is packaged as a SAR file or WAR file. The class visibility rules explained in section 8.3.1 of Java EE 7 specification for Web applications (WAR files) also apply to SIP Servlet applications.

9.11 Replacing a Servlet Application

A server should be able to replace an application with a new version without restarting the container. When an application is replaced, the container should provide a robust method for preserving session data within that application.

9.12 Servlet Application Environment

Java EE defines a naming environment that allows applications to easily access resources and external information without the explicit knowledge of how the external information is named or organized.

As servlets are an integral component type of Java EE, provision has been made in the SIP servlet application deployment descriptor for specifying information allowing a servlet to obtain references to resources and enterprise beans. The deployment elements that contain this information are:

- `env-entry`
- `ejb-ref`
- `resource-ref`

The `env-entry` element contains information to set up basic environment entry names relative to the `java:comp/env` context, the expected Java type of the environment entry value (the type of object returned from the JNDI lookup method), and an optional environment entry value. The `ejb-ref` element contains the information needed to allow a servlet to locate the home interfaces of an enterprise bean. The `resource-ref` element contains the information needed to set up a resource factory.

The requirements of the Java EE environment with regards to setting up the environment are described in chapter 5*(Resources,Namingand Injection)* of the Java™ Platform, Enterprise Edition (Java EE) Specification. Servlet containers that are not part of a Java EE compliant implementation are encouraged, but not required, to implement the application environment functionality described in the Java EE specification. If they do not implement the facilities required to support this environment, upon deploying an application that relies on them, the container should provide a warning.

Since the Servlet specification introduced the usage of annotations-based resource injection and this specification is based on Servlet specification v3.1, the annotations defined in Servlet specification v3.1 are supported, like `@Resource`, `@EJB`. See 22.3.1 Servlet alignment for details.

10 Application Listeners and Events

Java Servlet Specification Version 2.3 introduced the notion of application listeners [Servlet API, chapter 10]. The SIP Servlet API requires full support for application listeners and events as defined in that document, except for the HTTP-specific events defined in package `javax.servlet.http`. The servlet context events defined in `javax.servlet` must be supported. Additionally, this specification defines some SIP-specific events for which support is also required.

For full details on application listeners and events, see [Servlet API, chapter 10].

10.1 SIP Servlet Event Types and Listener Interfaces

The following describes the SIP-specific event types and their associated listener interfaces:

- Listeners on `AttributeStore`:
  - `javax.servlet.sip.AttributeStoreListener`: Implementations of this interface receive notifications when attributes are added, removed, or replaced from an `AttributeStore`.
  - `javax.servlet.sip.AttributeStoreBindingListener`: Attributes implementing this interface receive notifications when they are bound or unbound from an `AttributeStore.`
- Listeners on `SipApplicationSession`:
  - `javax.servlet.sip.SipApplicationSessionListener`: Implementations of this interface receive notifications when a `SipApplicationSession` is created, destroyed, timed out, or in the ready-to-invalidate state.

  - `javax.servlet.sip.SipApplicationSessionActivationListener`: Implementations of this interface receive notifications when a `SipApplicationSession` is activated or passivated.

- Listeners on `SipSession`:
  - `javax.servlet.sip.SipSessionListener`: Implementations of this interface receive notifications when a `SipSession` is created, destroyed, or in the ready-to-invalidate state.
  - `javax.servlet.sip.SipSessionActivationListener`: Implementations of this interface receive notifications when a `SipSession` is activated or passivated.
- Automatic Processing listener:
  - `javax.servlet.sip.AutomaticProcessingListener`: Implementations of this interface receive notification when messages are sent or received after the application relinquishes control to the container.
- Error listener:
  - `javax.servlet.sip.SipErrorListener`: Implementations of this interface receive notification when an expected ACK, PRACK, or dialog establishing NOTIFY is not received.
- Timer listener:
  - `javax.servlet.sip.TimerListener`: Implementations of this interface receive notification when a `ServletTimer` has fired.
- SipServlet listener:
  - `javax.servlet.sip.SipServletListener`: Implementations of this interface receive notification on `SipServlet` initialization.

11 Timer Service

The timer service is a container-provided service that allows applications to schedule timers and to receive notifications when timers expire. Timers are managed by the container like other resources, and may optionally be persistent, in which case they are stored along with session data. Timers can be scheduled to expire once after a specified time, or repeatedly at specified intervals.

The timer support consists of three interfaces:

- `TimerService`– Used when creating timers.
- `ServletTimer` – Represents scheduled timers and is passed to callbacks.
- `TimerListener` – The callback interface implemented by the application and invoked by the container on timer expiration.

11.1 TimerService

The `TimerService` interface is implemented by containers and is made available to applications as a `ServletContext` parameter with the name `javax.servlet.sip.TimerService`. The `TimerService` interface includes the following methods:

```
ServletTimer createTimer(SipApplicationSession appSession,
                         long delay,
                         boolean isPersistent,
                         java.io.Serializable info);

ServletTimer createTimer(SipApplicationSession appSession,
                         long delay,
                         long period,
                         boolean fixedDelay,
                         boolean isPersistent,
                         java.io.Serializable info);
```

`ServletTimer` objects are associated with an application session. The application can store data in the application session and retrieve it later when the timer fires. The `getTimers` method of the `SipApplicationSession` interface returns a `java.util.Collection` containing all `ServletTimer` objects currently scheduled and associated with that session. Similarly, the `getTimer(String id)` method on `SipApplicationSession` returns the specific `ServletTimer` object that is currently scheduled and is associated with that session.

The following describes the arguments for the `createTimer` method:

- `appSession` – the application session that the new `ServletTimer` will be associated with
- `delay` – the delay, in milliseconds, before `ServletTimer` expires the first time
- `period` – the interval, in milliseconds, between successive timer expirations
- `fixedDelay` – specifies whether a repeating timer is fixed-delay or fixed-rate. The semantics are similar to those of `java.util.Timer`. In both cases, the repeating timer expires at regular intervals.However, the fixed-delay rescheduling of the repeating timer ignores any “lateness” in previous expirations, whereas fixed-rate timers are rescheduled based on absolute time.
- `isPersistent` – if true, the `ServletTimer` should be reinstantiated when the server is shut down andis subsequently restarted. During the restart, the container calls `TimerInterface` for a timer that has expired during the shutdown. The `SipApplicationSession` associated with the `ServletTimer` should be persistent.
- `info` – the application information to deliver along with the timer expiration. This is useful for determining the significance of a timer expiration in applications that set multiple timers per application session.

11.2 ServletTimer

The `ServletTimer` interface represents scheduled timers. The application session, timer ID, and serializable information object can be retrieved from the `ServletTimer` in the expiration callback. The `ServletTimer` interface also allows applications to cancel timers and obtain information regarding last and next scheduled expiration time.

The `ServletTimer` interface includes the following methods:

```
SipApplicationSession getApplicationSession();
java.io.Serializable getInfo();
String getId();
long scheduledExecutionTime();
long getTimeRemaining();
void cancel();
```

11.3 TimerListener

The SIP Servlet container notifies `TimerListener` about the expiration of timers. This interface includes a single method:

```
void timeout(ServletTimer timer);
```

Applications using timers must implement the `TimerListener` interface and must declare the implementation class in a listener element of the SIP deployment descriptor (as described below) or use the `@SipListener` annotation (as described in 22.3.4 @SipListener Annotation). For example, an application might include a class `com.example.MyTimerListener` that implements `javax.servlet.sip.TimerListener`. It would then declare this listener in the SIP deployment descriptor as follows:

```
<listener>
  <listener-class>com.example.MyTimerListener</listener-class>
</listener>
```

Only one `TimerListener` implementation can be declared in the deployment descriptor.

12 Proxying

One important function of SIP is the ability to route requests. That is, for incoming requests to decide which destination or destinations should receive the request. The ability to proxy requests is essential to many SIP services. In some cases, the service may have a choice between proxying and redirecting but many services require proxying because they need to see responses and/or stay on the signaling path.

One of the most important differences between the HTTP and SIP Servlet APIs is that HTTP Servlets execute on origin servers only and are concerned only with responding to incoming requests, whereas SIP servlets are typically located on proxy servers and must be able to proxy incoming requests as well as respond to them directly.

Proxying is initiated and controlled via a `Proxy` object obtained from an incoming `SipServletRequest` and its associated `ProxyBranch` object. There is at most one `Proxy` object per SIP transaction, meaning that `SipServletRequest.getProxy()` returns the same `Proxy` instance whenever invoked on the original request object or on any other message belonging to that transaction.

12.1 Parameters

A number of `Proxy` parameters control various aspects of the proxying operation:

- **`recurse:`** flag specifying whether or not the servlet engine automatically recurses. If recursion is enabled, the servlet engine automatically attempts to proxy to contact addresses received in redirect (3xx) responses. The default value is true.

- **`recordRoute:`** flag controlling whether or not the application stays on the signaling path for this dialog. This should be set to true to allow applications to see subsequent requests belonging to the dialog. The default value is false.
- **`parallel:`** flag specifying whether to proxy to multiple destinations in parallel (true) or sequentially (false). In the case of parallel search, the server may proxy the request to multiple destinations without waiting for final responses to previous requests. The default value is true.
- **`supervised:`** whether the servlet is invoked to handle responses. Note that setting the supervised flag to false affects only the transaction to which the `Proxy` object relates. The default value is true.
- **`proxyTimeout:`** the timeout for the proxy, in general. If the proxy is a sequential proxy, this value behaves like the `sequential-search-timeout` which is deprecated since v1.1. If the proxy is a parallel proxy, this timeout acts as the timeout for the entire proxy i.e each of its parallel branches before it starts to send out CANCELs waiting for final responses on all INVITE branches and sends the best final response upstream. The default value is the value of the `proxy-timeout` element of the deployment descriptor or a container-specific value, if this is not specified.
- **`addToPath:`** flag controlling whether the application adds a Path header to the request. Thereby, adding itself into the Path created by the REGISTER requests as per RFC3327.

The `recordRoute` flag may be set only before the first call to `Proxy.proxyTo` or `Proxy.startProxy()`. Any attempt to set it afterwards results in an `IllegalStateException` being thrown. Applications may modify the `recurse`, `parallel`, and `supervised` flags as well as the `proxyTimeout` parameter while a proxying operation is in progress.

Note: Deprecation of Stateless Proxy in SIP Servlet Specification:

The `Proxy` implemented on containers compliant with this specification MUST always be transactional stateful, and they MUST NOT allow an application to change the transactional behavior of Proxy. Therefore, both `Proxy.setStateful()` and `Proxy.getStateful()` are deprecated in this specification.

For backward compatibility purposes:

- `Proxy.setStateful(boolean stateful)` now does nothing.
- `Proxy.getStateful` now always returns true.

Though stateless proxying by applications is deprecated in the 1.1 version of this specification, a proxy implemented on containers compliant with this specification MUST forward messages statelessly under certain conditions as described in [RFC 3261, Section 16.7, point 10 and 16.10]

12.2 Operation

There are two ways in which an application can perform a proxying operation by using the `Proxy` object obtained from an incoming `SipServletRequest`:

1. By calling the `proxyTo()` method on the `Proxy` object and passing the `URI`(s) to proxy the request to.
2. By creating a `ProxyBranch` object(s) from the `Proxy` object and then invoking the `startProxy()` method on `Proxy`.

When creating the `Proxy` object for an incoming INVITE request, the server also sends a 100 provisional response upstream. This is done to stop retransmissions from the client. The server MAY choose to send the 100 provisional response irrespective of the application type for the same reason.

Before proxying the request, the application may modify the `Proxy` parameters as appropriate and may also modify the request object itself. This includes adding or removing headers but the proxy MUST NOT add to, modify, or remove the message body as per [RFC 3261 16.6.1].

The `proxyTo()` method is overloaded to take either a single `URI` or a `List` of `URI`s specifying downstream destination(s). If the list of `URI`s passed to the `proxyTo()` method contains any duplicates (based on the definition of equality for the `URI` type), the proxy MUST ignore proxying to the duplicate `URI`s as per [RFC 3261 16.5]. Compliant servlet engines are required to be able to handle SIP and SIPS `URI`s and may know how to handle other `URI` schemes, for example, `Tel URI`s.

For each `URI` passed to it in one of the `proxyTo` methods, the container creates a new branch on which the request is proxied. The SIP Servlet container replaces the proxied request's request `URI` with that of the specified destination `URI` and is routed either based on that modified request `URI`, or based on the top Route header field value, if one is specified.

Until a final response is forwarded upstream, the application may invoke the `proxyTo` method any number of times to add additional addresses to the set of destinations the container should proxy to.

Applications that wish to stay on the signaling path but which do not perform any routing decisions or otherwise influence the call setup may proxy with record-routing enabled without changing the request `URI`. Specifically, it will do the following:

```
public void doInvite(SipServletRequest req) {
  ...
  Proxy p = req.getProxy();
  p.setRecordRoute(true);
  p.proxyTo(req.getRequestURI());
  ...
}
```

The request is proxied either to other applications according to the application composition mechanism or towards the external destination specified by the request `URI`.

Note that the SIP Servlet programming model is asynchronous. Therefore, the application code may not process a request in the context of the service method upcall, but save the request instead and act on it asynchronously. For example, it is perfectly fine for an application to obtain the `Proxy` object from a request in the context of a `service` method, but defer the act of starting the `Proxy` until later. For instance, trigger it by an application timer expiration. In other words, there is no expectation that the `Proxy` is started by the application code in the context of the `service` method.

12.2.1 Proxy Branches

A `ProxyBranch` object represents a proxy branch. Explicitly creating proxy branches, modifying them individually, and then starting the proxy process is another mechanism of proxying. The `ProxyBranch` object can be created from the `Proxy` object by using the following method:

```
java.util.List<ProxyBranch> createProxyBranches(java.util.List<? extends URI>
targets)

For example:
..
public void doInvite(SipServletRequest req) {
...
  Proxy p = req.getProxy();
  p.setRecordRoute(true);
  p.createProxyBranches(targetList);
  p.startProxy();
...
}
```

The effect of the above operation is the same as `p.proxyTo(targetList);` the proxy branch is not used directly by the proxy application in most cases except when the application requires having some branch-specific changes to the request.

Only the following operations are allowed on the `ProxyBranch` and the associated `SipServletRequest` object:

1. push different route headers to the associated `SipServletRequest` with the proxy branch.
2. cancel a proxy branch by issuing a CANCEL request. The mechanism that the container uses to resolve the race condition of receiving a final response of the branch and mechanism of not sending

CANCEL until the provisional response is received are same as defined for `Proxy.cancel()` method which cancels all the proxy branches and all branches created recursively.

3. set different record-route, path parameter, recurse flag, or proxy timeout values for the branch.
4. add/remove non-system headers from `SipServletRequest` before the request is proxied.
5. any other method invoked on the `SipServletRequest` object (like `send()` or `createCancel()`) that is not relevant for this context MUST throw `IllegalStateException`.

Until a final response is forwarded upstream, the application may invoke the `createProxyBranches()` and `startProxy()` methods any number of times to add additional addresses that the container should proxy to.

The `createProxyBranches()` method explicitly creates and the `proxyTo()` method implicitly creates `ProxyBranch`(es). These proxy branches can be retrieved from the `Proxy` object by using `ProxyBranch Proxy.getProxyBranch(URI)` on the `Proxy` interface. These are the top level branches created by user code providing a single `URI` or a list of `URI`s. Further, `java.util.List<ProxyBranch> Proxy.getProxyBranches()` on the `Proxy` interface returns all of the top level branches associated with `Proxy`.

Additionally, any `ProxyBranch` might recurse if the recurse flag is true for the proxy/branch and the branch received the 3xx class response with alternate Contact headers. It is possible to retrieve the recursed branches by using the `ProxyBranch` method `java.util.List getRecursedProxyBranches()`.

An application may choose to pass an implementation of the `ProxyBranch.Callback` interface to `Proxy.startProxy(ProxyBranch.Callback timeoutCallback)` to receive a callback when a `ProxyBranch` times out. Application may increase the timeout further by using the `ProxyBranch.setProxyBranchTimeout(int seconds)` method during the call back and thus extend the life time of the proxy branch.

12.2.2 Pushing Route headers

As mentioned in the SIP specification, a proxy may have a local policy that mandates that a request visit a specific set of proxies before being delivered to the destination [RFC 3261, section 16.6]. This is done by pushing a set of SIP/SIPS `URI`s to the Route header field of the request.

This is the purpose of the `pushRoute` method of the `SipServletRequest` interface. The argument identifies a proxy that should be visited before the request reaches its final destination. If only one Route header field value is added, the container may choose to not actually push a Route header field value but rather to use the alternative of bypassing the usual forwarding logic, and instead just send the request to the address, port, and transport for the proxy specified in the single `pushRoute` call. This is subject to the constraints specified in [RFC 3261].

12.2.3 Sending Responses

Applications may generate informational responses of their own before or during a proxy operation. This is done the same way it is done when acting as a UAS that is downstream to a proxy.

After a request is proxied, the application does not usually generate a final response of its own. However, there are cases where being able to do so is useful and thus is allowed with some constraints. As long as no final response is sent upstream, the application may create and send its own final response. The container behaves as if the final response was received from a (virtual) branch of the proxy transaction with the following qualifications:

- A virtual branch applies only in the context of a proxy.That is, `request.getProxy()` MUST be called before sending any response from the virtual branch upstream. After the application calls `request.getProxy()`, the proxy could send the response upstream either before or after calling `proxyTo()`. If the response is sent upstream before calling `request.getProxy()`, the subsequent proxying operation with `Proxy.proxyTo()` MUST throw an `IllegalStateException`.
- A virtual branch applies only for responses to initial requests.
- If it's a 2xx response, it is sent upstream immediately without notifying the application of its own response.
- A non-2xx final response generated while the proxy transaction has outstanding branches contributes to the response context as any response received from a real branch. If it's eventually selected as the best response, the container performs the usual best-response callback.
- If the best response received was a non-2xx and the application generated its own final response in the `doResponse` callback (be it a 2xx or non-2xx), the response is sent immediately without invoking the application again for its own generated response.

This last item allows applications to create, for example, a different error response from the one chosen by the container as the best response. Note that if an application does generate its own final response when passed the best response received, it cannot also proxy to more destinations.

These rules are designed to guarantee that SIP Servlet containers, when observed from the outside, do not violate the SIP specification.

From the container perspective, when a response from the virtual branch is sent upstream, a derived `SipSession` is created for the virtual branch as per 8.2.3.2 Derived SipSessions. The original

`SipSession` associated with the incoming request corresponds to the `Proxy` while the derived one represents the virtual UAS. Containers MUST restrict to creating only one virtual branch for each `Proxy`.

If the final response sent upstream was generated by the application itself, the container must update the `SipSession` state as if it was a UAS (which in fact it is).

Alternative to sending a response itself, a proxy could use application composition to delegate the responsibility of sending the response to another application. The composition approach is recommended in cases where more than one virtual branch may be required. However, the virtual branch approach is simpler in that it makes sure that no other application is invoked between the proxy and the UAS.

12.2.4 Receiving Responses

The servlet container is responsible for automatically forwarding upstream the following responses received for a proxying operation:

- all informational responses other than 100
- the best response received when final responses have been received from all destinations
- all 2xx responses for INVITE
- exactly one 2xx response for non-INVITE as per [RFC 3261 16.7, para - Check responses for forwarding]

Additionally, if the supervised flag is true, the servlet engine invokes the application for these responses before forwarding them upstream. The application may modify responses in the notification callback. In this case, the modified responses are forwarded upstream. This is useful, for example, for application-level gateways that need to modify addresses in the body of responses.

As in the UAC case, applications are not invoked for incoming 100 responses.

When a 6xx response is received, the server CANCELs all outstanding branches and does not create new branches. Similarly, when a 2xx response is received, the server CANCELs all outstanding branches and does not create new branches unless the proxy is configured to not cancel branches with `setNoCancel(true).`

When an application is invoked to handle the best final response received and it is not a 2xx or 6xx, the application may add addresses for further destinations through the `Proxy.proxyTo` method or by creating additional proxy branches and re-starting the proxy. The effect is that a new branch is created for each additional destination, as if the method was invoked before a (tentative) best answer was passed to the application. If, in the upcall informing a servlet of the best response received, the servlet proxies to one or more additional destinations, the container does not immediately forward the best response received so far as the new branch may result in a “better” response. The ability to call `proxyTo` in the callback for best response received is useful, for example, for writing call-forward-no-answer type services.

A consequence of the described behavior is that an application may be notified of more than one final response for a transaction. This can happen because:

1. the application proxied to more destinations in the notification for a final response
2. multiple destinations returned 2xx responses

In the first case, the application may end up being notified of the same final response more than once.

12.2.4.1 Handling 2xx Responses to INVITE

The 2xx responses to INVITEs are special in that they cause both client and server transactions to terminate immediately with retransmissions bypassing the transaction layer [RFC 3261, sections 13.3.1.4 and 17.1.1].

As per RFC 3261, SIP proxies handle 2xx retransmissions by forwarding them statelessly upstream based on the Via header field.

Containers are expected to handle any such 2xx retransmissions using a suitable mechanism. Note that RFC 6026 updated the RFC 3261 behavior of forwarding 2xx retransmissions. Hence, when the container supports RFC 6026, the 2xx responses are discarded by the proxy if there is no matching transaction state machine. SIP servlet 2.0 compliant containers should support RFC 6026.

12.2.4.2 Correlating responses to proxy branches

When notified of an incoming response, it is sometimes useful for applications to be able to determine which of several branches the response was received for when there was more than one branch outstanding. Applications identify branches by request URI. Thus, to test whether a response was for a particular branch, it is sufficient to compare the request URI of that branch with the URI object previously passed to the `proxyTo` or `createProxyBranches` method by the application.

For incoming responses to proxied requests, `SipServletResponse.getRequest()` returns a `SipServletRequest` object representing the request that was sent on the branch on which the response was received. The request URI can then be obtained from the branch request object and be compared against destination URIs previously passed to the `proxyTo` or `createProxyBranches` method.

Containers are required to ensure that the request URI of the branch's request object is equal to the URI object that the application passed to the `proxyTo` method. (Recursion can, of course, cause branches to get created that the application didn't explicitly request and for which it has no URI.)

Servlets may be interested in being notified of intermediate final responses before the best final response is selected as per [RFC 3261, Section 16.6]. Version 1.0 of this specification did not offer any mechanism to peek at these intermediate responses.

To handle final responses received on each proxy branch, this specification provides a method on the `Servlet` class – `doBranchResponse(SipServletResponse resp)`. A servlet implementation that is interested in being notified of intermediate final responses received on individual branches must override this method. Note that this method wont be invoked for 2xx or 6xx responses.This method is invoked by the container only if the supervised flag is true for `Proxy`. A typical use case for using the `doBranchResponse` method is to create and proxy to additional branches on the receipt of a non-2xx final response.

Any attempt to add additional branches in `doBranchResponse()` for a canceled branch or a canceled proxy results in `IllegalStateException`. Only after the last branch receives a final response, the container must determine the best final response and pass it to the `doResponse()` method. If an application resumes proxying requests in `doResponse()`, after completing each round of re-proxying, the container invokes `doResponse()` with the best final response of all the previous rounds of proxying. Note that if `doBranchResponse()` is not overridden, the `doResponse()` method is invoked only for the best final response as before.

12.2.4.3 Sequential Search Timeout

As of v1.1 of this specification, the sequential search timeout may only be set using the general `proxyTimeout` parameter rather than the `sequentialSearchTimeout` parameter, which has now been deprecated.

This section clarifies the case where a SIP Servlet acts as a sequential search proxy, and the sequential search timeout expires before a final response is received for the current branch.

The sequential search timeout defined by the SIP Servlet API is semantically similar to Timer C with some differences explained below.

The following procedure describes the handling in case of sequential searches. Sequential search timer is referenced as SST:

1. Forward request and start branch.
2. Start the SST on receipt of a provisional response. SST does not start before a provisional response is received.

3. If SST expired but the final response has not been received, send a CANCEL to the INVITE transaction and move to the next branch.
4. If the SST value is greater than Timer-C, Timer-C SHOULD assume the value of SST.
5. If the final response is received within SST, process the response normally.

12.2.4.4 Handling 3xx responses

If the proxy recurse flag is set to false, it is up to the application developer to keep track of the contact addresses received in the redirect (3xx) responses. If the contact addresses received in multiple 3xx responses contain the same URI, the container SHOULD throw an `IllegalStateException` if the application attempts to proxy to the same URI again. This is to prevent creation of duplicate branches.

If the proxy recurse flag is set to true, the container SHOULD proxy only to unique contact addresses and ignore any duplicates. This may occur when a branch does not recurse after receiving a 3xx response with an alternate contact as a branch for an existing contact. Calling `getRecursedProxyBranches()` on such a branch results in an empty list. Every recursed contact results in a new branch, and duplicate contacts are ignored as per [RFC 3261, Section 16.5]. The container MUST remove the redirect contacts from the 3xx response since they have produced new branches. A 3xx response without any contacts can never be a best response of a proxy according to [RFC 3261 16.7, point 4] and should never be propagated up to the application.

12.2.5 Sending CANCEL

An INVITE proxy operation is aborted by invoking `Proxy.cancel()` or `ProxyBranch.cancel()`. An overloaded version of the cancel method is available so that application developers and containers MAY specify the reason for canceling the `Proxy` or `ProxyBranch` in accordance with [RFC 3326]. Invoking cancel causes the container to terminate all outstanding branches by sending CANCEL requests to the corresponding proxied INVITEs or to cancel the specified branch respectively. The proxy operation then proceeds as in the normal case, as per the SIP specification. As far as response handling is concerned, the act of canceling a branch can be thought of as a way of speeding up the generation of a final response. The application is still invoked to handle responses (assuming the supervised flag is true). Calls to `Proxy.cancel()` cancel all branches currently in progress and clear the proxy transaction's current target set. If the application subsequently calls `proxyTo()` with additional targets, the proxy transaction creates branches for those targets as usual. Alternatively, applications can create new Proxy branches by invoking `Proxy.createProxyBranches()`.

The invocation of both variants of the `cancel` method results in calling the cancel recursively on the `ProxyBranch` objects if the proxy recursed and there are recursed proxy branches.

As in the UAC case, applications are not invoked when CANCEL responses are received.

The comments made in 13.1.9 Sending CANCEL regarding the container having to delay the sending of a CANCEL until a provisional response has been received also apply to proxy transaction branches .

12.2.6 Receiving CANCEL

When receiving a CANCEL for a transaction for which a `Proxy` object exists, the server responds to the CANCEL with a 200 and:

- if the original request has not been proxied yet, the container responds to it with a 487 final response
- otherwise, all branches are canceled, and response processing continues as usual

In either case, the application is subsequently invoked with the CANCEL request. This is a notification only, as the server has already responded to the CANCEL and canceled outstanding branches as appropriate. The race condition between the server sending the 487 response and the application proxying the request is handled as in the UAS case as discussed in 13.2.3 Receiving CANCEL.

The method for turning off the automatic CANCEL processing of the container as specified in 13.2.3 Receiving CANCEL also applies to proxies.

12.2.7 Sending ACK

The SIP specification requires stateful proxies to generate ACKs for non-2xx final responses to INVITE requests. The purpose of these ACKs is to stop response retransmissions at the downstream server and have no semantic significance. Therefore, in the SIP Servlet API, the servlet container is responsible for generating ACKs for non-2xx final responses.Thus, SIP Servlets should never generate ACKs for proxied requests.

12.2.8 Receiving ACK

ACKs for non-2xx final responses are dropped. ACKs for 2xx’s are treated as other subsequent requests and are proxied by the container. Applications should not attempt to proxy ACKs explicitly.

If the application proxied the corresponding INVITE with record-routing enabled, the application is invoked before the ACK is proxied so as to give it an opportunity to modify the ACK. In this case, as when proxying the original request, the ACK is proxied downstream in its modified form.

12.2.9 Handling Subsequent Requests

As discussed in 18.5 Responses, Subsequent Requests and Application Path, a “subsequent” request is one that is dispatched to applications based on a previously established application path as opposed to initial requests that are dispatched to applications based on the application selection process.

When proxying with the recordRoute flag set to true, the server may receive subsequent requests in the same dialog. In these cases, processing takes place as described above except that the server is responsible for proxying the request to the destination specified in the top Route header as specified in the SIP specification [RFC 3261, section 16.6].

The application is still passed the request object and may modify it in the up-call before the server proxies it downstream. Applications should not attempt to explicitly proxy subsequent requests. Any attempt to do so results in an `IllegalStateException`. The `isInitial` method on the `SipServletRequest` interface returns true if the request is initial as defined above, and can be used as an indication of whether the application should explicitly proxy an incoming request or not.

Applications are allowed to reject subsequent requests, but only when doing so in the up-call notifying the application of the subsequent request. After performing the up-call, the container proxies the request unless the application generated a final response for it. The ability of proxy applications to respond to subsequent requests is needed, for example, for applications wishing to perform proxy authentication programmatically.

12.2.10 Max-Forwards Check

The Max-Forwards header serves to limit the number of elements a SIP request can traverse on the way to its destination. It consists of an integer that is decremented by one at each hop. RFC 3261 specifies that for a request to be proxied, it must have a Max-Forwards value greater than 0 [RFC 3261, section 16.3].

Since a SIP Servlet container cannot know a priori whether an application will proxy a request or not, it cannot perform the Max-Forwards check before invoking the application. Instead, the container performs the check when the application invokes `getProxy()` on a `SipServletRequest`. If Max-Forwards is 0, a `TooManyHopsException` is thrown. If not handled by the application, it is caught by the container which must then generate a 483 (Too many hops) error response. However, if the Max-Forwards is 0 for an OPTIONS request, `Proxy` may not throw a `TooManyHopsException` but rather respond to the request.

The Max-Forwards header is used within the container for inter-container loop detection. See 18.9 Loop Detection.

12.2.11 Max-Breadth Check

The Max-Breadth header limits the number of parallel forks that can be made on a SIP request by the downstream proxies. As per RFC 5393, each downstream proxy distributes the Max-Breadth among the active parallel branches so that outgoing Max-Breadth is the sum of the Max-Breadth header field values in all forwarded requests in the response context that have not received a final response.

In a SIP Servlet container, the application decides the number of times the request is forked in parallel. An application may explicitly set the breadth on the forked requests by using the `SipServletRequest.setMaxBreadth` method. Applications may retrieve the forked request by using the `ProxyBranch.getRequest` method.

If a Proxy application does not set the breadth explicitly, the container distributes the available breadth evenly to the branches when the application invokes the `Proxy.proxyTo` or `Proxy.startProxy` methods.

At the time of forking, if the sum of Max-Breadth of active parallel branches exceeds the Max-Breadth of the original incoming request, the container throws `InsufficientBreadthException`. For a proxy, this check is done when the application invokes the `Proxy.proxyTo` and `Proxy.startProxy` methods. Applications may catch this exception and send an error response (440) or attempt to proxy again after adjusting the breadth.

Max-Breadth is used within the container for inter-container loop detection.See 18.9 Loop Detection for details.

12.3 Proxying and Sessions

When an application proxies a request, it may subsequently be invoked to handle requests and responses related to that transaction or to subsequent transactions, if the application is record-routed.

Whenever an application is invoked because an event occurred on an existing transaction or for a subsequent request and the application has previously obtained a `SipApplicationSession` and/or `SipSession`, the container must ensure that those same session objects are associated with subsequent requests and responses that the application is invoked to handle.

Also, as a result of forking proxies, an initial request may result in multiple dialogs being established. 8.2.3 Creation of SipSessions discusses under which circumstances a container must associate an incoming response or a subsequent request with a `SipSession` cloned from that of an existing dialog.

12.4 Record-Route Parameters

When record-routing, it is often convenient for a SIP proxy to push state to the dialog endpoints and user agents, and then have it returned in subsequent requests of the dialog. This can be achieved by adding parameters to the URI of the Record-Route header field inserted by the proxy. The Record-Route header field value, including URI parameters, contribute to the route set of the user agents and is returned to the proxy in a Route header in subsequent requests. The loose routing mechanism of RFC 3261 ensures that a SIP proxy gets back the URI it record-routed within a Route header field value of the subsequent request.

The `Proxy.getRecordRouteURI` method allows a record-routing proxy application (be it dialog stateful or stateless) to set parameters on the Record-Route header that will be inserted by the container into a proxied request:

```
SipURI getRecordRouteURI();
```

The URI returned by the `getRecordRouteURI` method should be used only for pushing application state in the form of URI parameters to the user agents. Applications must not set SIP URI parameters defined in RFC 3261. This includes `transport`, `user`, `method`, `ttl`, `maddr`, and `lr`. Other components of the URI(e.g., host, port, and URI scheme) must also not be modified by the application. These Record-Route URI components are populated by the container and may or may not have valid values at the time an application proxies a request.

Parameters added to the Record-Route header field values in this manner can be retrieved from subsequent requests by using the `getParameter` method of `SipServletRequest` or through access to the popped Route header, as discussed in 6.7.2 Parameters.

Note that this mechanism guarantees that when a proxy application adds parameters to a URI obtained through the `getRecordRouteURI` method, the value of those parameters can be retrieved by the same application by using `SipServletRequest.getParameter` on subsequent requests in that dialog (assuming the user agents involved are well-behaved). However, there is no guarantee that the parameters go unchanged into an actual Record-Route header. For example, due to application composition, the parameters may in some cases go into a Contact header. In addition, the container may choose to insert only one Record-Route header when there are multiple record-routing proxies on the application path. In this case, the container encodes parameters so as to avoid name clashes between applications. Implementations may even choose to store the parameter set locally, possibly along with other dialog-related data, and retrieve it based on a dialog ID computed for subsequent incoming requests. There is no hard bound defined for the size of data that can be pushed to endpoints in Record-Route header fields, but it is recommended that application writers keep in mind that some implementations may not function correctly with large data.

This version of the SIP Servlet API does not allow proxy applications to push different states to the two endpoints of a dialog. The UAS copies the Record-Route header field of requests unchanged into 2xx responses. The proxy application is not given the opportunity to modify its own previously inserted Record-Route parameters when processing 2xx responses. This would effectively rewrite the Record-Route header field value and cause the state received in subsequent requests from the caller to differ from that received in subsequent requests from the callee. This feature is potentially useful but may have an adverse effect on performance and security. This is because the Record-Route header in responses cannot be protected end-to-end if rewritten by proxies.

12.5 Path Header and Path Parameters

The Path introduced in the [RFC 3327] header provides the semantics of a Path created by a REGISTER request, which shall be followed by subsequently created dialogs. The concept is similar to Record-Route but defines a path outside of dialogs. The Path is added by intermediate proxies and maintained by the Registrar.

The Home Proxy shall then look up the Path alongside the AOR binding and add preloaded routes to the request destined for the registered UA. The requirements on SIP Servlet API are summarized below:

1. Proxy applications may want to add the `Proxy` (itself) to the Path in REGISTER
2. A proxy application, besides adding its own Path, may also push arbitrary Path [RFC 3327 Section 5.2]
3. A registrar application supporting paths is required to include a set of Path headers in the response it returns to a REGISTER request.

Similar to `getRecordRouteURI()` (see 12.4 Record-Route Parameters), the `getPathURI()` API is additionally defined on the `Proxy` interface. This gives applications a chance to set some parameters on the Path URI. These parameters can be subsequently retrieved as the Request URI parameters or from the popped route mechanism as defined in 6.7.3 Popped Route Header.

The SIP option tag "`path`" should be present in the immutable `List` of supported extensions as described in section 3.3 Extensions Supported in addition to the `100rel` option tag.

A `pushPath()` method is defined on `SipServletMessage` similar to the `pushRoute()` method. However, `pushPath()` shall throw an `IllegalStateException` on non-REGISTER Requests, non-REGISTER Responses, and non-200 class Responses to REGISTER requests.

A `pushLocalPath()` method is defined in `SipServletMessage`. Applications may use this method for adding container address in the Path header. However, `pushLocalPath()` shall throw

`IllegalStateException` on non-REGISTER Requests, non-REGISTER Responses, and non-200 class Responses to REGISTER requests.

Typically, Path is no different from Record-Route in semantics, except that it creates a path for subsequent dialogs. Therefore, the Path header is a "System Header" (see 6.4.2 System Headers) managed by the SIP Servlet container.

13 Acting as a User Agent

This chapter describes how SIP servlets perform various UA operations such as initiating and responding to requests. The SIP specification lists client and server transaction state machines that define at which points in processing that various actions are allowed. As mentioned in 6.2 Implicit Transaction State, when an application attempts an action that violates the SIP state machine, the container throws an `IllegalStateException`.

The notions of SIP client and server are meaningful only in relation to a particular transaction. An application acting as a UA often needs to act as both a client and a server in a dialog.

13.1 Client Functions

13.1.1 Creating Initial Requests

When initiating a request, a UAC creates a `SipServletRequest` object, optionally modifies it, and then sends it Applications create the initial request object by invoking the overloaded `createRequest` method on a `SipFactory`.

More precisely, a SIP servlet carries out the following steps in order to send a new request which does not belong to an existing dialog:

- look up a `SipFactory` as an attribute on the `ServletContext`, see 3.2 The SipFactory
- invoke one of the overloaded `SipFactory.createRequest` methods
- modify the resulting `SipServletRequest` as appropriate, for example, set the content
- invoke `send` on the request object

The `SipFactory` interface defines the following methods for creating new requests:

```
SipServletRequest createRequest(
    SipApplicationSession appSession,
    String method,
    Address from,
    Address to);

SipServletRequest createRequest(
    SipApplicationSession appSession,
    String method,
    URI from,
    URI to);

SipServletRequest createRequest(
    SipApplicationSession appSession,
    String method,
    String from,
    String to) throws ServletParseException;
```

The returned `SipServletRequest` exists in a new `SipSession` which belongs to the specified `SipApplicationSession`. The handler for the newly created `SipSession` is the application's default servlet for v1.0 applications, see 8.2.9 The SipSession Handler and the main servlet for v1.1 applications, see 19.2 Servlet Selection. This can be changed by the application by invoking `SipSession.setHandler`. The container is required to assign the returned request a fresh, globally unique Call-ID as defined by the SIP specification. The From and To headers are as specified by the from and to parameters, respectively. The container is responsible for adding a CSeq header. The request method is given by the method parameter. The default value of the request URI is the URI of the To header, with the additional requirement that for REGISTER requests the user part of a SIP request URI is empty. The application can change this by invoking `setRequestURI` on the newly created `SipServletRequest`.

Note: ACK and CANCEL requests have special status in SIP. 13.1.7 Sending ACK and 13.1.9 Sending CANCEL discuss how and when applications generate those requests. The `SipFactory` methods listed above MUST throw an `IllegalArgumentException` when invoked to create ACK or CANCEL requests.

Once constructed, the application may modify the request object, for example, by setting the request URI, adding headers, or setting the content. The request is then sent by invoking the `send()` method on `SipServletRequest`. The request is routed by the container based on the request URI and any Route headers present in the request, as per the SIP specification. A fourth version of `SipFactory.createRequest` is intended specifically for writing back-to-back user agents, and is discussed in 15.2 B2BUA Helper Class.

13.1.1.1 Copying From and To Addresses

Ordinarily, when `Address` objects are set as header field values of a message, they are not copied. However, the From and To header fields are special in that they are container managed and have certain requirements regarding tag parameters imposed from the SIP specification. For these reasons, the `SipFactory.createRequest` method makes a deep copy of the *from* and *to* arguments before making them the values of the From and To header fields of the newly created request. If the copied *to* `Address` has a *tag* parameter it's removed as the To header field of outgoing initial requests must be tag-free. The copied *from* `Address` is given a fresh `tag` parameter according to the SIP specification. Any component of the *from* and *to* URIs not allowed in the context of SIP From and To headers are removed from the copies. This includes, headers and various parameters as referenced in [RFC 3261, Section 20].

The copied *from* and *to* `Address` objects are associated with the new `SipSession`. If a dialog is established, the container must update the *to* `tag` to the value chosen by the peer SIP element. Subsequent requests belonging to the same `SipSession` will have the same From and To headers. Applications can retrieve the copied *from* and *to* `Address` objects (reflecting new tags) from either the `SipSession` or the newly created request, but are not allowed to modify them.

13.1.2 Creating Subsequent Requests

For subsequent requests, that is, requests made in an already established dialog, applications use the following method on the `SipSession` representing the dialog:

```
SipServletRequest createRequest(String method);
```

This method constructs a request with container provided values for request URI and Call-ID, From, To, CSeq, and Route headers. The application may modify the request before invoking it by invoking the `send` method on the request object.

13.1.3 Pushing Route Header Field Values

A UAC may push Route header field values, basically SIP or SIPS URIs, onto the Route header field of the request. The effect is that the request visits the set of proxies identified in those Route values before being delivered to the destination. This is what is known as a preloaded Route in the SIP specification.

The `SipServletRequest` interface defines the `pushRoute` method for adding entries to the Route header field:

```
void pushRoute(SipURI uri);
```

The `pushRoute` method is applicable to be used by UAs only until the time when the dialog has not yet been established. This means that `pushRoute` can only be done on initial requests. Subsequent requests within a dialog follow the route set. Any attempt to do a `pushRoute` on a subsequent request in a dialog MUST throw an `IllegalStateException`.

The Route header behaves like a stack – `pushRoute` adds items to the head of the list and the SIP routing algorithm pops items from the top. Therefore, if `pushRoute` is called more than once, the request will visit the most recently added proxy first. The same mechanism is available for proxying applications and is discussed in 12.2.2 Pushing Route headers.

13.1.4 Sending a Request as a UAC

Once created, a request is sent by invoking `send()` on the `SipServletRequest` object:

```
void send() throws IOException;
```

If the `send` method call throws an exception, it means the request was not successfully sent and the application will not receive any callbacks pertaining to that transaction, that is, the application will not receive any responses for this request.

If the `send` method does not throw an exception, the container presents the application with a final response except for an ACK request. Containers may send the request asynchronously in which case sending may fail after the `send` method has returned successfully. In this type of situation, the container will generate its own final response except for an ACK request. In this particular case, a 503 (Service Unavailable) response would be appropriate as per [RFC 3261 8.1.3.1 and 16.9].

13.1.5 Receiving Responses

The UAC application is invoked for all incoming responses except 100 responses (and retransmissions). The servlet invoked is the current handler for the `SipSession` to which the request belongs (see 8.2.9 The SipSession Handler). The container invokes `Servlet.service` with `SipServletResponse` and a null value for the request argument. If the servlet extends the `SipServlet` abstract class, the response is further dispatched based on the value of the status code.

13.1.5.1 Handling Multiple Dialogs

Due to forking at downstream proxies, multiple 2xx responses may be received for a single INVITE request. In this (and similar) cases, the container clones the original `SipSession` for the second and subsequent dialogs, as detailed in 8.2.3.2 Derived SipSessions. The cloned `SipSession` object belongs to the same `SipApplicationSession` as the original `SipSession`, and contains the same application data.Howerver, invoking its `createRequest` method creates requests belonging to the second or subsequent dialog, that is, with a To `tag` specific to that dialog.

13.1.6 Transaction Timeout

If a timeout occurs in the transaction layer as specified by RFC3261 (i.e., In INVITE client transaction, if Timer B fires; and in non-INVITE client transaction, Timer F fires), the container generates its own 408 Request Timeout final response and passes it to the application.

13.1.7 Sending ACK

In SIP, final responses to INVITE requests trigger sending of an ACK for that response from the UAC to the UAS. ACKs to non-2xx final responses are needed for reliability purposes only and are sent hop-by-hop. That is, they are generated by proxies as well as the UAC. ACKs to 2xx responses, on the other hand, signal completed session setup and may carry semantically useful information in the body, for example, IP addresses and codecs for the media streams of the session. These ACKs are sent end-to-end from the UAC all the way to the UAS. Generally speaking, only ACKs for 2xx responses are of interest to services, so SIP servlet containers are responsible for generating ACKs for non-2xx responses However, applications are responsible for generating ACKs for 2xx responses. A request object representing the ACK is created by calling the `SipServletResponse` method:

```
SipServletRequest createAck()
```

The application may modify the returned ACK object before invoking `send` on it. It is the container’s responsibility to retransmit application- generated ACKs for 2xx’s when a 2xx retransmission is received and the container must not deliver the 2xx retransmission to the UAC application. It is recommended that containers generate ACKs for non-2xx final responses prior to invoking the application, so as to stop response retransmission as soon as possible.

13.1.8 Sending PRACK

RFC 3262 introduced Reliable provisional responses and the new PRACK method. This RFC also added two new system headers RSeq and RAck. To allow for generation of PRACKs, use the following method on `SipServletResponse`:

```
SipServletRequest createPrack()
```

The RAck header is populated by the container in such PRACKs according to procedures specified in RFC 3262.

13.1.9 Sending CANCEL

A UAC can cancel an in-progress INVITE request y sending a CANCEL request for the INVITE. A SIP servlet acting as a UAC can invoke the following `SipServletRequest` method on the original INVITE request object:

```
SipServletRequest createCancel()
```

The application sends the returned CANCEL request by invoking `send` on it. Responses to CANCEL requests are not passed to the application. UACs and proxies are not allowed to cancel an INVITE request before a 1xx response has been received [RFC 3261, section 9.1]. SIP Servlet applications may do so, though. It is the container's responsibility to delay sending of the CANCEL until a provisional response has been received.

13.2 Server Functions

By definition, a SIP servlet that responds to an incoming request with a final response becomes a UAS for the corresponding transaction.

13.2.1 Sending Responses

A servlet may generate a number of provisional responses as well as a single final response for an incoming request. It does so by invoking `createResponse` on the request object. The resulting `SipServletResponse` is then subsequently sent, possibly after having been modified, by invocation of the `send` method. 2xx responses to INVITEs are special in that they are retransmitted end-to-end. The SIP specification is defined in terms of a layered software model in which the transaction user (a UAS in this case) accepting an INVITE is responsible for periodically retransmitting the 2xx [RFC 3261, section 13.3.1.4]. In the SIP Servlet API, this task must be handled by the container. This is fully within the scope of the logical model used for purposes of presentation in RFC 3261.

13.2.2 Receiving ACK

Applications are notified of incoming ACKs for 2xx responses to INVITEs that the application sent upstream. Applications are not notified of incoming ACKs for non-2xx final responses to INVITE. These ACKs are needed for reliability of final responses but are not usually of interest to applications.

It is possible that a UAC fails to send an ACK for an accepted INVITE request before the transaction times out. This is communicated to the application through the `SipErrorListener` mechanism discussed in 10.1 SIP Servlet Event Types and Listener Interfaces.

13.2.3 Receiving CANCEL

When a CANCEL is received for a request that has been passed to an application and the application has not responded yet or proxied the original request, the container:

- responds to the original request with a 487 (Request Terminated)
- responds to the CANCEL with a 200 OK final response
- notifies the application by passing it a `SipServletRequest` object representing the CANCEL request. The application should not attempt to respond to a request after receiving a CANCEL for it, nor should it respond to the CANCEL notification.

Clearly, a race condition may occur between the container generating the 487 response and the SIP servlet generating its own response. This should be handled by using standard Java mechanisms for resolving race conditions. If the application wins, it is not notified that a CANCEL request was received. If the container wins and the servlet tries to send a response before (or for that matter after) being notified of the CANCEL, the container throws an `IllegalStateException`.

To comply with the behavior specified in RFC 6141, some applications may want to respond to the INVITE with a 2xx response instead of 487, when a CANCEL arrives. An application may use the `disableCancelHandling` element of the `SipApplication` annotation to turn off the processing of the CANCEL request by the container. When CANCEL handling is turned off, container neither generate 487 response nor cancel the branches automatically as previously explained in this section. Application will need to handle such scenarios explicitly, for example, when it is invoked with the CANCEL request.

13.2.4 Handling Error Conditions

13.2.4.1 Receiving unimplemented subsequent requests

When a UAS application does not handle subsequent requests (for example, `doMessage()` is not overridden to handle an in-dialog MESSAGE request) the container SHOULD send back a 501 Not Implemented response. However, this does not apply to ACKs for 2xx responses for INVITE.

13.2.4.2 Handling Pending Invites

As per [RFC 3261 14.2], if the container receives a second INVITE before it sends the final response to a first INVITE with a lower CSeq sequence number on the same dialog, it MUST return a 500 (Server Internal Error) response to the second INVITE and MUST include a Retry-After header field with a randomly chosen value of between 0 and 10 seconds.

If the container receives an INVITE on a dialog while an INVITE it had sent on that dialog is still in progress, it MUST send a 491 (Request Pending) response to the received INVITE.

14 SIP Over WebSockets

The WebSocket Protocol defined by RFC 6455 enables two-way communication between a client running untrusted code in a controlled environment, such as web browsers, to a remote host that has opted-in to communications from that code. RFC 7118 defines a new WebSocket sub-protocol for transporting SIP messages between a WebSocket client and a server.

SIP servlet containers support incoming WebSocket connections from WebSocket clients that contain “sip” as one of the sub-protocols during handshake. After the handshake negotiation completes, the established WebSocket connection can be used for the transport of SIP requests and responses. The WebSocket messages transmitted over this connection MUST conform to the negotiated WebSocket sub-protocol. The SIP Servlet Container follows the procedures specified in RFC 3261 and RFC 7118 for implementing the SIP Websocket Transport.

> Support for use of WebSocket transports by a SIP container is optional. Containers with this capability must include the string "7118" in the list of supported rfcs available to applications via the `ServletContext`. see 3.3 Extensions Supported.

14.1 WebSocket Transport

RFC7118 updates RFC 3261 to introduce WebSocket as another reliable protocol apart from the ones already defined. SIP Servlet containers are required to support the procedures specified in section 5 of the RFC 7118, where ever it is applicable to the SIP WebSocket Server. The procedures specific to the SIP WebSocket client are not applicable to this version of the SIP Servlet specification.

Notes: A future version of this specification might define APIs to support the SIP WebSocket Client.

14.2 SipWebSocketContext

Each SIP WebSocket connection established to a SIP Servlet Container has an associated `SipWebsocketContext`. A `SipWebSocketContext` can be retrieved by a SIP Servlet using `SipServletMessage.getLocalSipWebsocketContext(`. A `SipWebSocketContext` pertaining to a specific `Flow` can also be retrieved by using `SipServletContext.getSipWebSocketContext(String flowId).`

14.2.1 Accessing WebSocket URL

The URL on which the SIP servlet container received the SIP Message can be accessed by SIP Servlets using the `SipWebSocketContext.getURL()` method. The returned URL contains a protocol, server name, port number, server path, and include query string parameters.

14.2.2 Initializing SipWebSocketContext

Information available during the WebSocket handshake is often useful to the SIP application. Some examples of such information are the relevant headers from the original HTTP upgrade request (eg: User-Agent header) and HTTP servlet security principal. This information is made available to SIP servlets by using the following methods of `SipWebSocketContext`:

- `getWebSecurityPrincipal()`: Returns a `java.security.Principal` object containing the name of the authenticated user at the time of the WebSocket handshake.
- `getHttpHeader(String name)`: Returns the HTTP header saved by the SIP container at the time of the WebSocket handshake. A SIP container that supports SIP over WebSockets is required to allow administrators to configure the names of HTTP headers that are saved. If no http header is configured to be saved during the handshake, this method returns null.

14.3 Creating an Initial Request

When a SIP servlet application needs to create an initial request, the application must specify the WebSocket connection to use for sending the SIP message. RFC7118 suggests SIP Outbound Extension [RFC 5626] as a way to reach the client. Thus, the SIP servlet application can use `the SipSession.setFlow(Flow flow)`, `Proxy.setFlow(Flow flow),` or `ProxyBranch.setFlow(Flow flow)` methods to specify the Flow representing the SIP WebSocket Connection.

14.4 Application Composition

When an initial request is received on WebSocket transport, just like any other SIP Servlet Request on other transports, the message is passed to the application router to begin the application selection and composition process as specified in Chapter 18, “Application Selection And Composition Model” of this specification.

14.5 Authentication

A SIP Servlet container is required to support authentication prior to sending SIP requests as specified in section 7 of the RFC 7118. As per RFC 7118, SIP WebSocket Server may use web authentication for authenticating the SIP messages on WebSocket connections and may additionally employ SIP protocol authentication such as DIGEST authentication. To achieve this, the application may configure identity assertion scheme “WEBSOCKET” as shown below.

Listing 14-1 WEBSOCKET identity assertion scheme

```
@SipApplication(
        loginConfig=@SipLogin(
                authMethod = "DIGEST",
                realmName = "example.com",
                identityAssertion = SipLogin.IdentityAssertion.WEBSOCKET,
                identityAssertionSupport = SipLogin.IdentityAssertionSupport.SUPPORTED
        )
)
package com.example;
```

When the identity assertion scheme is configured, the SIP container checks whether the WebSocket connection is authenticated. If the WebSocket connection is already authenticated (for example, by the web application that originally handles the context-root of the websocket URL), the principal established during that authentication process is made available to the SIP Servlet application by using the `SipServletMessage.getRemoteUser()` and `SipServletMessage.getUserPrincipal()` methods. In the case of converged applications, the principal established for SIP servlets may differ from the one established for HTTP servlets depending on container configurations.

Similarly, `SipServletMessage.isUserInRole` returns true if the user in question has the relevant role defined in the `security-role` element of the deployment descriptor of the SIP Servlet application.

14.5.1 Converged SIP and HTTP applications

As per RFC 7118, a SIP identity is assigned to a user logged into the web. This SIP identity is later used for authenticating SIP messages as explained in 14.5 Authentication. In case of a converged SIP and HTTP application, this SIP identity is looked up by the web application from an appropriate source (eg: HSS). For an authenticated `HttpServletRequest`, this SIP identity is set on the container by using `SipServletContext.assignWebSocketSipIdentity(HttpServletRequest httpRequest, String sipIdentity)` method as shown in the example below.

Listing 14-2 Assigning SIP identity

```
if (httpServletRequest.getUserPrincipal() != null) {
    String sipIdentity =
findSipIdentity(httpServletRequest.getUserPrincipal());
    SipServletContext context =
    (SipServletContext) httpServletRequest.getServletContext();
    context.assignWebSocketSipIdentity(httpServletRequest, sipIdentity);
}
```

After the SIP identity is assigned, the converged SIP and HTTP container is required to take necessary steps to associate the SIP identity with the WebSocket connection. When an application invokes `HttpServletResponse.encodeURL`, the container should encode the URL with the SIP identity (among other things) so that after a WebSocket handshake completes, the SIP identity is used to authenticate the SIP messages on that WebSocket connection. Similarly, an authenticated `HttpSession` retains the association with SIP Identity so that SIP messages on a WebSocket connection initiated on that session is authenticated.

15 Back To Back User Agents

A back-to-back user agent (B2BUA) is a SIP element that acts as an endpoint for two or more dialogs and forwards requests and responses between those two dialogs in some fashion. Experience has shown that they are one of the most frequently used application types.

B2BUA's are sometimes considered undesirable because of their potential to break services. This potential stems from the fact that they sit between two endpoints and in some way mediate the signaling between the endpoints. If the B2BUA doesn't know about an "end-to-end" service being used between those two endpoints, it may inadvertently break it.

B2BUA's are, however, an important tool for SIP application developers and as such are supported by the SIP Servlet API. This specification provides a helper class and additional methods to the SIP Servlet API, making the B2BUA pattern very easy to implement.

15.1 What is a B2BUA

There is no IETF standard that defines how a B2BUA behaves. However, it is largely assumed to be an entity that receives a request upstream as a UAS and relays it as a new request based on the received request downstream as a UAC. Effectively, the application relays the request downstream.

From a SIP servlet perspective, an application is considered a B2BUA if it indicates the routing directive as CONTINUE. An example of explicit linking is shown below.

Listing 15-1 B2BUA relays initial request and signals the CONTINUE directive

```
          +---------------------------------------+
        |                                          |
  req1  | req2 = factory.createRequest(appSession, |   req2
------->| "INVITE", req1.getFrom(), req1.getTo()); |--------->
        | // copy headers and content from req1    |
        | req2.setRoutingDirective(CONTINUE, req1);|
        | req2.send();                             |
        +------------------------------------------+
```

15.1.1 Navigating between UAS and UAC side of a B2BUA

A common behavior of B2BUA is to forward the requests and responses received on the UAS side of the B2BUA to the UAC side and vice versa. Since both UAS and UAC sides of the B2BUA contain their own `SipSession` objects, while receiving a request or response on one side, an application often needs to navigate to a `SipSession` on the other side. To facilitate this, applications can store session IDs of the peer session as an attribute in the `SipSession` as shown in Listing 15-2.

Listing 15-2 Linking SipSessions

```
public void linkSessions(SipSession s1, SipSession s2) {
    s1.setAttribute(LINKED_SESSION_ID, s2.getId());
    s2.setAttribute(LINKED_SESSION_ID, s1.getId());
}
```

The linked sessions can then be retrieved by using the stored attributes as shown in Listing 15-3.

Listing 15-3 Retrieving linked SipSessions

```
private SipSession retrieveOtherSession(SipSession s) {
   String otherSessionId = (String) s.getAttribute(LINKED_SESSION_ID);
    return s.getApplicationSession().getSipSession(otherSessionId);
}
```

Many times, UAC and UAS sides of the B2BUA need to access the `SipServletRequest` on the other side. To facilitate this, applications may store the request IDs of the SipServletRequest in each other as shown in Listing 15-4.

Listing 15-4 Linking Requests

```
    @Bye
    protected void handleBye(SipServletRequest bye1) throws IOException {
        SipSession otherSession = retrieveOtherSession(bye1.getSession());
        SipServletRequest bye2 = otherSession.createRequest("BYE");
        linkRequests(bye1, bye2);
        bye2.send();
    }

    private void linkRequests(SipServletRequest r1, SipServletRequest r2) {
        r1.setAttribute(LINKED_REQUEST_ID, r2.getId());
        r2.setAttribute(LINKED_REQUEST_ID, r1.getId());
    }
```

When needed, the requests can then be accessed from the session, if they are active as shown in Listing15-5. More details about accessing requests can be found in 6.7.1 Accessing Active Requests.

Listing 15-5 Retrieving Requests

```
    @SuccessResponse @Bye
    protected void byeResponse(SipServletResponse r1) throws IOException {
        SipServletRequest request = retriveLinkedRequest(r1.getRequest());
        SipServletResponse r2 = request.createResponse(r1.getStatus(),
r1.getReasonPhrase());
        r2.send();
    }


    private SipServletRequest retriveLinkedRequest(SipServletRequest r1) {
        String requestId = (String) r1.getAttribute(LINKED_REQUEST_ID);
        SipSession session = retrieveOtherSession(r1.getSession());
        return session.getActiveRequest(requestId);
    }
```

15.1.2 ACK and PRACK handling in B2BUA

To forward an ACK request, the B2BUA needs to access the servlet on the other side and the application needs to access final response. The application can retrieve the final response of the INVITE request by using the `SipServletRequest.getFinalResponse()` method. When an ACK request arrives, B2BUA first retrieves the INVITE method from the other side by using the

SipSession.getActiveInvite(UAMode) method and then accesses the final response from SipServletRequest. Listing15-6 shows an example code.

Listing 15-6 B2BUA relaying ACK

```
    @Ack
   protected void handleAck(SipServletRequest uasAck) throws IOException {
        SipSession uacSession = retrieveOtherSession(uasAck.getSession());
        SipServletRequest uacInvite =
uacSession.getActiveInvite(UAMode.UAC);
       SipServletRequest uacAck = uacInvite.getFinalResponse().createAck();
        uacAck.send();
    }
```

For reliable provisional responses, B2BUA may establish a relationship between an incoming reliable provisional response with the one relayed by the B2BUA. This may be done by saving the reliable sequence number (RSeq) and request ID as attributes in the response. Listing 15-7 shows an example code.

Listing 15-7 Linking reliable provisional responses

```
    @ProvisionalResponse
    void provisionalResponse(SipServletResponse r1) throws Exception {
        SipServletRequest request = retriveLinkedRequest(r1.getRequest());
        SipServletResponse r2 = request.createResponse(r1.getStatus());
        if (r1.isReliableProvisional()) {
            String respId = r1.getProvisionalResponseId();
            r2.setAttribute(LINKED_RELIABLE_RESPONSE_ID, respId);
            r2.sendReliably();
        }
    }
```

The linked provisional response may then be retrieved when the B2BUA forwards a PRACK as shown in Listing 15-8.

Listing 15-8 Forwarding PRACK

```
    @Prack
    void handlePrack(SipServletRequest prack) throws Exception {
        SipServletResponse r1 = prack.getAcknowledgedResponse();
        SipSession session = retrieveOtherSession(prack.getSession());
        String respId =
        (String) r1.getAttribute(LINKED_RELIABLE_RESPONSE_ID);
        SipServletResponse r2 =
        session.getUnacknowledgedProvisionalResponse(respId);
        r2.createPrack().send();
    }
```

15.1.3 B2BUA and Forking

When an INVITE request is forked downstream, one request may get responses on different dialogs (derived sessions). Each response causes a transaction branch and is represented in the SipServlet API by the `InviteBranch` interface. Refer to section 8.5 InviteBranch for more information about `InviteBranch`.

A B2BUA forwards such responses on different branches as UAS to create a new `InviteBranch` by using `SipServletRequest.createInviteBranch` object. Listing 15-9 shows a code example.

Listing 15-9 Sending a response on a particular branch

```
    @Invite
    void handleInviteResponse(SipServletResponse r1) throws IOException {
        InviteBranch branch = r1.getSession().getActiveInviteBranch();
        if (branch != null) {
            SipSession uasSession = retrieveOtherSession(r1.getSession());
            if (uasSession == null) {
                SipServletRequest req =
                retriveLinkedRequest(r1.getRequest());
                branch = req.createInviteBranch();
            }
            SipServletResponse r2 =
            branch.createResponse(r1.getStatus(), r1.getReasonPhrase());
            r2.send();
        }
    }
```

Note that one `SipServletRequest` may result in more than one branch. Hence, when there are circumstances where a B2BUA needs to access a particular branch, it is required to use the corresponding `InviteBranch` object. For example, while relaying an ACK message, the B2BUA may access the final response from the `InviteBranch` object as shown in Listing 5-10.

Listing 15-10 Relaying ACK on a particular branch

```
@Ack
protected void handleAck(SipServletRequest uasAck) throws IOException {
    SipSession uacSession = retrieveOtherSession(uasAck.getSession());
    InviteBranch branch = uacSession.getActiveInviteBranch();
    if (branch != null) {
       SipServletResponse response = branch.getFinalResponse();
       response.createAck().send();
    }
}
```

Although non-INVITE requests can also be forked, the absence of ACK and PRACK messages make their call flow much simpler. Similarly, non-INVITE requests do not support sending responses on more than one dialog from the UAS. Thus, the SIP Servlet API does not provide an API equivalent to the `InviteBranch` for non- INVITE requests.

15.2 B2BUA Helper Class

A B2BUA helper class contains useful methods for simple B2BUA operations. The B2BUA helper class instance can be retrieved from the `SipServletRequest` by invoking `getB2buaHelper()` on it.

```
B2buaHelper getB2buaHelper() throws IllegalStateException;
```

This also indicates to the container that this application wishes to be a B2BUA.

Any UA operation is permitted by the application but the application cannot act as `Proxy` after that, so any invocation to `getProxy()` must then throw `IllegalStateException`.

Similarly, the `getB2buaHelper()` method throws an `IllegalStateException` if the application already retrieved a proxy handle by an earlier invocation of `getProxy()`.

Note: B2BUA helper class functionality is limited to that explained in this section. It can only be used for linking 2 legs (inbound and outbound) and does not support all cases of forking at the B2B or downstream. A B2BUA application may come across complex scenarios and call flows that can not be implemented by using B2BUA helper class. Hence, this specification recommends using the APIs explained in the section above (see 15.1 What is a B2BUA) to implement B2BUAs.

15.2.1 Creating new B2BUA Request

The behavior specified here aims at minimizing the risk of breaking end-to-end services by suggesting that all unknown headers be copied from the incoming request to the outgoing request.

When an application receives an initial request for which it wishes to act as a B2BUA, it may invoke the `createRequest` method available on `B2buaHelper`:

```
SipServletRequest B2buaHelper.createRequest(
  SipServletRequest origRequest,
  boolean linked,
  java.util.Map<java.lang.String, java.util.Set> headerMap)
  throws IllegalArgumentException
```

This method creates a request identical to the one provided as the first argument according to the following rules:

- All unknown headers and Route, From, and To headers are copied from the original request to the new one. The container assigns a new From tag to the new request.
- Record-Route and Via header fields are not copied. As usual, the container adds its own Via header field to the request when it's actually sent outside the application server.
- The headers in the new request can be taken for the optional `headerMap` which is a `Map` of headers that will be used in place of the ones from `origRequest`. For example:

  ```
  {"From" => {sip:myname@myhost.com},
   "To"     =>  {sip:yourname@yourhost.com} }
  ```

  where "From" and "To" are keys in the map. The only headers that can be set using this `headerMap` are non-system headers and From, To, and Route headers. For Contact headers, only the user part and some parameters are to be used as defined in 5.1.3 The Contact Header Field. The values in the map is a `java.util.Set` to account for multi-valued headers.

  The values in `headerMap` MUST override the values in the request derived from the `origRequest`. Specifically, they do not append header values. Attempts to set any other system header results in an `IllegalArgumentException`.
- The "`linked`" boolean flag indicates whether the ensuing `SipSession` and `SipServletRequest` are to be linked to the original ones. The concept of linking is discussed in 15.2.2 Linked SipSessions And Linked Requests.
- For non-REGISTER requests, the Contact header field is not copied but is populated by the container as usual.

These methods are included for convenience and performance. Like other `createRequest` methods, the returned request belongs to a new `SipSession`.

Note: The `SipFactory.createRequest(SipServletRequest origRequest, boolean sameCallId)` method has been deprecated in this specification as the usage of this method with `sameCallId` flag as "true" actually breaks the provisions of [RFC 3261] where the Call-ID value is to be unique across dialogs. Instead, the use of `B2buaHelper.createRequest(SipServletRequest origRequest)` is recommended.

15.2.2 Linked SipSessions And Linked Requests

15.2.2.1 Explicit Session Linkage

A B2BUA usually contains two `SipSessions` (although there can be more than two). The most common function of a B2BUA is to forward requests and responses from one `SipSession` to the other, after performing some transformation [application of business logic]. The B2BUA helper class simplifies the usage of this pattern. It optionally links the two `SipSessions` and `SipServletRequests` when you use the `B2buaHelper` to create the new request.

```
SipServletRequest B2buaHelper.createRequest(
SipServletRequest origRequest,
boolean linked,
java.util.Map<java.lang.String, java.util.Set> headerMap)
```

The effect of this method when the `linked` parameter is true is to create a new `SipServletRequest` using the original request, such that the two `SipSessions` and the two `SipServletRequests` are linked together. When the two `SipSessions` and requests are linked, you may be able to navigate from one to the other.

The linked sessions can later be accessed as:

```
doSuccessResponse(SipServletResponse response) {
  ....
  otherSession = B2buaHelper.getLinkedSession(response.getSession());

  // do something on otherSession
  ....
}
```

getLinkedSession is a method defined on the B2buaHelper class. The helper works like a Visitor to the SipSession (and other classes) and encapsulates the functionality useful to a B2BUA implementation.

Similar to SipSessions, the linked SipServletRequest can be obtained from the method: B2buaHelper.getLinkedSipServletRequest(SipServletRequest).

Besides the B2buaHelper.createRequest() method, the linking can also be explicitly achieved by calling:

```
B2buaHelper.linkSipSessions(session1, session2) throws
IllegalArgumentExcetion;
```

IllegalArgumentException is thrown when sessions cannot be linked together, such as when one or both sessions have terminated or belong to different SipApplicationSessions or one or both have been linked to some different SipSessions.

Following method unlink other sessions that are linked with this session. -

```
B2buaHelper.unLinkSipSessions(session) throws IllegalArgumentException;
```

One SipSession at any given time can be linked to only one other SipSession belonging to the same SipApplicationSession.

The linkage at the SipServletRequest level is implicit whenever a new request is created based on the original with link argument as true. There is no explicit linking/unlinking of SipServletRequests.

15.2.2.2 Implicit Session Linkage

Another useful method on B2buaHelper for subsequent requests is:

```
B2buaHelper.createRequest(SipSession session, SipServletRequest
origRequest, java.util.Map<java.lang.String, java.util.Set>
headerMap) throws IllegalArgumentException
```

The session is the SipSession on which this subsequent request is to be sent. The origRequest is the request received on another SipSession on which this request is to be created. The headerMap can contain any non-system header which needs to be overridden in the resulting request. Any attempt to set a system header results in an IllegalArgumentException. The semantics of this method is similar to `SipSession.createRequest` This also results in automatically linking the two `SipSessions` (if they are not already linked) and the two `SipServletRequests`.

15.2.3 Access to Un-Committed Messages

A method on `SipServletMessage` defines the committed semantics for a message -

```
public boolean isCommitted();
```

The conditions under which a message is considered committed is detailed under 6.2 Implicit Transaction State.

A method defined on the new B2BUA Helper class gives the application a list of uncommitted messages in the order of increasing CSeq based on the UA mode. This is because there may be more than one request/response uncommitted on a `SipSession`.

```
List<SipServletMessage> B2buaHelper.getPendingMessages(SipSession,
UAMode);
```

The `UAMode` is an enum with values of `UAC` or `UAS`. The same session can act as `UAC` or `UAS`,and the `UAMode` indicates messages pertaining to which mode is queried.

For example, consider a B2BUA involved in a typical INV-200-ACK scenario that receives an ACK on one leg and wishes to forward it to the other. The B2BUA could call `B2buaHelper.getPendingMessages(leg2Session, UAMode.UAC)` to retrieve the pending messages which (as per 6.2 Implicit Transaction State, point 7) would contain the original 200 response received on the second leg. The B2BUA could then create the ACK by using the `SipServletResponse.createAck()` API. A PRACK request could be created in a similar way from a reliable 1xx response.

15.2.4 Original Request and Session Cloning

The incoming request that results in creation of a `SipSession` is called the original request.The application can create a response to the original request even if the request was committed and the application does not have a reference to this request. This is necessary because the B2BUA application may require to send more than one successful response to a request. For example, when a downstream proxy forked and more than one success responses are to be forwarded upstream. This can only be required on initial requests, as only original requests shall need multiple responses.

```
SipServletResponse
  B2buaHelper.createResponseToOriginalRequest(SipSession session, int
    status, String reasonPhrase) throws IllegalStateException;
```

The generated response MUST have a different "To" `tag` from the other responses generated to the request and must result in a different `SipSession`. In this (and similar) cases, the container clones the original `SipSession` for the second and subsequent dialogs, as detailed in 8.2.3.2 Derived SipSessions. The cloned session object contains the same application data but its `createRequest` method creates requests belonging to that second or subsequent dialog, that is, with a "To" `tag` specific to that dialog.

15.2.4.1 Cloning and Linking

In the case above when more than one response is received on the UAC side, it results in the cloned UAC `SipSession`. When the response is sent on the UAS side using the original request, it is in context of the cloned UAS `SipSession`. These `SipSessions` are thus pair-wise linked for easy navigation.

If UAS-1 is the `SipSession` on which the incoming request was received and UAC-1 is the `SipSession` on which the outgoing request was relayed, then in the case of multiple 2xx responses, one response is processed by the UAC-1 `SipSession.` When another 2xx response is received, the container MUST clone the UAC-1 `SipSession` to create UAC-2 `SipSession` to process this new response. The `B2buaHelper` has the convenient `getLinkedSession` method to navigate from UAS-1 to UAC-1 and vice versa. As for the cloned `SipSession` UAC-2, the `B2buaHelper` MUST be able to furnish the UAS-2 (a clone of UAS-1) when the `getLinkedSession` is invoked with UAC-2. Containers may clone the UAS-2 `SipSession` lazily. The applications can then create the response to the original request but now in the context of UAS-2 by invoking the method -

```
SipServletResponse
  B2buaHelper.createResponseToOriginalRequest(SipSession session-uas-2,
    int status, String reasonPhrase) throws IllegalStateException;
```

16 Converged Container and Applications

The SIP Servlet API facilitates the creation of SIP-based applications in an easy servlet programming model. The SIP Servlet API is based on the Servlet API and, thus, is a peer to the HTTP Servlet API. Many applications require the use of both SIP and HTTP protocols in a single application.For example, a conference call application providing a web portal for management and monitoring, a classic click-to-call application, and a CBSNA application are all applications that require both SIP and HTTP protocols. Therefore, using the SIP Servlet API and HTTP Servlet API together in a converged application is a natural fit. The HTTP Servlet API is closely associated with Java EE standards and is also based on real world requirements for a comprehensive convergence model.Thus, a converged container must support many features to enable not only SIP and HTTP convergence but also SIP Servlets and rest of Java EE. This chapter details the converged container features.

16.1 Converged Application

A converged application not only has a SIP Servlet component but also has at least one HTTP Servlet or Java EE component like an EJB.

A converged application having SIP and other non-HTTP Java EE components MUST be packaged as a single EAR (Enterprise Archive) file. SIP and Web components of a converged application SHOULD be packaged as a single SAR or WAR archive, either as a standalone archive or as part of an EAR archive. However, SIP and Web applications MAY be packaged into a single EAR archive file with different individual SAR or WAR files.

16.1.1 SIP and Java EE Converged Application

An application EAR file with its optional application.xml deployment descriptor is the precinct of a converged application. A converged application having a SIP and a Java EE component other than Web MUST be packaged and deployed as a single EAR archive file. EAR file is defined as an application delivery format for Java EE applications and is specified in Java EE specifications.

If the application.xml specifies the modules included in the application archive, the deployment descriptor's <web> element MUST be used for SIP Servlet applications. For example:

```
<module>
  <web>
    <web-uri>mysipapp.war</web-uri>
  </web>
</module>
```

If the application.xml deployment descriptor is not present, all files in the application package are considered converged SIP components if one of the following is true:

- The file extension is .sar
- The file extension is .war, and it contains the sip.xml deployment descriptor
- The file extension is .war, and it contains at least one class containing at least one SIP component defining annotation (e.g.,: `@SipServlet` or `@SipApplication`)

This application scope determines the extent of availability of convergence features in Java classes as detailed in the following sections on `SipFactory` and `SipSessionsUtil` injections.

16.1.2 SIP and HTTP Converged Application

The SIP and HTTP convergence is a special case because SIP Servlet and HTTP Servlet APIs have a shared base API and by virtue of that some Servlet API specific constructs are available to them for better convergence features. The converged SIP and Web application may be packaged as a single standalone SAR or WAR archive as described in 9 SIP Servlet Applications.

Converged SIP and Web applications must follow the following rules:

1. The archive MUST have the packaging directory structure as in 9.4 Deployment Hierarchies
2. SIP and HTTP Servlets MUST have the same view of the application, which includes:
   - Context parameters

– Context attributes
– Context-listeners and context-attribute listeners
– Application class-loader
– Application-scoped JNDI, which can be looked up against "java:comp/env" on the `InitialContext`.

Converged SIP and Web applications MAY also be packaged as independent SAR and WAR files respectively, within a single application EAR file. If this packaging structure is used, the convergence features are similar to SIP and Java EE components. Specifically, the benefits of a single SAR/WAR file of a shared context are not available.

16.2 Accessing SIP Factory

SIP servlet containers MUST make a `SipFactory` instance available to applications through a `ServletContext` attribute with the name "`javax.servlet.sip.SipFactory`" (See 3.2 The SipFactory). Access tothe `SipFactory` instance can be made by the servlet applications. To make `SipFactory` accessible to other Java EE components in the application, the container MUST makethe `SipFactory` instance available by means of an Annotations-based Dependency Injection.

The `@Resource` annotation defined in the Common Annotations for Java Platform (JSR250) is used for `SipFactory` injection.

When this annotation is applied to a field of type `SipFactory` in a converged application on a converged container, the container MUST inject an instance of `SipFactory` into the field when the application is initialized. This annotation can be used in place of the existing `ServletContext` lookup for the `SipFactory` from within a `Servlet`.

```
SipFactory sf =
  (SipFactory) getServletContext().getAttribute(SIP_FACTORY);
```

Usage above and below are equivalent.

```
@Resource
private SipFactory sf;
```

For converged containers, the injected `SipFactory` appears as `sip/<appname>/SipFactory` in the application scoped JNDI tree, where appname is the name of the application as identified by the container (see 9.6 Application Names).

In this example, the annotation can be used in any SIP Servlet or a Java EE application component deployed on the converged container in the same EAR archive file. The container must inject an instance of `SipFactory` into the field ("`sf`" in this example) during application initialization. For the purposes of associating a `Servlet`, with, say the responses received to the request created using the `SipFactory`, the container must use the application's main servlet as defined in 19.2 Servlet Selection. This can be changed to another servlet through the `setHandler` method of the `SipSession`.

Note that containers MAY allow the `@Resource` annotation to be present outside of SIP applications. In such cases, the name element of the `@Resource` annotation MUST identify the JNDI name of the desired factory to inject. Otherwise, an invocation of `SipFactory.createApplicationSession()` would be unable to determine the intended SIP application.

16.3 Accessing SipApplicationSession

Converged containers MUST provide a `SipSessionsUtil` lookup object to access the `SipApplicationSession` instances using any of the methods specified in this section. The reference to this lookup utility can be obtained from the `ServletContext` attribute `javax.servlet.sip.SipSessionsUtil` or by use of the `@Resource` annotation as described in 22.3.7 Annotation for SipSessionsUtil Injection.

Similar to the constraints on injection of `SipFactory`, none of the methods specified in this section can be used to access `SipApplicationSessions` outside the boundary of the application. If a module in an EAR file needs to access the `SipApplicationSession` of another module in the same EAR file, the application must lookup the corresponding `SipSessionsUtil` and use that object to access the `SipApplicationSession`.

`The SipApplicationSession` instance is identified by a container-specific ID, and it is frequently required to access the `SipApplicationSession` with its ID. Converged applications can access the `SipApplicationSession` using the method:

```
SipSessionsUtil.getApplicationSessionById(java.lang.String
applicationSessionId)
```

Similarly, a `SipApplicationSession` instance created with a specific application session key (using `SipFactory.createApplicationSessionByKey`) can be looked up by:

```
getApplicationSessionByKey(java.lang.String applicationSessionKey, boolean
create)
```

SIP Servlets can add and remove index keys to a `SipApplicationSession` by using `SipApplicationSession.addIndexKey(String indexKey)` and `SipApplicationSession.removeIndexKey(String indexKey)` methods respectively. Index keys are arbitrary keys that can be used to search for matching `SipApplicationSession`s. Thus, the same index key can be added to one or more `SipApplicationSession`s. A `Set` of IDs of all the `SipApplicationSession`s matching a specific index key can be retrieved by using the `SipSessionsUtil.getSipApplicationSessions(String indexKey)` method. Converged applications may retrieve the `SipApplicationSession` instance by using the `SipSessionsUtil.getApplicationSessionById(java.lang.String applicationSessionId)` method.
A `Set` of IDs of all the `SipApplicationSession`s of a particular application can be retrieved using the `SipSessionsUtil` of that application by `SipSessionsUtil.getApplicationSessions()` method.

16.4 Encoding URLs

The `SipApplicationSession.encodeURL()` encodes a URL (for example, HTTP URL) with the application session ID by using the `sipappsessionid` parameter. The encoded URL SHOULD have the application session ID encoded so that the parameter value which encodes the application session ID is unique across implementations. The recommended way is to use the implementation's java package name, like "`com.acme.appsession`".

Applications can use this mechanism to encode the HTTP URL with the `SipApplicationSession` ID. The encoded URL can then be sent out through some out of band mechanism to an external UA. When the HTTP Request comes back to the converged container with this URL, the container MUST associate the new `HttpSession` with the encoded `SipApplicationSession`.

If the HTTP request is not a new request but a follow-on request already associated with an existing HTTP session, converged containers MUST use the HTTP session association mechanism described in the HTTP Servlet specification, to route the request to the correct HTTP session. If the HTTP session is not associated with the encoded `SipApplicationSession` in the request, the association MUST be created by the container. This mechanism is similar to how the (deprecated) `encodeURI()` operates for SIP.

16.5 Association of HTTP Session With SipApplicationSession

8.6.1 Message Context defines the concept of a message context and how `SipSessions` are associated with `SipApplicationSessions`. Similarly, `HttpSessions` MUST be associated with `SipApplicationSessions` as described below.

- Even though the `SipFactory` instance is available in the HTTP scope through the context attribute or the annotation and using the `SipFactory` a new `SipApplicationSession` can be created, the recommended way of creating a `SipApplicationSession` is to use the `ConvergedHttpSession.getApplicationSession()` method.

  Converged containers MUST make available `javax.servlet.sip.ConvergedHttpSession`, an extension of `javax.servlet.http.HttpSession,` to the applications. Invoking this method causes the associated application session to be returned. However, if there is no associated application session, it is created, associated with the `HttpSession,` and returned.

- Another way to associate a new `ConvergedHttpSession` with a `SipApplicationSession` is to include an encoded URL with an existing `SipApplicationSession` in the request, as described in 16.4 Encoding URLs.

If the application contains a method annotated with @SipApplicationKey as specified in section 22.3.5 @SipApplicationKey Annotation, the container uses the string returned by this method to associate HTTP Session with `SipApplicationSession`.

If the container detects a method that accepts `HttpServletRequest` as the only argument, it invokes the method before invoking the HTTP Servlet's `service()` method.

If the method returns a non-empty string, the container uses that as the application session key. The `SipApplicationSession` corresponding to the returned application session key is associated with the request's `HttpSession` if the `HttpSession` is not yet associated with any other `SipApplicationSession` (for example, using the method described in 16.4 Encoding URLs.) If there is no corresponding `SipApplicationSession`, it is created when the container does the association.

Note that `HttpSessions` are often created lazily when the application actually accesses it from the `HttpServletRequest.` Hence, there may be no `HttpSession` associated with the request when the `service()` method is executed. In such cases, the actual association of the `HttpSession` with the `SipApplicationSession` happens only when the `HttpSessions` are actually created.

16.6 Finding Parent SipApplicationsSession From A Protocol Session

It is often required to access the parent `SipApplicationSession` from the protocol session (SIP or HTTP). `SipSession` has a method defined `getApplicationSession()` to access the parent `SipApplicationSession`. Applications can access parent `SipApplicationSession` from `HttpSession` by using the `javax.servlet.sip.ConvergedHttpSession` interface.

16.7 Encoding HTTP URLs With HTTP Sessions In Non-HTTP Scope

Given an application with multiple sessions (SIP and HTTP), it is often required to create an encoded HTTP URL such that the subsequent HTTP requests are routed to the correct HTTP session. This encoding is different from the one defined in 16.4 Encoding URLs because the `HttpSession` ID is encoded with parameter "`jsessionid`" as defined in the HTTP Servlet specification. The methods for this are defined on the `ConvergedHttpSession` interface.

17 Container Functions

17.1 Division of Responsibility

One of the goals of the SIP Servlet API is to keep the application programming model simple (see 1.1 Goals of the SIP Servlet API). This is accomplished by delegating a number of tasks to the container and freeing up the applications to focus on their business logic. Tasks such as management of network listen points, SIP message and transaction processing, and handling of headers (see 6.4.2 System Headers), like Via, Contact, CSeq, RSeq, and Path, are handled by containers.

As a rule of thumb, functionality that is "behind" the SIP Servlet API is managed by the container. For example, the API exposes the `SipSession` interface and allows applications to perform operations like creating in-dialog requests, getting and setting attributes, and invalidation on `SipSession` but it does not allow the application to create a new `SipSession` explicitly. `SipSession` is an object managed by the container and is created as a result of incoming request processing or creation of a new request by the application.

The API strives to strike a balance between providing the ability to write simple SIP Servlet applications and allowing for powerful constructs using the base API.

Consider the example of SIP-specific Event handling [RFC 6665]. The SIP Servlet API provides for handling of SUBSCRIBE and NOTIFY requests. Since these requests are dialog creating the container manages these SIP dialogs transparently for applications. However, applications maintain the "Subscription", which is the application-specific property. RFC 6665 specifies a mechanism for installation of Subscription, namely SIP SUBSCRIBE. The Subscription duration is specified either through the SUBSCRIBE request's "Expires" header or through the specific event package. It is also possible for the application to accept the Subscription for a shorter duration and convey the same in the 200 OK Response "Expires" header. Containers provide primitives like a Timer API, access to Session, and Request to facilitate the applications carrying out such functions.

The previous two examples illustrate the care taken to provide the right level of abstraction to the application developer. Having said that, it is expected that some repetitive use cases will be found and some patterns will have to be re-used within SIP Servlet Applications. This specification provides a powerful mechanism for application composition [see 18 Application Selection And Composition Model].

17.1.1 Protocol Compliance

The responsibility of protocol compliance is again divided between the container and applications. The container enforces the protocol behavior of container managed objects and state, like sessions, transactions, dialog state, and address/URI formats. Applications influence the state changes by exercising API methods. In some cases, it is possible that the underlying state machine is in disagreement with the action that the application wishes to perform. In these cases, the container MUST throw `java.lang.IllegalStateException`. These actions include not only the base RFC 3261 assertions and state machine but also the API constraints, expressed also as `java.lang.IllegalArgumentException`. As much as possible, the more obvious of these violations are described in the API documentation. However, it is not possible to list every situation where an exception may be thrown. Implementors MUST make sure that the protocol behavior of all the container managed objects is consistent with the SIP RFCs they comply with.

Further, the API declares the `Exceptions` that get thrown when either the state, context or parameters are not correct. Implementors MUST throw these exceptions when the specified conditions occur. As a general guideline, even for the conditions that are not mentioned specifically in the API, the implementations SHOULD perform the following:

- Throw an `IllegalArgumentException` when any argument in the method call is not suitable in the context, or the argument is an object the state of which is not suitable for the context.
- Throw a `NullPointerException` where the method tries to set an attribute of the object, and the null attribute is not allowed by the domain object.
- Throw an `IllegalStateException` where the method receiver (object on which the method is called) is not in a state where the invocation is suitable.
- Throw a `ConcurrentModificationException` if the base collection is modified when the `Iterator` is being used, unless the collection is otherwise concurrent.

The idea is to provide an environment of least surprise to the application developer.

17.2 Multi-homed Host Support

Containers must manage the network listen points. An application might use the information about the inbound interface to decide about invocation of specific features, or an application may decide to mandate the use of a certain outbound interface to send the message to a specific domain or host.

The following shows four use cases for a deterministic selection of a network interface on a multi-homed host:

- The container executes on a multi-homed host, with one interface for SIP signaling and a different interface for management. The container should only receive and send SIP messages on the first interface.
- The container executes on a multi-homed host, which has a network interface on a trusted domain and another network interface on an untrusted domain. These domains may represent an internal and an external network. The application logic could differ based on whether a request comes from a trusted or an untrusted element. The application must be able to identify the inbound address on which the message was received. The use of the remote address or its type is not sufficient, as the address is probably assigned dynamically and both networks may use either public or private address ranges.
- The container executing on a multi-homed host, connected to a trusted and an untrusted network domain (same setup as in use case 2), is sending out a request. For security requirements, internal and external traffic, e.g. traffic to own servers and external customers has to be separated on a physical level. Thus, the application needs the ability to mandate the use of a particular outbound interface.
- The container may execute on a multi-homed host that has network interfaces into a number of networks that are separated on the physical layer. Each network interface is on a separate IP address and each may route to the same addresses. Some applications may choose the network by selecting an interface rather than allowing the operating system routing tables to automatically handle the selection. To send originating requests to the correct network, the application needs the ability to mandate the use of a particular outbound interface.

Applications can access the list of container supplied SIP URIs for sending outbound requests. Applications can access the list through the`ServletContext` attributes `javax.servlet.sip.outboundInterfaces` and `javax.servlet.sip.outboundAddresses. The SipServletContext` interface has equivalent methods for accessing the same information.

The container MUST implement a method to set the outbound interface for a certain session `setOutboundInterface(InetAddress address)`.The container must then use this interface for all messages associated with that `SipSession`. This method is provided on `Proxy`, `ProxyBranch`, and `SipSession` objects. Invoking this method impacts the system headers, such as the Via and Contact headers, generated by the container for this `SipSession`. The supplied IP address is used to construct these system headers.

This setting may be overwritten by subsequent calls to the method. In case there is no specific outbound interface set, the container shall apply default behavior of interface selection.

After the interface is selected on a multi-homed host, the SIP servlet container must populate the Systems Headers (see 6.4.2 System Headers) accordingly.

17.2.1 Application Composition on Multihomed Container

In case there are multiple applications involved in a call, the `setOutboundInterface()` method must adhere to the following requirements (see 18 Application Selection And Composition Model for details):

- For the last application in the chain acting as a proxy, if it does not call `setOutboundInterface()`, it uses the setting from previous applications (if any).
- For the last application acting as a proxy, if it invokes `setOutboundInterface()`, it overrides the setting by the previous applications (if any).
- If any application in the chain acts as a UAS (or a B2BUA), the previous application’s setting (if any) is no longer relevant.

It is expected that the interface IP addresses returned by the `ServletContext` parameters `javax.servlet.sip.outboundInterfaces` and `javax.servlet.sip.outboundAddresses` shall include the IP addresses of virtual interfaces or publicly observable IP addresses of the load balancers in case of clustered setup. However, it shall be up to the applications to decide which interface to use.

This mechanism of explicit outbound interface selection is to be used by the applications that are aware of the network topology by virtue of some configuration or other means and have strong reasons to use this mechanism. It is expected that most of the applications shall not need to use this advanced feature.

17.3 SIP Servlet Concurrency

Multiple Servlets executing simultaneously may have active access to shared resources like the `SipSession` and `SipApplicationSession` objects. These resources may also be accessed concurrently from `ServletTimer` objects, from other Java EE modules in a converged Java EE application, and by the SIP Servlet Container. Operations, such as updating values of session attributes, carried out by different threads on these objects can create concurrency issues, including deadlock, for some applications. This section explains ways to develop portable applications without concurrency issues.

17.3.1 Specifying Concurrency Mode

A SIP Servlet application can specify the required level of concurrency control the container should apply while executing applications in the `SipApplication` annotation or in the deployment descriptor. This specification supports two values for the `ConcurrencyMode` enum: `VENDORSPECIFIC` and `APPLICATIONSESSION`. The following shows an example of a servlet specifying the concurrency mode.

```
package com.example;
import javax.servlet.sip.SipServlet;
@SipApplication (name = "Foo", concurrencyMode =
ConcurrencyMode.APPLICATIONSESSION)
```

Table 17-1 Concurrency Modes

VENDORSPECIFIC	Indicates that the container can choose any concurrency level it deems appropriate. The level of guarantee of thread safety is determined by the SIP Servlet Container. This is the default
APPLICATIONSESSION	Indicates that the container performs concurrency control at the level of the application session. It ensures that two messages belonging to the same application session are never processed simultaneously. It also ensures that various timer tasks or internal threads managed by the container that access the application session are not executed at the same time.

17.3.2 Concurrency Utilities

The *Concurrency Utilities for Java EE* specification explains how an application can obtain Concurrency Utilities in an enterprise server environment. A SIP servlet container uses these utilities to execute asynchronous tasks and to develop thread safe applications. A SIP Servlet application may use any of the Managed Objects (`ManagedExecutorService, ManagedScheduledExecutorService, ContextService and ManagedThreadFactory)` specified by the *Concurrency Utilities for Java EE* specification. SIP servlet implementations built on a Java EE 7 platform are required to support the functionality described in this section.

17.3.2.1 Propagating SipApplicationSession Context

When the SIP Servlet application specifies a concurrency control of `ConcurrencyMode.APPLICATIONSESSION,` the container is expected to make sure that appropriate concurrency control is maintained. To effectively maintain the concurrency control at the SipApplicationSession level, the active SIP application session also needs to be passed as the context information.

To enable this, a SIP Servlet container exposes one or more `ManagedExecutorService` and `ManagedScheduledExecutorService` objects that execute submitted tasks in a thread pool managed by the container. While executing the tasks, containers make sure that concurrency control specified by the application is maintained.

Applications are strongly recommended to use the default managed objects as specified in section 17.3.2.7 Default Managed Objects to propagate a SIP Application Session.

When a servlet (or any other part of the application) submits a task using the default `ManagedExecutorService` or `ManagedScheduledExecutorService`, apart from the context information specified in the Concurrency Utilities for EE specification, the active application session is also passed as the context information. The active application session is the one responsible for the execution of the thread where the application logic is running. Some examples are application session of the `SipServletMessage` that triggered servlet execution, application session of the executing `ServletTimer`, relevant application session of the application listener, and application session of `ConvergedHttpSession` of the HTTP Servlet. If the task is submitted from a thread where no application session is active, then application can use the mechanism specified in the next section for specifying an application session as the context.

17.3.2.2 Specifying Application Session Programmatically

A thread might want to submit the task with a different application session as the context than the one active when the task is submitted. In this case, the application is expected to create a contextual object proxy by using `ContextService` for submitting the task or by using `ManagedTask`. Refer to the *Concurrency Utilities for Java EE* specification for more information about the `ContextService` and `ManagedTask` interfaces. The application can then specify an application session of its choice as the context by using one of the following execution properties:

- `javax.servlet.sip.ApplicationSessionKey`: Specifies the SIP application key.
- `javax.servlet.sip.ApplicationSessionId`: Specifies the application session ID.
- `javax.servlet.sip.ApplicationSession.create`: Indicates that the container creates a new `SipApplicationSession` and use it as the context.

Applications can also access these constants from `javax.servlet.sip.SipServlet` fields `SIP_APPLICATIONSESSION_KEY`, `SIP_APPLICATIONSESSION_ID` and `SIP_APPLICATIONSESSION_CREATE` respectively.

To avoid concurrency issues, applications are required to submit asynchronous tasks whenever they use an application session different from the active application session. This rule applies for all kinds of application components, such as SIP Servlets, HTTP Servlets, other Java EE components, Servlet Timers, and asynchronous tasks.

If an application specifies both `javax.servlet.sip.ApplicationSessionKey` and `javax.servlet.sip.ApplicationSessionId,` then `javax.servlet.sip.ApplicationSessionId` take precedence. If the application session specified by either `javax.servlet.sip.ApplicationSessionKey` or `javax.servlet.sip.ApplicationSessionId` is invalid or cannot be found, the container aborts the execution of the task and `ManagedTaskListener.taskAborted` is called with an `AbortedException`.

The container will discard `javax.servlet.sip.ApplicationSession.create,`if `javax.servlet.sip.ApplicationSessionId` is also specified by the application. If only `javax.servlet.sip.ApplicationSession.create` is specified, the container will create a new `SipApplicationSession` and use it as the context. If both `javax.servlet.sip.ApplicationSession.create` and `javax.servlet.sip.ApplicationSessionKey` are specified, then container MUST use the specified application session key while creating the `SipApplicationSession`. If the container finds an existing application session with the specified application key, that application session will be used as the context.

SIP Servlet containers ensure that tasks are run in a thread safe manner with respect to the sip application sessions specified as the context. However, application should be careful while sharing the objects between tasks with different application session context. Thread safety of such objects, whether it is an application specific object or an object received from the container, need to be maintained by the application. For example, objects like `SipURI`, `Address` may be cloned before sharing it between the tasks.

17.3.2.3 ContextService

As specified in section 3.3 of the *Concurrency Utilities for Java EE* specification, `javax.enterprise.concurrent.ContextService` allows applications to create contextual objects without using a managed executor. Just like a submitted contextual task, whenever a method on the contextual object is invoked, the method executes with the thread context of the associated application component instance.

If a contextual proxy created using the default SIP context service (section 17.3.2.7 Default Managed Objects) and that contextual proxy is executed in a thread known to the SIP container (e.g., HTTP Servlet thread or MDB thread), the application session context is set properly. This application session context is based on the thread that created the contextual proxy. The application may use the execution properties mentioned in section 17.3.2.2 Specifying Application Session Programmatically to use a different application session as the context.

Note: If the contextual proxy is executed directly on a thread that already has an active SIP application session and the contextual proxy is created with another SIP application session as the context, the container throws `IllegalStateException`.

17.3.2.4 Impact on ApplicationSession Invalidation.

If an application invalidates the application session by using `SipApplicationSession.invalidate()` after the task is submitted, the container invokes `Future.cancel(false).`

17.3.2.5 Example Usage 1. Submitting a task with active SipApplicationSession Context.

```
@SipServlet
public class FooPOJO{

    @Resource(lookup="java:comp/ManagedSipExecutorService")
    ManagedExecutorService mes;

    @Invite
    protected void onInvite(SipServletRequest req) {

        // Create a task instance. MySipTask implements Callable.
        MySipTask sipTask = new MySipTask();

        // Submit the task to the ManagedExecutorService
        Future sipFuture = mes.submit(sipTask);
    }
}
```

17.3.2.6 Example Usage 2. Scheduling a task with a different SipApplicationSession Context.

```
@SipServlet
public class FooPOJO{

    @Resource(lookup=”java:comp/ManagedScheduledSipExecutorService”)
    ManagedScheduledExecutorService mses;

    @Resource(lookup=”java:comp/SipContextService”)
    ContextService ctxSvc;

    @Invite
    protected void onInvite(SipServletRequest req) {

        SipApplicationSession sas = //...lookup using SipSessionsUtil

        // Set any custom context data through execution properties
        Map<String, String> execProps = new HashMap<>();
        execProps.put("javax.servlet.sip.ApplicationSessionId",
sas.getId());

        // Create a task instance. MySipTask implements Callable.
        MySipTask sipTask = new MySipTask();

        Callable task = ctxSvc.createContextualProxy
                (sipTask, execProps, Callable.class);

        // Submit the task to the ManagedScheduledExecutorService
        Future sipFuture = mses.submit(task);
    }
}
```

17.3.2.7 Default Managed Objects

SIP Servlet containers must provide the preconfigured, default managed objects shown in Table 17-2.

Table 17-2 Default Managed Objects

`ManagedExecutorService`	`java:comp/ManagedSipExecutorService`
`ManagedScheduledExecutorService`	`java:comp/ManagedScheduledSipExecutorService`
`ContextService`	`java:comp/SipContextService`
`ManagedThreadFactory`	`java:comp/ManagedSipThreadFactory`

The types of contexts to be propagated by these default objects must include naming context, classloader, and security information apart from the application session. Note that an application MUST not use these default managed objects if they do not want the application session context to be propagated.

17.3.2.8 Accessing active application session

It is often useful to access the application session that is active on the current thread, be it the thread running a submitted or scheduled task or from the thread executing the servlet. Though it is possible to access `SipApplicationSession` by using `SipSessionsUtil`, the ID, or application key as specified in section 16.3 Accessing SipApplicationSession, the SIP Servlet container provides more direct access to the active application session by using the `SipSessionsUtil.getCurrentApplicationSession()` method. This is especially useful when the tasks are not CDI managed beans and hence CDI built-in beans for injecting `SipApplicationSession` cannot be used. More information about the CDI built-in beans can be found in section 22.5.3 SIP Specific CDI Beans.

17.3.2.9 Accessing Futures of tasks

It is often useful to access the `Future` (`java.util.concurrent.Future`) objects of the tasks, submitted or scheduled by using a `ManagedExecutorService` or `ManagedScheduledExecutorService`, that are not completed nor canceled. Applications can access futures of such tasks corresponding to the `SipApplicaationSession`by using the `SipApplicationSession.getTaskFutures()` method. It is also possible to access a `Future` of a specific task by using `SipApplicationSession.getTaskFuture(String identityName)`. Note that to use this method successfully, applications are required to provide a unique identity name within the `SipApplicationSession` for the `javax.enterprise.concurrent.ManagedTask.IDENTITY_NAME` execution property while submitting or scheduling the task.

17.4 Managing Client Initiated Connections

SIP outbound specification [RFC 5626] specifies techniques for keeping connections that UA establishes alive, especially in environments where the UA is behind a NAT device or firewall. A SIP Servlet application may take different roles with respect to those defined in RFC 5626 for keeping the connections alive. For example, a SIP Servlet application may act as an authoritative proxy, an edge proxy, a registrar, or a UA. Refer to RFC 5626 for more details about the exact procedure for managing client- initiated connections.

SIP Servlet containers support sending keep alive messages from a user agent as well as responding to keep alive messages defined in RFC 5626. They also support a `Flow` interface for managing the connections as described in the following sections.

> Support for SIP outbound is optional for SIP Servlet containers. If implemented, the container must include the string “outbound” in the list of supported extensions available through the ServletContext, see 3.3 Extensions Supported.

17.4.1 Flow Interface

As per RFC 5626, a flow is a transport-layer association between two hosts that is represented by the network address and port number of both ends and by the transport protocol. For TCP, a flow is equivalent to a TCP connection. For UDP, a flow is a bidirectional stream of datagrams between a single pair of IP addresses and ports of both peers.

From a SIP Servlet application’s perspective, flow is a transport layer association between the SIP Servlet container and another SIP endpoint on a specific transport.

Table 17-3 Flow interface methods

`getToken()`	Retrieves the flow token representing a flow.
`getLocalURI()`	Retrieves the `SipURI` representation of the local end of transport association.
`getRemoteURI()`	Retrieves the `SipURI` representation of the remote end of transport association.

Table 17-3 Flow interface methods

`release()`	Indicates to the container that the application will not use the Flow object any more.
`isActive()`	Indicates whether the flow is active or not. Flow becomes inactive when it is closed.

Applications may get an instance of `Flow` that represents this transport layer association by using the `SipSession.getFlow(), Proxy.getFlow(), or ProxyBranch.getFlow()` method.

17.4.1.1 Retrieving a flow object from the container

A SIP Servlet application can retrieve the `Flow` object corresponding to a flow token from the container by using the `SipServletContext.getFlow(String flowToken)`method.

17.4.1.2 UAC Sending Keep-Alive

If a SIP Servlet acting as a UAC sends a REGISTER request with the headers required for RFC 5626 support, on receiving a 200 response to such a request, if that response contains an outbound option-tag in the Require header field, the container starts sending keep-alive messages. These keep-alive messages may be double CRLFs for connection-oriented transports, such as TCP and STUN binding requests as described in RFC 5626. While sending keep-alive messages, the SIP Servlet container is required to handle the `Flow-Timer` header as specified by RFC 5626. The following is an example implementation of such a UAC:

```
void sendRegister(SipServletRequest register) throws Exception {
        register.setHeader("Supported", "path,outbound");
        // Add the instance id
       Parameterable contact = register.getParameterableHeader("Contact");
        contact.setParameter("+sip.instance", <instanceId>);
        // Add the reg id
        contact.setParameter("reg-id", <registrationId>);
        register.send();
    }


//Container start sending keep-alive messages.
    @SuccessResponse
    public void handle200ok (SipServletResponse resp) {
        Flow flow = resp.getSession().getFlow();
    }
```

17.4.1.3 Handling Flow Failures

An instance of `FlowListener` may be configured as a `SipListener`. Whenever the flow fails, the container invokes all configured `FlowListener`s. Applications may retrieve the flow and complete the necessary business logic, which typically includes sending the REGISTER message again. The `FlowListener` and `FlowFailedEvent` definitions are given below.

```
public interface FlowListener extends EventListener {
    void flowFailed(FlowFailedEvent flowFailedEvent);
}
public class FlowFailedEvent extends EventObject {
    public FlowFailedEvent(Flow flow) {
        super(flow);
    }

    public Flow getFlow() {
        return (Flow) super.getSource();
    }
}
```

17.4.1.4 Reusing the Flow

UAC or UAS may specify the flow to send messages on by using one of the following methods.

- `SipSession.setFlow(Flow flow).`
- `Proxy.setFlow(Flow flow)`
- `ProxyBranch.setFlow(Flow flow)`

For example, a UAC can specify that the same flow that is used for sending REGISTER messages be used for sending INVITE messages.

```
void sendInvite(SipServletRequest invite){
        Flow flow = //Retrieve the flow
        invite.getSession().setFlow(flow);
        invite.send();
    }
```

17.4.1.5 Edge Proxy

When an edge proxy is implemented by using SIP Servlets, the application inserts the flow token into the request as explained in RFC 5626. The application can retrieve flow tokens from the `Flow` object for this purpose. The following is an example of how an application might implement this logic:

```
@Register
public void handleRegister(SipServletRequest request){
    Proxy proxy = request.getProxy();
    Flow flow = proxy.getFlow();
    proxy.setAddToPath(true);
    SipURI pathURI = proxy.getPathURI();
    pathURI.setUser("edge");
    pathURI.setParameter("flow", flow.getToken());
    pathURI.setParameter("ob", "true");
    proxy.proxyTo(request.getRequestURI());
}
```

Similarly, when a different request that needs to be forwarded to UA arrives at the edge proxy, it can look up the Flow object by using the flow token present in the request. After the retrieved flow object is set on the `Proxy` (or `ProxyBranch`) by using the `Proxy.setFlow(Flow flow)` or `ProxyBranch.setFlow(Flow flow)` method, the SIP Servlet container uses the corresponding transport association to send the message. The following example code explains one such implementation:

```
@Invite
public void handleInvite(SipServletRequest request)  {
    Address route = request.getPoppedRoute();
    String flowToken = route.getURI().getParameter("flow");
    Proxy proxy = request.getProxy();
    Flow flow = sipServletContext.getFlow(flowToken);
    proxy.setFlow(flow);
    proxy.proxyTo(request.getRequestURI());
}
```

17.4.1.6 Releasing a Flow

The container decides when to close the transport association linked with the flow based on the container-specific configuration. For example, a container may choose to terminate the transport association when there are no messages on the flow for a prolonged period of time. An application can provide a hint to the container that it is no longer interested in using `Flow` by invoking the `Flow.release()` method. The container may use this hint while making the decision to stop the keep-alive and/or terminating the underlying transport association. Since more than one application may be using the same transport association, `Flow.relase()` should not be considered equivalent to terminating the transport association.

18 Application Selection And Composition Model

SIP Servlet application servers are typically provisioned with many different applications. Each application provides specific functionality, but, by invoking multiple applications to service a call, the deployer can build a complex and complete service. This modular and compositional approach makes it easier for application developers to develop new applications and for the deployer to combine applications from different sources and manage feature interaction. A typical example from traditional telephony is a call-screening application and a call-forwarding application. If the application server receives an incoming INVITE destined to a callee who subscribes to both services, both applications should be invoked.

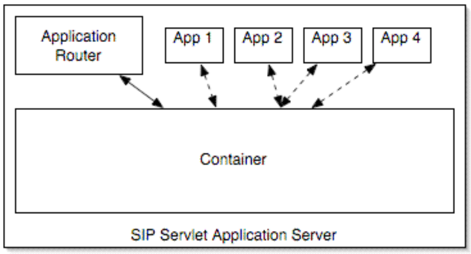
One key requirement for application composition is that the correct set of applications be invoked in the correct order to service a call. Therefore, an important function of SIP Servlet application servers is to select applications for invocation and route SIP messages to them. When determining which applications to invoke and in which order, containers treat initial requests, subsequent requests, and responses differently. Generally speaking, an initial request is a request for which the container has no prior knowledge. It may or may not be a request capable of establishing a dialog [RFC 3261, Section 12]. Appendix B contains a precise definition of initial requests. Initial requests are routed based on an application selection process. The routing of an initial request establishes a path of applications. Subsequent requests and responses are then routed along the path taken by the initial request.

The principles of application independence and composition described in this section are adapted from the Distributed Feature Composition (DFC) architecture [DFC1998], [CN2004], [SE2005]. The original definition of DFC [DFC1998] was improved as a result of extensive experience with it.

18.1 Application Selection Process

A key component of the SIP application selection procedure is a logical entity called the Application Router. The Application Router plays a central role in the SIP application selection process. A SIP Servlet application server is thus made up of the container, the Application Router, and a number of applications, as shown in Figure 18-1. The concerns of these three entities are cleanly separated, and this section discusses their separate roles.

Figure 18-1 The SIP Servlet Container, Application Router, and Applications



18.1.1 The Role of the Application Router

The Application Router is called by the container to select a SIP Servlet application to service an initial request. It embodies the logic for choosing which applications to invoke. An Application Router is required for a container to function, but it is a separate logical entity from the container. The Application Router is responsible for application selection and must not implement application logic. For example, the Application Router cannot modify a request or send a response.

The Application Router implements the `SipApplicationRouter` interface, which defines the API between the container and the Application Router. There is no direct interaction between the Application Router and applications. It is also important to note that, besides the information passed by the container, the Application Router is free to use other information or data stores. How it accesses that information and what information it uses depends on its implementation and is not defined in this specification.

The role of the deployer is defined in the Servlet API. The deployer in a SIP Servlet environment controls application composition by defining and deploying the Application Router implementation. Giving the deployer control over application composition is desirable because it is the deployer who is most aware of and responsible for the totality of services provided to his or her subscribers. Furthermore, this specification intentionally allows the Application Router implementation to consult arbitrary information or data stores. This is because the deployer maintains subscriber information, which is often private and valuable.

18.1.2 The Role of Applications

In contrast to the deployer, application developers are concerned with the application logic of individual applications. An application executes independently of other applications and should not make any assumptions about the presence or location of other applications. This promotes modularity and re-use and facilitates application development.

As part of its application logic, an application may choose to proxy initial requests, relay initial requests as a B2BUA, terminate initial requests as a UAS, or send initial requests as a UAC. Although these actions influence application composition, the decision of which application to invoke for a request issued by an application rests solely with the Application Router. It follows that the Application Router must be aware of the application's intention when it sends a request. In some cases, it can be implicitly inferred. However, in some cases the application MAY use a new API call to indicate its intention explicitly. This is crucial for correct application composition and is discussed in section 18.2.2 Sending an Initial Request.

18.1.3 The Role of the Container

The container receives initial requests from external entities or from an internal application, calls the Application Router to obtain the application to invoke, and then invokes the selected application.

Since an Application Router is required for the functioning of a container, all compliant container implementations MUST provide at least a default Application Router as defined in Appendix C. The default Application Router enables the support of simple application composition based on a configuration file. Container implementations MAY also provide alternative implementations of Application Routers. It is expected that implementations each have their Application Routers suited to the needs of a deployment situation and have means to configure them.

The container informs the Application Router when new SIP applications are deployed or undeployed by using the `SipApplicationRouter.applicationDeployed()` and `SipApplicationRouter.applicationUndeployed()` callback methods respectively. Each SIP application packaged within an EAR is registered or unregistered with the Application Router individually.

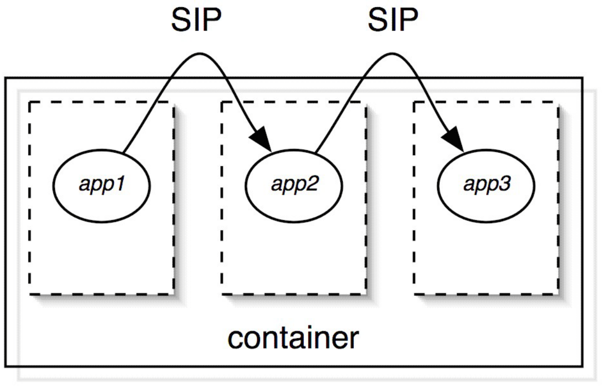
18.1.4 Application Independence

When the container receives an initial SIP request from an external SIP entity, the container invokes the application selection process to select the first application to service this request. If this first application subsequently proxies the request, or if it acts as a B2BUA and on the UAC side sends a new initial request, the container again invokes the application selection process to select and invoke the next application, often within the same container. In this way, as applications proxy or send requests, a chain of applications is created. The chain ends when either an application acts as a UAS or when no more applications are selected and the request is sent to an external SIP entity. It is important to note that the application composition happens as a side effect of applications sending the requests and no special APIs are used for composition.

There are two very important considerations that guide the way application composition happens on SIP Servlet containers.

- Each application in a chain shall see a consistent view of the context in which it is operating regardless of how many other applications are involved in the same communications session. Because the context is consistent regardless of the presence of other applications, application developers are guaranteed that they can write an application as if it were the only one involved in the communication session. Applications should not know nor need to know whether a request they send or proxy is going to be sent directly outside the container or to another application.
- An application composition chain shall be composed independent of the location of the application. The constituent applications in the chain can be spread over multiple container instances. In other words, the invocation sequence should be independent of deployment.

Figure 18-2 Maintaining SIP Context with Multiple Applications Active on Same Container



Note: The SIP Servlet applications are modeled as isolated, individually deployable components of larger composite applications or services. It is foreseeable that in large systems which maximally benefit from the innovations of unrelated developers and development projects, the relationship between deployed SIP Servlet application implementations will be defined solely by the deployer and/or the implementor of the Application Router. Composition of those SIP Servlet applications into application usages of the SIP network itself is facilitated in this way, and the developer of a given SIP Servlet application should make no assumptions about the model for deployment and method of composition of these composite applications. In such cases it is possible that SIP Servlet applications may be deployed within a single SIP Servlet container or may be deployed in multiple SIP Servlet containers, each representing a different but compliant implementation of this specification.

18.1.5 Subscriber Identity and Routing Regions

SIP Servlet applications typically act on behalf of subscribers. Some applications are only invoked when their subscriber is originating a call(i.e., the subscriber is the caller). Examples include Speed Dial and Outbound Call Screening. Other applications are only invoked when their subscriber is receiving a call(i.e., the subscriber is the callee). Examples include Call Blocking and Location Service. Finally, there are applications that are invoked for both calling and called subscribers. For example, Call Waiting and 3-Way Calling are invoked when their subscribers are either placing or receiving a call. There are also applications that provide services independent of any particular subscriber. For example, a call logging application may be invoked for all calls, regardless of whether the subscriber is a caller or callee.

The concepts of Subscriber Identity and Routing Region are introduced here so that an application may learn in which region it is invoked and on behalf of which subscriber. When the Application Router selects an application to service a request, it MUST specify the routing region in the `SipApplicationRouterInfo` object that it returns to the container. The routing region can be one of ORIGINATING_REGION, TERMINATING_REGION, NEUTRAL_REGION, or one of their sub-regions. The Application Router MUST also specify the subscriber identity in the `SipApplicationRouterInfo` object when the routing region is set to either ORIGINATING_REGION or TERMINATING_REGION. The Application Router MAY specify the subscriber identity when the routing region is set to NEUTRAL_REGION. The subscriber identity is a javax.servlet.sip.URI object. The container MUST make the subscriber identity and routing region values as set by the Application Router available to the selected application by using the `SipSession.getSubscriberURI()` and `SipSession.getRegion()` methods.

In most cases, an application selected to serve the calling subscriber is invoked in the originating region, while an application selected to serve the called subscriber is invoked in the terminating region. Often, an application not designed to serve any specific subscriber (such as the call logging application mentioned above) is invoked in the neutral region and the subscriber identity is not set by the Application Router. (Note that the Application Router sets the routing region and subscriber identity along with the application names specified in section 9.6 Application Names on the `SipApplicationRouterInfo` object that is returned to the container as part of the `SipApplicationRouter.getNextApplication()` call.)

The routing regions are extensible and an implementation of the Application Router may further subdivide these regions into smaller regions if necessary. For example, one may define new sub-region, - TERMINATING_REGISTERED and TERMINATING_UNREGISTERED, within the terminating region by extending `SipApplicationRoutingRegion` as shown below:

```
TerminatingRegisteredRegion extends SipApplicationRoutingRegion {
  public TerminatingRegisteredRegion () {
    super("TERMINATING_REGISTERED",TERMINATING_REGION);
  }
}

TerminatingUnRegisteredRegion extends SipApplicationRoutingRegion {
  public TerminatingUnRegisteredRegion () {
    super("TERMINATING_UNREGISTERED",TERMINATING_REGION);
  }
}
```

Deployers could thus define applications to be invoked in the above sub-regions. An application like the Location Service application defined in the TERMINATING_REGISTERED sub-region is invoked for subscribers whose terminals are registered with the registrar; while applications like the Voicemail application defined in the TERMINATING_UNREGISTERED region is invoked only for those subscribers who have not yet registered.

18.1.6 Routing Directives

Routing directives are provided by applications when they send or proxy an initial request. Routing directives provide a means for applications to indicate their intention clearly to the Application Router, which can then perform application selection accurately. Sections 18.2 through 18.4 discuss routing directives in more detail.

18.2 Application Environment and Behavior

This section is intended for application developers. The concepts relevant to application developers are discussed, including how an application indicates its routing intention, and what additional information related to application composition an application can obtain from the container.

18.2.1 Receiving an Initial Request

When an application is invoked to service an initial request, the container calls the application's doXXX() method with the `SipServletRequest` in a new `SipSession`. The application can obtain the identity of the subscriber that the application is serving and the routing region in which it is invoked from the `SipSession` object. These attributes do not change throughout the lifetime of the `SipSession` object.

18.2.2 Sending an Initial Request

Whenever an application proxies or sends an initial request, the container invokes the Application Router to obtain the name of the next application that should service the request. The Application Router needs to know whether the sent request is related to a request that the application has received earlier. Though related, the new request could be a modified version of the originally received request or a brand new request. If the sent request is not related to any received request, the Application Router starts a new application selection process. On the other hand, if the sent request is related to a received request, the Application Router uses the state information associated with the received request and resumes the selection process from the previous state.

This is crucial to the selection of applications. As an example, consider a deployment where all initial INVITE requests received by the container are serviced by two applications, A and B, in this order. When an INVITE request is received externally, or if an application Z acts as a UAC and sends an INVITE request, the container invokes the Application Router. The Application Router starts the selection process from the beginning and selects A. When A is invoked and proxies the INVITE request, the container invokes the Application Router again. Now, the Application Router must be made aware that this is related to the previous request so that it can determine that A has already been invoked and that B must be invoked as the next application.

When an application acts as a UAC and sends a request, the request is not based on any previously received request. That is, the application intends the request to be regarded as a new request unrelated to any other request. Although the application intention is implicit in this case, it is clear that a new application selection process should take place.

Listing 18-1 Application acts as UAC and sends initial request

```
        +--------------------------------------+
        |                                      |
        | req = factory.createRequest          |
        |        (appSession,                  |  req
        |         "INVITE", from, to);         |------>
        | req.send();                          |
        |                                      |
        +--------------------------------------+

        req starts a new selection process
```

In the case of a proxy application, there is no ambiguity. It is clear that the application's intention is to send the request along the way, or in other words to continue the call.

Listing 18-2 Application proxies an initial request

```
        +------------------------------+
  req1  |                              |   req2
------->|  proxy = req1.getProxy();    |--------->
        |  proxy.proxyTo(...);         |
        |                              |
        +------------------------------+
        req2 is a continuation of req1
```

If the application acts a B2BUA and uses the SipFactory.createRequest(`SipServletRequest` origRequest, boolean sameCallId) or B2buaHelper.createRequest(`SipServletRequest` origRequest, Map headers) methods, it is clear that the application's intention is to continue the call.

Listing 18-3 Application acts as B2BUA to relay initial request

```
        +--------------------------------------------+
  req1  |                                            |  req2
------->| req2 = factory.createRequest(req1, ...); |--------->
        | req2.send();                               |
        |                                            |
        +--------------------------------------------+

        req2 is a continuation of req1
```

In the cases encountered so far, the application's intentions are clear from context. Such intentions are referred to as *routing directives*. In the first case, the application implicitly signaled a NEW directive, and in the second and third cases the application implicitly signaled a CONTINUE directive.

Unlike the cases described above, there are times where an explicit directive is required.

Listing 18-4 Application acts as B2BUA to relay initial request and signals the CONTINUE directive explicitly

```
        +--------------------------------------------+
        |                                            |
  req1  | req2 = factory.createRequest(appSession,  |   req2
------->|   "INVITE", newFrom, req1.getTo());        |--------->
        | // copy headers and content from req1      |
        | req2.setRoutingDirective(CONTINUE, req1);|
        | req2.send();                               |
        +--------------------------------------------+

        req is a CONTINUE
```

In another case, an application wishes to send a new request based on a received request but intends the request to be regarded as a new request, thereby initiating a new selection process. The application could create the new request separately and copy all of the headers and content, but it is far easier to create the request based on an existing request and then indicate a routing directive explicitly by using a method in the `SipServletRequest` class introduced in version 1.1: `setRoutingDirective(SipApplicationRoutingDirective dir, SipServletRequest req)`

Listing 18-5 Application acts as B2BUA to send NEW request

```
          +-------------------------------------------+
  req1    |                                           |  req2
- - - ->| req2 = factory.createRequest(req1, ...); |--------->
          | req2.setRoutingDirective(NEW, null);      |
          | req2.send();                              |
          |                                           |
          +-------------------------------------------+

          req2 is a NEW
```

In the above case, a new application selection process begins for req2.

Finally, a third directive, REVERSE, is necessary but is used less commonly. REVERSE is used when an application reverses the direction of the call. There are two cases. In the first case, an application invoked to service the caller for an initial request now wishes to place a call back to the caller. A practical example is a Call Waiting (CW) application. When a subscriber to CW places a call, CW is invoked to serve the caller. When this call is still up, the subscriber receives an incoming call. CW alerts the subscriber, who then activates CW to put the first callee on hold, and switch to the new caller. If the subscriber finishes the conversation with the new caller and hangs up the phone, forgetting that there is another call on hold, the CW feature places a call to the subscriber. In this call, the subscriber is now the callee.

In the second case, an application that is invoked to service the callee in a request, now wishes to place a call on behalf of that same subscriber as the caller. A practical example is a three-way calling (3WC) application. When the subscriber receives a call, the 3WC application is invoked to serve the subscriber as the callee. In the middle of the call, the subscriber activates the 3WC application to call a third person, with the subscriber as the caller.

In both cases, the direction of the call is reversed, and the applications must specify the REVERSE directive so the Application Router is able to select the next application correctly.

Listing 18-6 Application reversing the call direction by placing a call back to the caller

```
                +-------------------------------------------+
                |                                           |
          req1  | req2 = factory.createRequest(appSession,  |
        ------->|   "INVITE", newFrom, req1.getFrom(),...); |
Caller          | // copy headers and content from req1     |
          req2  | req2.setRoutingDirective(REVERSE, req1);  |
        <-------| req2.send();                              |
                +-------------------------------------------+

        req2 is a REVERSE
```

It is important to note that in the cases where an application is sending a request based on a previously received request, the transaction of the received request may have completed, or indeed the SIP dialog may have completed and terminated. Indeed, in 3WC the transaction is already completed, and in busy retry the SIP dialog has already terminated. The application may store the request for later use. Note also that the second argument to `setRoutingDirective` must be an initial request received by the application(i.e., `SipServletRequest.isInitial()` must be true). Table18-1 summarizes the routing directive under different scenarios.

Table 18-1 Routing Directives

Application Action	Directive
Req = factory.createRequest(appSession, "INVITE", from, to);	NEW
request.getProxy().proxyTo()	CONTINUE
req2 = factory.createRequest(req1, ...); req2.send();	CONTINUE
req2.setRoutingDirective(directive, req1);	Explicit directive

Note: If recursion is enabledand the container automatically proxies to contact addresses received in redirect (3xx) responses, these requests are treated as if the application explicitly called `proxyTo()` to proxy to these contact addresses. Therefore, they also have the CONTINUE directive.

18.3 Application Router Behavior

When the container receives an initial request from an external entity, or when an application acts as a UAC and sends an initial request with a NEW routing directive, the application selection process is started fresh. In this case, the Application Router is called with the following information:

- The `SipServletRequest`
- The routing directive which is NEW

Based on the supplied information, its configuration of which subscriber subscribes to which set of applications, and any other information that it may wish to use (e.g., time of day, network condition, or external subscriber profile database), Application Router returns the following information:

- the name of the selected application
- the subscriber identity that the selected application is to serve
- the routing region that the application serves in
- an array of zero or more routes to push (the routes array)
- a route modifier that tells the container how to interpret the route
- an optional stateInfo serializable object

The Application Router can return routes to the container via its `SipApplicationRouterInfo.getRoutes()` method. The routes can be external or internal. External routes are used by the Application Router to instruct the container to send the request externally. Internal route is returned when the Application Router wishes to modify the popped route header as seen by the application code (through the `SipServletRequest.getPoppedRoute()` API call). When the request is received by the container, the request may have had a Route header belonging to the container, which the container removes and makes available through the provisions of 5.6.3 Popped Route Header. The route modifier returned by the Application Router tells the container how to use the routes returned by it and also how the popped route needs to be presented. The route modifier can be one of the following enum values: ROUTE, ROUTE_BACK, or NO_ROUTE.

- ROUTE modifier indicates that `SipApplicationRouterInfo.getRoutes()` returns valid routes. The container decides if they are external or an internal route was returned. All of the routes returned MUST be of the same type, so the container can make the determination by examining the first route only.

- ROUTE_FINAL indicates that `SipApplicationRouterInfo.getRoutes()` returns valid routes. The returned route can contain internal routes, external routes, or both. The container pushes all routes returned by the Application Router.
- ROUTE_BACK directs the container to push its own route before pushing the routes obtained from `SipApplicationRouterInfo.getRoutes()`.
- NO_ROUTE indicates that Application Router is not returning any routes and the `SipApplicationRouterInfo.getRoutes()` value, if any, should be disregarded.

The behavior of the container with respect to the route modifiers is explained in section 18.4.1 Procedure for Routing an Initial Request.

The stateInfo serializable object is useful for the Application Router to store state information from one invocation to the next. An Application Router implementation may choose to put any information in the stateInfo object, and this object is opaque to the container and not accessible to the applications. Typically, an Application Router implementation may store information such as subscriber identity, the name of the application last invoked, and a precomputed list of applications that are to be invoked next. Application Router MUST NOT change the stateinfo after it is returned to the container, and each result returned to the container must contain a different stateinfo object.

If the selected application subsequently proxies or sends a new initial request based on the first one with a CONTINUE or REVERSE routing directive, the Application Router is called again. This time, in addition to the `SipServletRequest` and the routing directive, it is also supplied with the stateInfo object that it previously returned. In this way, the Application Router delegates the maintenance of the application selection state to the container, and thus it can be stateless with respect to each initial request it processes.

If the Application Router determines that no application is selected to service a request, it returns null as the name.

18.3.1 Order of Routing Regions

Because proximity of an application to its subscriber confers priority, it is beneficial for the management of feature interaction that originating applications are closest to the caller, and that terminating applications are closest to the callee. This can be satisfied if the following rules are followed:

- The originating region applications should be invoked first followed by terminating region applications.
- The applications that service a subscriber are contiguous(i.e., no insertion of applications that service other subscribers in between).

On the other hand, it is entirely possible that the Application Router progresses directly to the terminating region if the caller is not a subscriber, or the caller does not subscribe to any applications. It is also possible that this application server does not serve any originating subscribers or has determined through some means that the originating applications have already been invoked and it should only look for terminating applications.

18.3.2 Inter-Container Application Routing

This specification supports applications distributed across multiple containers. The Application Router may return external routes in its `SipApplicationRouterInfo.getRoutes()` method that point to other application servers that it wishes the request to be routed to. The container MUST then push the routes onto the Route header stack of the request and send the request externally. If this request arrives at a container compliant with this specification, it invokes the Application Router residing in that container so that the application selection process may continue. The first Application Router may pass any state information to the second Application Router by embedding it in the Route header. This is in accordance with the cascaded services model [SERL] as the applications can reside on different hosts and still participate in the application composition process.

Note: Some architectures require that the originating and terminating applications be hosted on different servers. The deployer can easily accomplish this by configuring the Application Routers such that one server hosts only originating applications and the other only terminating applications. Either the subscriber data can be partitioned such that the first server only serves originating users and the second serves terminating users.Alternatively, the Application Routers can collaborate by passing some state information in the Route headers, indicating for example that the first server has already completed the invocation of originating services.

18.4 Container Behavior

The container is responsible for instantiating and initializing the Application Router and providing to it the initial list of deployed applications. Further, when new applications are deployed or when applications are undeployed, the container must also inform the Application Router.

The container is responsible for receiving an initial request from an external entity or from an application, invoking the Application Router to obtain the name of the application to service the initial request, and dispatching the request to the main servlet within this application as described in section 19 Mapping Requests To Servlets. The container is also responsible for maintaining application selection state including:

- the routing directive associated with this request
- routing region (originating, terminating, or neutral)
- acting on the route returned from the Application Router in conjunction with the route modifier
- arbitrary, opaque state information returned from the Application Router

18.4.1 Procedure for Routing an Initial Request

When the container receives a new initial request, it first creates and initializes the various pieces of application selection state as follows:

Directive:

- If a request is received from an external SIP entity, directive is set to NEW.
- If a request is received from an application, directive is set either implicitly or explicitly by the application.

Application router stateInfo:

- If a request is received from an application and the directive is CONTINUE or REVERSE, stateInfo is set to that of the original request that this request is associated with.
- Otherwise, stateInfo is not set initially.

Subscriber URI and Routing Region:

These are not set initially.

The following procedure is then executed:

1. Call the `SipApplicationRouter.getNextApplication()` method of the Application Router object. The Application Router returns a `SipApplicationRouterInfo` object, named 'result' for this discussion.
2. Check `result.getRouteModifier()`

- If `result.getRouteModifier()` is ROUTE, get the routes by using result.getRoutes().
  - If the first returned route is external (does not belong to this container), push all of the routes on the request's Route header stack and send the request externally. Note that the first returned route becomes the top route header of the request.
  - If the first returned route is internal, the container MUST make it available to the applications via the `SipServletRequest.getPoppedRoute()` method and ignore the remaining ones, if any. This allows the AR to modify the popped route before passing it to the application.
- If `result.getRouteModifier()` is ROUTE_FINAL, push all of the routes, regardless of whether the route is internal or external, on the request's route header stack and send the request. This mode is introduced in SIP Servlet 2.0 and should be preferred over ROUTE wherever possible.
- If `result.getRouteModifier()` is ROUTE_BACK, push a route back to the container followed by all of the routes obtained from result.getRoutes() and send the request externally. When the request eventually returns back, container MUST set the routing directive to CONTINUE and also retrieve the AR state (routingRegion, stateInfo) and pass it in the call to `getNextApplication()`to continue processing the application chain. To retrieve the AR state, the container route in the request originally sent externally SHOULD include AR state (routingRegion and stateinfo) encoded as a route parameter.
- If `result.getRouteModifier()` is NO_ROUTE, disregard `result.getRoutes()` and proceed.

3. Check `result.getNextApplicationName()`
   - If `result.getNextApplicationName()` is not null:
     - Set the application selection state on the `SipSession`: stateInfo to result.getStateInfo(), region to result.getRegion(), and URI to result.getSubscriberURI().
     - Follow the procedures of Chapter 16 to select a servlet from the application.
   - If `result.getNextApplicationName()` is null:
     - If the Request-URI is not addressed to this container, or if there are one or more Route headers, send the request out according to the standard SIP mechanism.
     - If the Request-URI is addressed to this container and there is no Route header, the container should not send the request as it will cause a loop. Instead, the container

must reject the request with a 404 Not Found final response with a no Retry-After header.

Note that the effect of sending the request externally (e.g, as a result of ROUTE_FINAL or ROUTE_BACK routing directive) can cause the message to come back to the container and present again to AR. The container may choose to optimize this case by calling `getNextApplication()` directly instead of sending the message. The container SHOULD take necessary steps to detect loops as described in section 18.9 Loop Detection.

Note: As a guideline, it is strongly recommended for applications to not rely on Via or Record-Route headers for their application logic. As within the container, the container implementations may choose to optimize the handling of the system headers by either aggregating them while going out or just replacing them with one container level header and maintaining the state internally. The applications SHOULD instead use the `SipServletMessage` methods `getLocalXXX`, `getRemoteXXX` if they are interested in the upstream entity.

If `SipApplicationRouter.getNextApplication()` throws an exception, the container should send a 500 Server Internal Error final response to the initial request.

18.4.2 Application Router Packaging and Deployment

The container is responsible for loading and instantiating Application Router implementations. To be portable across containers, the Application Router implementation MUST be packaged in accordance with the rules specified by the Java SE Service Provider framework. Specifically, the JAR file containing the Application Router implementation must include the `META-INF/services/javax.servlet.sip.ar.spi.SipApplicationRouterProvider` file. The contents of the file indicate the name of the concrete public subclass of the `javax.servlet.sip.ar.spi.SipApplicationRouterProvider` class. The concrete subclass must have a no-arg public constructor.

As specified by the Service Provider framework, the providers may be installed by:

1. Including the provider JAR in the system classpath
2. Including the provider JAR in the extension class path
3. Container-specific means

If the container uses classpath-based deployment, the first Application Router JAR file found in the classpath is installed. To avoid ambiguity when multiple Application Router implementations are present in the classpath, this specification also defines a system property that instructs the container to load a given provider. The `javax.servlet.sip.ar.spi.SipApplicationRouterProvider` system property can override loading behavior and force a specific provider implementation to be used. For portability reasons, containers that provide their own deployment mechanism for the Application Router SHOULD observe the system property, if specified by the deployer.

18.5 Responses, Subsequent Requests and Application Path

If an initial request results in a SIP dialog being established and at least one application is on the signaling path, be it a UA or a record-routing proxy, the container sets up state so that other requests in the same dialog can be routed to the set of applications within the container that are on the signaling path. Requests that are routed internally in a container based on such state are called *subsequent requests* in this specification. Subsequent requests are not dispatched to applications based on the application selection process.

Correct routing of subsequent requests and responses can be achieved in several ways, and it is up to implementors to choose one. It is recommended that when routing messages internally between applications, the container SHOULD use the SIP mechanism of adding the Via header and Record-Route (if it is record-routing proxy or a B2BUA) to allow for a robust stateless composition chain. However, it is entirely possible that the implementations MAY choose to hide the internal topology by maintaining the application path state outside of the SIP message. The implementation MUST, however, ensure that the parameters added to the Record-Route headers by individual applications are made available through the Route header on the subsequent request. The goal is to provide SIP consistency both at the application interface level and external SIP entity level.

Subsequent requests originated from the caller follow the path, or more precisely a subset of the path, of the corresponding initial request. Subsequent requests originated from the callee as opposed to the caller follow the reverse path. Responses always follow the reverse of the path taken by the corresponding request. This is true for responses to both initial and subsequent requests. The application path is a logical concept and as such may or may not be explicitly represented within containers.

For illustration, suppose three applications, A, B, and C, are invoked one after another to process an initial INVITE request. All three applications proxy but only A and C record-route, and so the application path consists of A and C along with associated contexts (sessions). A subsequent BYE request received from the callee is then routed based on this application path, and is passed first to C and then to A.

The distinction between initial and subsequent requests also applies to dispatching of locally initiated requests. If, for example, application A initiates an INVITE, and this is passed to application B which proxies it with record-routing enabled towards a destination outside the application server, then a subsequent BYE from A for the same dialog will be passed to B and then further downstream. Proxying of subsequent requests is discussed in section 12.2.9 Handling Subsequent Requests.

It is worth noting that `SipSession` can belong to at most one application path. This is because initial requests are processed in the context of new `SipSession`s and because there is a one-to-one correspondence between application paths and SIP dialogs. If the initial request results in more than one dialog being set up, the container creates derived `SipSession`s for the second and subsequent paths being created. See section 8.2.3.2 Derived SipSessions.

When `SipSession` terminates, either because the `SipApplicationSession` it belongs to times out or is explicitly invalidated or because the `SipSession` itself is explicitly invalidated, it is removed from the application path it was on, if any. If application paths are represented explicitly within containers, they are removed when the path becomes empty.

18.6 Transport Information

The `SipServletMessage` class provides methods for an application to obtain information about the transport used to receive a message. The container MUST return the following values irrespective of whether the request is initial or subsequent.

- `getRemoteAddr()`MUST return the address of the remote SIP interface if the message was received from an external entity.However, if the message was internally routed (from one application to the next on the same container), it MUST return the address of the container's SIP interface.
- `getRemotePort()` MUST return the port of the remote SIP interface if the message was received from an external entity.However, for internally routed messages, the container is free to choose any port value consistent with one that would be chosen were the container to have actually sent the message out and back to itself through the local TCP/IP stack.
- `getLocalAddr()` the address of the SIP listening interface on which this message was originally received from the external SIP entity throughout the chain.
- `getLocalPort()` the port number of the SIP listening interface on which this message was originally received from the external SIP entity throughout the chain.

- `getTransport()` the actual transport on which this message was received from the external entity and for the internally routed request - TCP or TLS to indicate that this is a reliable transport.
- `getInitialRemoteAddr()` the IP address of the upstream/downstream hop from which this message was initially received by the container. This method returns the same value regardless of which application invokes it in the same application composition chain.
- `getInitialRemotePort()` the port number of the upstream/downstream hop from which this message was initially received by the container. This method returns the same value regardless of which application invokes it in the same application composition chain.
- `getInitialTransport()` the name of the protocol with which this message was initially received by the container. This method returns the same value regardless of which application invokes it in the same application composition chain.
- for the incoming TLS connection, the X.509 certificate MUST be made available only to the first application that receives the message from outside of the container through the javax.servlet.request.X509Certificate for requests and javax.servlet.response.X509Certificate for responses. However, if the container is able to authenticate the remote user based on the credentials in the certificate, that authentication information MUST be made available to all applications as described in section 17.6 Server Tracking of Authentication Information

18.7 Popping of Top Route Header

18.7.1 Container Behavior

With application composition, the requirement of popping the top route header belonging to the container [as described in 6.7.3 Popped Route Header] is to be done on receipt of the request, not just on the network interface from the external SIP entity but also within the container when the request is received from one of the applications. This happens when one of the applications pushes a route header that belongs to the container. This popping of a container's route can happen irrespective of the request being initial or subsequent. Besides this, the Application Router can also modify the route header originally popped on initial request processing as per section 18.3 Application Router Behavior.

18.7.2 Top Route Processing Examples

This section contains three examples to help illustrate the behavior of route popping, Application Router, `SipServletRequest.getPoppedRoute()`, as well as the `SipServletRequest.getInitialPoppedRoute()` API methods.

18.7.2.1 Request with two route headers arriving at the container

This example illustrates an application chain with two applications (App1, App2), where neither application pushes routes onto the incoming request.

```
    (Alice) - AR - App1 - AR - App2 - AR - (Bob)
```

1. Request R from Alice arrives at the container.
2. The request contains two route headers, A_i and B_e. A_i represents a route pointing to the container, B_e means it is an external route. The AR receives the request R.
3. Inside of the AR, `R.getPoppedRoute()` returns A_i as A_i is popped by the container prior to passing the request onto the AR. Also, `R.getInitialPoppedRoute()` returns A_i. R still contains theB_e route header. The AR returns App1 and the NO_ROUTE modifier to the container.
4. App1 executes.
5. Inside of App1, `R.getPoppedRoute()` returns A_i and R's top route is B_e. `R.getInitialPoppedRoute()` returns A_i. App1 proxies the request to Bob without calling pushRoute().
6. The AR intercepts the request again.
7. Inside of the AR, `R.getPoppedRoute()` returns null and R still contains B_e as its top route. `R.getInitialPoppedRoute()` still returns A_i. AR returns App2 andthe NO_ROUTE modifier to the container.
8. App2 executes.
9. In App2, `R.getPoppedRoute()` returns null, and R's top route is B_e. `R.getInitialPoppedRoute()` returns A_i. App2 proxies R to R's Request-URI (Bob). The AR executes.

   AR returns null application to the container and NO_ROUTE. This causes the container to send the request to B_e, as per section 18.4.1 Procedure for Routing an Initial Request, point 3.

18.7.2.2 Application pushes route pointing back at the container

This example illustrates container behavior when the application pushes a route onto the incoming request pointing back to the container.

```
(Alice) - AR - App1 - AR - (Bob)
```

1. Request R from Alice arrives at the container.
2. The request contains two route headers, A_i and B_e. A_i represents a route pointing to the container, B_e means it is an external route. The AR receives the request R.
3. Inside of the AR, `R.getPoppedRoute()` returns A_i as A_i is popped by the container prior to passing the request onto the AR. `R.getInitialPoppedRoute()` returns A_i. R still contains B_e route header. The AR returns App1 and the NO_ROUTE modifier to the container.
4. App1 executes.
5. Inside of App1, `R.getPoppedRoute()` returns A_i and R's top route is B_e. `R.getInitialPoppedRoute()` returns A_i. App1 calls `R.pushRoute(C_i)`, where C_i represents a Route header pointing back at the container. Next, App1 proxies the request to Bob. The AR intercepts the request again.

   Inside of the AR, `R.getPoppedRoute()` returns C_i, and R's top route header is B_e. `R.getInitialPoppedRoute()` returns A_i. AR decides to return null application to the container, and NO_ROUTE. This causes the container to send the request to B_e, as per section 18.4.1 Procedure for Routing an Initial Request, point 3.

18.7.2.3 Application pushes external route

This example illustrates container behavior when an application pushes an external route.

```
(Alice) - AR - App1 - AR - (Bob)
```

1. Request R from Alice arrives at the container.
2. The request contains two Route headers, A_i and B_e. A_i represents a Route pointing to the container, and B_e means it is an external Route. The AR receives the request R.
3. Inside of the AR, `R.getPoppedRoute()` returns A_i as A_i is popped by the container prior to passing the request onto the AR. `R.getInitialPoppedRoute()` returns A_i. R still

contains the B_e Route header. The AR returns App1 and the NO_ROUTE modifier to the container.

4. App1 executes.

5. Inside of App1, `R.getPoppedRoute()` returns A_i and R's top route is B_e. `R.getInitialPoppedRoute()` returns A_i. App1 calls `R.pushRoute(D_e)`, where D_e represents a Route header pointing to an external address. Next, App1 proxies the request to Bob. The AR intercepts the request again.

   Inside of the AR, `R.getPoppedRoute()` returns null, and R's top route header is D_e. `R.getInitialPoppedRoute()` returns A_i. AR decides to return null application to the container, and NO_ROUTE. This causes the container to send the request to D_e, as per section 18.4.1 Procedure for Routing an Initial Request, point 3.

18.8 Examples

The first example uses a simple two application scenario to illustrate the interactions between the container, Application Router, and applications along with the concept of application path and routing of subsequent requests and responses. It also serves to illustrate the concept of message context discussed in section 8.6.1 Message Context. The examples that follow the first examine more complex application composition scenarios.

18.8.1 Example with Two Applications

Consider the Location Service (LS) application in section 1.6.1 A Location Service, and a new Speed-Dial (SD) application. This latter application allows a calling party to specify a short speed-dial number (e.g. "1"), which it translates into a complete address based on prior configuration. LS allows the called party to control where calls are received. For the purpose of illustration, assume that both SD and LS applications record-route.

The following diagram shows the routing of an initial INVITE request. Assuming that the caller and callee of the INVITE request are both subscribers served by this server, SD is needed to serve the caller and LS is needed to serve the callee. The following step-by-step walk through of the call flow indicates message context as a triplet (app, as, ss) where app is the application name, as specifies an application session, and ss specifies a `SipSession`.

Example of Application Composition:

```
-------> SD ------> LS -------->
```

1. The container receives an INVITE request. The INVITE does not belong to an existing SIP dialog and so the container calls the Application Router with the request to obtain the name of the application to invoke.
2. The Application Router determines SD is the first application to serve the caller. The Application Router returns the name "SD", the caller's identity, and the originating region to the container together with some state information.
3. The container invokes the SD application in the context of (SD, as1, ss1).
4. SD, based on the caller's identity, performs database lookup to obtain the caller's speed-dial settings and proxies to the full address that corresponds to the speed-dial number.
5. The container receives the proxied request and again calls the Application Router with the request and the state information.
6. Based on the state information, the Application Router determines that SD was already invoked and that there are no other applications for the caller. Assuming that there are no applications in the neutral region, the Application Router proceeds to the terminating region and determines that LS is needed to service the callee. It returns the name "LS", the callee's identity, and the terminating region to the container, together with some state information.
7. The container invokes the LS application in the context of (LS, as2, ss2).
8. LS, based on the callee's identity, performs database lookup to obtain the callee's location settings and proxies the request to destination d1.
9. The container receives the proxied request and calls the Application Router with the request and state information.
10. The Application Router determines that no applications are required for d1 and returns null to the container, indicating there is no further application to be invoked.
11. The container proxies the request towards d1, that is, outside the application server.
12. The container receives a 200 (OK) response for the INVITE. The 200 response is passed upstream along the reverse path of the request(i.e., the container passes it first to the LS application in context (LS, as2, ss2) and then to the SD application in context (SD, as1, ss1), and then sends it to the caller).
13. Since the 200 response establishes a dialog, an application path is created. This means that the container routes subsequent requests in this dialog to applications along the signaling path. In this case, the application path consists of (SD, as1, as2) and (LS, as2, ss2).

14. An ACK for the 200 is received. This is recognized as being a subsequent request and is associated with the previously established application path. (Incoming ACKs for non-2xx final responses are needed for protocol reasons only and are simply dropped.)
15. The container passes the ACK to the SD application in the context (SD, as1, ss1) and when the up-call to SD returns, it passes the ACK to the LS application in the context (LS, as2, ss2). When the up-call to LS returns, the container proxies the ACK outside the application server according to standard SIP routing rules.
16. Assuming that the callee hangs up first, the BYE is passed along the application path in the reverse direction(i.e., it is passed to LS first and then to SD).

Note that, logically, each application has its own set of SIP client and server transaction objects operating in accordance with the transaction state machines specified in the SIP specification [RFC 3261, chapter 17]. Likewise, logically, each proxy application executes its own instance of the proxy logic. For example, it has its own response context [RFC 3261, section 16]. This specification then "augments" the RFC 3261 defined state machines with additional rules. For example, the 100 responses, ACKs for non-2xx responses, and responses for CANCELs are not delivered to the applications.

18.8.2 Simple call with no modifications of requests

As an example, assume that Alice, Bob, and Carol all subscribe to the following applications:

- Originating region applications:
  - Originating call screening (OCS)
- Terminating region applications:
  - Call Forwarding (CF)
  - Incoming Call Logging (ICL)
- Applications in both originating and terminating regions:
  - Call Waiting (CW)
  - 3-Way Calling (3WC)

Assume that all of these applications proxy or relay the initial requests they receive without modification. Also assume that the Application Router selects applications based on a simple ordering: the originating applications in the order CW, 3WC, OCS, and the terminating applications in the order ICL, CF, 3WC, CW.

When the container receives a call from Alice to Bob, it calls the Application Router which selects CW to service Alice, because CW is the first application in Alice's originating region. When CW relays the INVITE as a B2BUA, the container again calls the Application Router. Because the application selection process is carried forward from the INVITE that CW received to the INVITE that CW is now sending, the Application Router determines that CW was already invoked based on the current state of the selection process and that the next application is therefore 3WC. The same process is repeated when 3WC relays the INVITE request and the OCS application is invoked next. When OCS relays the INVITE request, the Application Router determines that there are no more originating applications for Alice. The application selection process then proceeds to the callee’s (i.e., Bob's) terminating region, where ICL, CF, 3WC, and CW are then invoked in turn. During this call, we assume that CF, although present, is not active. That is, it does not forward the call to another URI. When Bob's CW relays the INVITE, there are no more terminating applications for Bob and thus the container sends the request to Bob's UA.

When complete, the application selection process yields the following application path:

```
(Alice) - CW - 3WC - OCS ------ ICL - CF - 3WC - CW - (Bob)

     Alice's originating     |     Bob's terminating
```

18.8.3 Modification of Headers

An application may change the originating address and/or the terminating address by modifying specific headers in the SIP request. As mentioned above, applications are usually subscribed to and are invoked to service an address either in the originating region or the terminating region. Therefore, if an application modifies one or both addresses, in general the application selection process is affected.

As an example, assume Bob subscribes to Call Forwarding and configures it to forward all calls to Carol. When Alice calls Bob, the Call Forwarding (CF) application is invoked in the terminating region to service Bob. CF proxies the INVITE request to Carol's Request-URI. At this point, the Application Router typically stops invoking other terminating region applications for Bob, and instead begins invoking Carol's terminating region applications.

Following on from the example above, the invocation sequence is:

```
(Alice) - CW - 3WC - OCS ------ ICL - CF ------ ICL - CF - 3WC - CW -
(Carol)

        Alice's Originating   Bob's Terminating    Carol's Terminating
```

The following is an example where the caller's address information is modified, resulting in a different set of originating applications being selected. Assume that Alice works at home as a customer care agent for her employer. She now subscribes to an additional originating application, Agent Identification (AI). If Alice is not working, this application does nothing. However, after Alice logs on to the employer's system and places a call, this application modifies the caller URI (e.g. the From header) in the INVITE request to be sip:customer-care@employer.com and relays the request. The Application Router then selects originating applications subscribed to by the new caller URI, such as a Supervisor Monitor (SM) application so that the supervisor may listen in and coach Alice.

```
(Alice) - AI ---------------- SM - ...

        Alice's Originating   customer-care's originating
```

As another example, consider the case when Alice calls Carol and they are engaged in a conversation. The application path looks like:

```
(Alice) - CW - 3WC - OCS ------ ICL - CF - 3WC - CW - (Carol)

        Alice's Originating       Carol's Terminating
```

When Alice activates 3WC to call Bob, 3WC continues the INVITE request it received previously, but modifies the callee to Bob. Because 3WC is in the originating region and the caller address is not modified, the application selection is not affected at this point and the remaining originating application, OCS, is selected. Then, the terminating applications of Bob are selected. The resulting application path would then look like:

```
(Alice) - CW - 3WC - OCS ------ ICL - CF - 3WC - CW - (Carol)
                 \
                  \ - OCS ------- ICL- CF - 3WC - CW - (Bob)

        Alice's Originating       Bob's Terminating
```

Note that this usually happens some time after the initial Alice-Carol call was established. 3WC maintains a reference to the INVITE request it received initially so that it may use it to initiate a call to another party if it is ever activated. The application selection state information associated with that INVITE request has the same lifetime as the request.

18.8.4 Reversing the Direction of the Call

This example illustrates the use of the REVERSE directive. Consider the case when Carol calls Alice and they are engaged in a conversation. The application path is:

```
(Carol) - CW - 3WC - OCS ------ ICL - CF - 3WC - CW - (Alice)
```

In this state, Alice's 3WC application was invoked in a call where Alice is the callee. However, if Alice activates 3WC to call Bob, 3WC sends a new INVITE request where Alice is the caller and Bob is the callee. This distinction affects the application selection process. Normally, if 3WC is simply continuing the request in the same direction towards Alice as the callee, the next application invoked to serve Alice would be CW. However, in this case, the correct next application is OCS, followed by Bob's terminating region applications. The resulting application paths should look like:

```
(Carol) - CW - 3WC - OCS ------ ICL - CF - 3WC - CW - (Alice)
                                             \
                                              \ - OCS ------ ICL -
 - CF - 3WC - CW - (Bob)
```

In other words, 3WC was previously operating in Alice's terminating region, but for the new call, 3WC is operating in Alice's originating region. Therefore, the Application Router selects Alice's originating applications, and in this case the next such application is OCS. In fact, one would observe that the application path from Alice's 3WC to Bob is identical to the corresponding portion.

18.8.5 Initiating a New Request

Consider a network-based alarm application that calls its subscriber at a preset time. This application acts as a UAC and sends a new INVITE request towards the subscriber(i.e., with an implicit NEW directive).

This request is treated in the same way as a request received from an external SIP entity. The application selection process is started afresh.

Using the alarm application (AL) as an example, it creates and sends an INVITE request to Alice with the From header set to alarm-service@example.com. The Application Router begins from the originating region of alarm-service@example.com, but in this case there are no originating applications to invoke. Alice's terminating applications are then selected and invoked as usual, resulting in the application path shown below:

```
    AL ------ ICL - CF - 3WC - CW - (Alice)
```

18.9 Loop Detection

There is a possibility that loops may occur in invoked applications, such as A proxies to B which proxies back to A. It is important that such loops be detected and handled. This specification does not mandate a particular mechanism. One strategy that is consistent with the cascaded services model is to mimic the existing SIP "inter-host" mechanisms for loop detection in the "intra-host" case of SIP Servlet containers. In particular, a simple and effective mechanism is to decrement the value of the Max-Forwards header whenever a request is proxied internally, or whenever a request is forwarded by a servlet acting as a B2BUA. Section 12.2.10 Max-Forwards Check discusses the issue of when to generate 483 (Too Many Hops) error responses on the basis of the Max-Forwards header.

The SIP container also checks the Max-Breadth header to verify whether the request can be proxied in parallel or forwarded to multiple destinations by a servlet acting as a B2BUA in parallel. As specified in section 12.2.11 Max-Breadth Check,the container generates an `InsufficentBreadthException` when the Max-Breadth check fails.

The SIP container may also implement a loop detection algorithm and updates to RFC 3261 as defined in section 4 of RFC 5393.

18.10 Session Targeting

There are three session targeting mechanisms supported by this specification. All of them allow the container to associate seemingly unconnected initial requests with the same `SipApplicationSession`. The encode URI mechanism, defined in v1.0 of this specification, has been deprecated in v1.1 of this specification as it is problematic and does not mesh well with the v1.1 application composition mechanism. The Session Key based targeting mechanism introduced in v1.1 of this specification is now the preferred mechanism to associate a request with a particular `SipApplicationSession`. It is also more powerful than the deprecated encode URI mechanism. Finally, the optional Join [RFC 3911] and Replaces [RFC 3891] mechanisms also allow an initial request (INVITE) to be targeted to a specific `SipSession` (SIP dialog) thereby targeting its parent `SipApplicationSession`. For this discussion, any request that invokes one of these three session targeting mechanisms is termed a *targeted request*. Unlike normal initial requests, targeted requests target a particular `SipApplicationSession` object and hence, implicitly, a particular application. Further, for two of these mechanisms, the encode URI and the Join/Replaces support, the session targeting includes the targeting to an application in addition to the `SipApplicationSession` object. That is, a targeted request to an encoded URI or a request with either a Join or a Replaces header is targeted to a particular application before the Application Router is invoked to perform application selection. Thus, in addition to describing the mechanisms themselves, the sections below must address how targeted request mechanisms are harmonized with Application Router based application composition.

18.10.1 Session Targeting and Application Selection

Targeted requests employing the encode URI mechanism or the Join/Replaces mechanism are considered initial requests and therefore the container must invoke the Application Router to service such targeted requests just as with any other initial request. However, to simplify the Application Router implementation logic in recognizing that a request is targeted, a `SipTargetedRequestInfo` object is provided by the container to the Application Router. If present in a call to `getNextApplication()`, this object indicates to the Application Router that the request is targeted and further provides two pieces of information:

1. Targeted Request Type. This enumerated value indicates that the request contains a Join header, contains a Replaces header, or is targeted to an encoded URI.
2. Application Name. This string identifies the name of the application that owns the SipApplicationSession to which this request is targeted.

Note: There is no targeted request type corresponding to the Session Key based targeting mechanism defined in the v1.1 specification. That is because this mechanism is different from the other two: the Session Key based targeting mechanism operates only after the container has called the Application Router's `getNextApplication()` method to determine the next application to be invoked. For all of the three session targeting mechanisms, the servlet to be invoked within the application MUST be the designated "main" servlet as discussed in section 19.2 Servlet Selection. (Unless of course the encode URI targeted application is a v1.0 based application). The sections below describe each of these three session targeting mechanisms in more detail.

18.10.2 Session Key Based Targeting Mechanism

In general, invoking an application creates a `SipApplicationSession` object belonging to that application. Thus, processing an initial request normally creates a new `SipApplicationSession` instance.

However, sometimes it is required to route all requests for a subscriber, application, and region combination (or other factors) to a single `SipApplicationSession` instance.

For example, consider the CW (Call Waiting) application that is invoked on behalf of the subscriber Carol in the originating region, whose URI appeared in the From header. The application chain is subsequently formed with other applications and the dialog is established.An initial request is received and during the application selection process, the Application Router selects the CW application. Further, if the subscriber is Carol and the region is originating, the container must associate this request with an existing `SipApplicationSession`. The CW application should have a way to indicate its desire for such an association.

Applications indicate their desire to associate the request with an existing `SipApplicationSsession`s through a key generating mechanism described below. Applications can use the method annotation `@SipApplicationKey` within one and only one of its classes, generally the `Servlet` class. This annotated static public method takes the (read-only) `SipServletRequest` as its argument and returns a String. The returned String is then used as a "key" to lookup `SipApplicationSession`. Note that the applications themselves may generate this key based on the contents of the request, the routing region, the subscriber, or any other relevant information. The container uses the application generated key as an index for the application sessions. As the `application-session-id` must have the property of being unique across applications in a JVM, the containers may prefix app-names or other such application-specific identifiers to keys before assigning them as `application-session-id`s. This ID then informs the container which `SipApplicationSession` instance this initial request should be routed to.

The static method annotated by `@SipApplicationKey` is prohibited from modifying the request in any way, and any attempts to do so SHOULD result in an `IllegalStateException` thrown by `SipServletRequest`.

While processing the initial request after selecting the application, the container MUST look for this annotated static method within the application. If found, the container MUST call the method to get the key and generate an `application-session-id` by appending a unique identifier. If the resultant `application-session-id` identifies an existing `SipApplicationSession` within the container JVM, the container MUST associate this request with that `SipApplicationSession` rather than create a new one.

If the generated `application-session-id` does not identify an existing `SipApplicationSession` instance, a new instance MUST be created.

The absence of the `@SipApplicationKey` method annotation indicates that the application does not want to use the Session Key based targeting mechanism.Thus, the container MUST (by default) create a new `SipApplicationSession` instance for every initial request.

It is strongly recommended that the same `application-session-id` be returned by the `getId()` method of `SipApplicationSession`. If the annotation exists, the container should use the return value of the `@SipApplicationKey` annotated method in the return value of `getId()` after adding a unique prefix, as required.

As an example, consider a chat room application that wants all requests with request URI "sip:mychatroom1@example.com" to be handled with the same `SipApplicationSession`. The application defines a method with annotation `@SipApplicationKey` that takes `SipServletRequest` as the argument.

1. An initial request comes into the container, which consults the Application Router.
2. The Application Router selects the chat application and informs the container.
3. The container calls the method annotated with `@SipApplicationKey` and gets an application session key. (Since the application wants to send all requests with a certain Request-URI to the same `SipApplicationSession`, it returns a key, based on the Request-URI, like a hash or perhaps just the Request-URI itself).
4. The container prefixes the key with the application name for uniqueness and then uses the resultant `application-session-id` to check if a `SipApplicationSession` already exists.
5. If found, that `SipApplicationSession` associates `SipServletRequest` and `SipSession`; otherwise, a new `SipApplicationSession` is created.

In this example, the first request with the Request-URI, "sip:mychatroom1@example.com", causes the creation of a `SipApplicationSession` object. Other requests with the same Request-URI, though still initial, are routed to the same `SipApplicationSession` instance through Session Key targeting.

Note that since it is left up to applications to generate the keys for session association, they can use any information at their disposal to generate a key. This accords tremendous flexibility to applications to associate initial requests or new `SipSession`s with existing `SipApplicationSession`s.

Similar to the other session targeting mechanisms described below, further processing of the request can result in an application chain different from the already existing chain of which the targeted `SipApplicationSession` is a part.

As noted above, this Session Key based targeting mechanism is different from the other two session targeting mechanisms described below. This is because the targeting mechanism does not operate until after the Application Router selects an application. That is, the SessionKey based targeting mechanism does not constrain the Application Router in any way regarding application selection. Consequently, for targeted requests that use session keys, the `SipTargetedRequestInfo` argument passed to the Application Router's `getNextApplication()` method is set to null.

18.10.3 The Encode URI Mechanism

Note: The encode URI mechanism is deprecated.

The `encodeURI()` method is defined on `SipApplicationSession`. When called with a URI, it encodes an identifier of `SipApplicationSession` in the URI, for example, by adding the `sipappsessionid` parameter. For SIP and SIPS URIs, the container may also rewrite the host, port, and transport protocol components of the URI based on its knowledge of local listen points.

Subsequently, if the encoded URI appears in the top Route header or as the Request-URI in a request received by the container, the container must associate this request with the identified `SipApplicationSession`. If the container is not able to find `SipApplicationSession` by using the encoded URI, it should treat the request as a normal, untargeted initial request.

The `encodeURI` method allows applications to correlate events that would otherwise be treated as independent.That is, as belonging to different `SipApplicationSession`s. For example, an application might send an instant message(IM) with an HTML body to someone. The IM body may then contain a SIP URI pointing back to the SIP Servlet container and to the `SipApplicationSession` in which the IM was generated. This ensures that an INVITE triggered by the IM recipient accessing that URI is associated with this `SipApplicationSession` when received by the container.

Upon receiving an initial request for processing, a container MUST check the top-most Route header and Request-URI (in that order) to determine whether it contains an encoded URI. If it does, the container MUST use the encoded URI to locate the targeted `SipApplicationSession` object. If a valid `SipApplicationSession` is found, the container must determine the name of the application that owns the `SipApplicationSession` object. When calling the Application Router's `getNextApplication()` method, the `SipTargetedRequestInfo` object must be populated to indicate that this request is a targeted request of type, "`ENCODED_URI`", and must supply the application name that owns the identified `SipApplicationSession` object. If the container cannot identify the `SipApplicationSession` object with the encoded URI, it MUST not identify this request as a targeted one and the `SipTargetedRequestInfo` argument in the `getNextApplication()` method is set to null.

If an application receives a request targeted to an encoded URI and subsequently either proxies it or relays it onwards as a B2BUA without having changed the encoded URI appearing in the

Request-URI, the container receives the targeted request for processing. In this case, the container MUST handle the request in the same manner as described in the previous paragraph.

Note: It is unexpected that an application receiving a request targeted to an encoded URI would proxy it or relay it onwards. Depending upon the application and Application Router implementations, such behavior could result in a routing loop. By mandating that the container handle these requests in a consistent manner, the policy is established that it is not the container's responsibility to detect and/or remedy the possibility of routing loops due to unexpected treatment of targeted requests by applications.

18.10.4 Join and Replaces Targeting Mechanism

Container support for Join [RFC 3911] and Replaces [RFC 3891] is optional in this specification. Support for [RFC 3911] and [RFC 3891] is indicated by the presence of the "join" and "replaces" option tags respectively in the `javax.servlet.sip.supported` list. (See section 3.3 Extensions Supported.) If a container supports Join and Replaces, it uses the following procedure when handling a request with a Join or Replaces header:

1. If a container supporting [RFC 3911] or [RFC 3891] receives an initial INVITE request with a Join or Replaces header, the container must first ensure that the request passes the RFC-defined validation rules. If the container does not find a dialog matching the Join or Replaces header, the container does not respond with 481 directly. After the container invokes an application, the application may choose to respond to the request with 481.

2. For a request that passes the validation step, the container MUST attempt to locate the `SipSession` object matching the tuple from the Join or Replaces header. In locating this `SipSession`, the container MUST not match a `SipSession` owned by an application that acted as a proxy for the candidate `SipSession`. It must match only a `SipSession` for an application that acted as a UA. If the matching `SipSession` is found, the container must locate the corresponding `SipApplicationSession` object along with the name of the application that owns `SipApplicationSession`.

3. The container MUST populate a `SipTargetedRequestInfo` object with the request's header type (e.g., `JOIN` or `REPLACES`) and with the application name that owns the identified `SipApplicationSession` and `SipSession`. The container MUST then pass the request to the Application Router's `getNextApplication()` method as a targeted request (i.e., along with the populated `SipTargetedRequestInfo` object).

4. If no `SipSession` matching the tuple is found, the container MUST pass the request to the Application Router as an untargeted request(i.e., where the `SipTargetedRequestInfo` argument to `getNextApplication()` is null).

5. If the Application Router returns an application name that matches the application name found in step 2, the container must create a `SipSession` object and associate it with the `SipApplicationSession` identified in step 2. The association of this newly created `SipSession` with the one found in step 2 is made available to the application through the `SipSessionsUtil.getCorrespondingSipSession(SipSession session, String headerName)` method.

6. If the Application Router returns an application name that does not match the application name found in step 2, the container invokes the application by using its normal procedure, creating a new `SipSession` and `SipApplicationSession`.

7. If the Application Router returns an application name of null, the container MUST follow the procedure given in section 18.4.1 Procedure for Routing an Initial Request with the following modification: If the Request-URI does not point to another domain and there is no Route header, the container should not send the request because it will cause a loop. Instead, the container must reject the request with a 404 Not Found final response with a no Retry-After header.

If an application receives a request targeted to a particular `SipApplicationSession` because of the presence of a Join or Replaces header, and that application subsequently proxies it or relays it onwards as a B2BUA without having changed or removed the Join/Replaces header, the container receives the targeted request for processing. In this case, the container MUST handle the request in the same manner as described above.

Note: As with applications handling encoded URI requests, an application receiving a targeted request due to a Join/Replaces header is not expected to proxy it or relay it onwards, leaving the Join/Replaces header intact in the request. Depending upon the application and Application Router implementations, such behavior could result in a routing loop. By mandating that the container handle such requests in a consistent manner, the policy is established that it is not the container's responsibility to detect and/or remedy the possibility of routing loops due to unexpected treatment of targeted requests by applications. Container implementations MAY allow configuration options to automatically reject all requests with Replaces and Join headers unconditionally or when no matching `SipSession` is found.

18.10.5 Resolving Session Targeting Conflicts

An initial request could employ one, two, or all three session targeting mechanisms. For example, a request whose Request-URI is an encoded URI could also contain a Join header. The application selected to handle this request could use the Session Key based targeting mechanism to identify an existing `SipApplicationSession`. In this case, it is possible that one, two, or even three distinct `SipApplicationSessions` are simultaneously targeted by the three different mechanisms when processing a single initial request. In the presence of such conflicts, the container should use the following priority order to determine which `SipApplicationSession` object is targeted by a particular initial request:

1. Join/Replaces
2. Session key based targeting
3. Encoded URI

In other words, if the request contains a Join or a Replaces header, the container uses that header value to identify an existing `SipApplicationSession`. If an application and `SipApplicationSession` are identified, that `SipApplicationSession` is the one targeted by the request even if the request has a Request-URI that is an encoded URI recognized by this container. Furthermore, when invoking the application identified by the Join or Replaces header, the container MUST not consult the application's session key method even when that method is present. However, if an initial request has a Request-URI that is an encoded URI recognized by the container, the container MUST call the application's session key generating method even though an `SipApplicationSession` has already been identified by the encoded URI. In this case, according to the above priority list, if the session key based targeting mechanism identifies a different `SipApplicationSession` than the one identified by the encoded URI, the container uses the `SipApplicationSession` identified by the session key targeting.

19 Mapping Requests To Servlets

18 Application Selection And Composition Model described the Application Router component and how it is instrumental in the application selection process. Servlet mapping is a mechanism for selecting a servlet after the container selects an application.

19.1 Multiple Servlets

One SIP Servlet application can comprise of many different servlets, potentially written by different developers. All of these servlets, however, are related such that together they provide the services of a complete application. Roughly, a servlet can be considered as a means for separating responsibilities between servlet classes or as a means of modularizing a project so that multiple developers can work on the same application.

Having said that, the logic coded as different servlets could be coded into just one servlet class in the application (with business logic in non-Servlet classes). Multiple servlets in a SIP Servlet application is not strictly a specification requirement but can be used to modularize code and distribute the responsibility of handling different SIP messages amongst them. This specification supports multiple servlets packaged in a single application archive.

19.2 Servlet Selection

After an application is selected for an initial request, one of the servlets within the application is invoked next. We have seen that the application can be comprised of multiple servlets. The 1.1 version of the specification introduces the declaration for a main servlet. One and only one servlet amongst the servlets in the application can be marked as the main servlet. When using this mechanism for servlet selection, if there is only one servlet in the application, this declaration is optional and the lone servlet becomes the main servlet. However, if there are multiple servlets present in the application while using this mechanism, one of those servlets MUST be declared as the main servlet. This declaration can be done in the `@SipApplication` annotation's "`mainServlet`" element (see 22.3.3 @SipApplication Annotation) or by the presence of the element `<main-servlet />` in the deployment descriptor. The main servlet MUST then be invoked for the initial request of the selected application.

The main servlet is tasked with the responsibility of processing the initial request. The main servlet can also forward the request to another servlet within the application. It can do that by using the `RequestDispatcher` interface (see 8.2.9 The SipSession Handler). If needed, the servlets can pass request attributes while passing requests back and forth amongst them to convey some information.

The servlet that handles the request becomes the handler for the `SipSession` that may come into existence and is delivered the subsequent requests and responses directly by the container. The handling servlet or the main servlet can also designate any other servlet as the handler for subsequent request or responses by calling the `SipSession.setHandler()` method.

19.2.1 Servlet Mapping Rules

Version 1.0 of this specification specifies an XML-based rules language, which can be used for selecting servlets. A rule consists of a set of conditions, each of which test some property of the incoming request. The rule language specified here is defined in terms of an object model for SIP requests (see Appendix D SIP Request Object Model). Rules are simply predicates over SIP requests. Given a SIP request, a rule evaluates to true or false. A container may trigger a matching rule in which case the corresponding servlet is invoked to process the request.

The choice of which matching rule to trigger for an initial request is a matter of container policy. The following two conditions must be observed by all containers when choosing between several matching rules:

- A triggered rule must match at the time the corresponding SIP servlet is invoked.
- If two or more rules belonging to the same application match, they must be triggered in the order they are listed in the deployment descriptor.

19.2.1.1 Example mapping rule

```
INVITE sip:watson@boston.bell-tel.com SIP/2.0
From: A. Bell <sip:a.g.bell@bell-tel.com>;tag=3pcc
To: T. Watson <sip:watson@bell-tel.com>
...
```

The SIP request shown above evaluates the following XML mapping rule to false.

```
<pattern>
    <and>
        <equal>
            <var>request.method</var>
            <value>INVITE</value>
        </equal>
        <not>
            <contains ignore-case="true">
                <var>request.from.display-name</var>
                <value>bell</value>
            </contains>
        </not>
        <subdomain-of>
            <var>request.from.uri.host</var>
            <value>bell-tel.com</value>
        </subdomain-of>
    </and>
</pattern>
```

In this example, the first and third clauses of the *and* are satisfied but the rule fails because the second clause is false (the From display name contains the string “bell” when disregarding the case).

Note: This specification makes a restriction that only one servlet selection mechanism, either the `<main-servlet />` OR `<servlet-mapping />`, shall be employed for a given application. After the Application Router chooses a application for an initial request, the container chooses the servlet to be invoked either using the `<main-servlet />` declaration OR the `<servlet-mapping />` rules. Applications employing both mechanisms via annotations or descriptor declarations MUST fail to deploy. If multiple servlets are present in applications, one of the two servlet selection mechanisms MUST be used.

19.2.2 Compatibility with v1.0 Specification

The applications compliant with v1.0 of this specification (JSR 116) can be deployed on containers compliant with this specification without any change. When the Application Router selects the v1.0 application for invocation, the container MUST select the servlet going through the servlet mapping rules. The main-servlet declaration can be used only by applications written compliant to v1.1 (or later) of this specification.

20 Security

Servlet applications are created by application developers who give, sell, or otherwise transfer the application to a deployer for installation into a runtime environment. Application developers need to communicate to deployers how the security is to be set up for the deployed application. This is accomplished declaratively by use of the deployment descriptor.

This chapter describes deployment representations for security requirements. Similar to servlet application directory layouts and deployment descriptors, this chapter does not describe requirements for runtime representations. It is recommended, however, that containers implement the elements set out here as part of their runtime representations.

20.1 Introduction

A servlet application represents resources that can be accessed by many users. These resources are accessed over unprotected, open networks such as the Internet. In this environment, a substantial number of servlet applications have security requirements.

Although the quality assurances and implementation details may vary, servlet containers have mechanisms and infrastructure for meeting these requirements that share some of the following characteristics:

- **Authentication:** The means by which communicating entities prove to one another that they are acting on behalf of specific identities that are authorized for access.
- **Access control for resources:** The means by which interactions with resources are limited to collections of users or programs for the purpose of enforcing integrity, confidentiality, or availability constraints.

- **Data integrity:** The means used to prove that information was not modified by a third party while in transit.
- **Confidentiality or data privacy:** The means used to ensure that information is made available only to users who are authorized to access it.

20.2 Declarative Security

Declarative security refers to the means of expressing an application's security structure, including roles, access control, and authentication requirements in a form external to the application. The deployment descriptor is the primary vehicle for declarative security in servlet applications. The deployer maps the application's logical security requirements to a representation of the security policy that is specific to the runtime environment. At runtime, the servlet container uses the security policy representation to enforce authentication and authorization.

The security model applies to servlets invoked to handle requests on behalf of either caller or callee. The security model does not apply when a servlet uses `RequestDispatcher` to invoke a static resource or servlet using a `forward`.

20.3 Programmatic Security

Programmatic security is used by security aware applications when declarative security alone is not sufficient to express the security model of the application. Programmatic security consists of the following methods of the `SipServletMessage` interface:

- `getRemoteUser.` returns the user name that the client used for authentication.
- `isUserInRole.` determines whether a remote user is in a specified security role.
- `getUserPrincipal.` returns the principal name of the current user and returns a `java.security.Principal` object.

These methods allow servlets to make business logic decisions based on the information obtained.

If no user has been authenticated, the `getRemoteUser` method returns null, the `isUserInRole` method returns false, and the `getUserPrincipal` method returns null.

Note: A response may contain credentials of the UAS. For this reason, the programmatic security methods apply to responses as well as to requests. However, since there is no mechanism for a proxy to challenge a UAS upon unsuccessful response authentication, the SIP deployment descriptor cannot express a requirement that responses be authenticated.

The `isUserInRole` method expects a `String` user `role-name` parameter. A `security-role-ref` element should be declared in the deployment descriptor with a `role-name` sub-element containing the rolename to pass to the method. A `security-role` element should contain a `role-link` sub-element whose value is the name of the security role that the user may be mapped into. The container uses the mapping of `security-role-ref` to `security-role` when determining the return value of the call.

For example, to map the security role reference "FOO" to the security role with `role-name` "manager", the syntax is:

```
<security-role-ref>
  <role-name>FOO</role-name>
  <role-link>manager</role-link>
</security-role-ref>
```

In this case, if the servlet called by a user belonging to the "manager" security role made the API call `isUserInRole("FOO")`, the result would be true.

If no `security-role-ref` element matching a `security-role` element has been declared, the container must default to checking the `role-name` element argument against the list of `security-role` elements for the servlet application. The `isUserInRole` method references the list to determine whether the caller is mapped to a security role. The developer must be aware that the use of this default mechanism may limit the flexibility in changing rolenames in the application without having to recompile the servlet making the call.

20.4 Roles

A security role is a logical grouping of users defined by the application developer or assembler. When the application is deployed, roles are mapped by a deployer to principals or groups in the runtime environment. A servlet container enforces declarative or programmatic security for the principal associated with an incoming request based on the security attributes of the principal. This may happen in either of the following ways:

1. A deployer maps a security role to a user group in the operational environment. The user group to which the calling principal belongs is retrieved from its security attributes. The principal is in the security role only if the principal's user group matches the user group to which the security role has been mapped by the deployer.
2. A deployer maps a security role to a principal name in a security policy domain. In this case, the principal name of the calling principal is retrieved from its security attributes. The principal is in the security role only if the principal name is the same as a principal name to which the security role was mapped.

20.5 Authentication

A SIP user agent can authenticate a user to a SIP server by using, for example, one of the following mechanisms:

- SIP digest authentication
- The P-Asserted-Identity header, thus trusting an upstream/downstream proxy to authenticate the caller/callee, as specified in [privacy]
- The Identity and Identity-Info headers, as specified in [RFC 4474]

The HTTP Servlet specification also allows user authentication based on SSL. With SSL, TLS, and IPSec, it is actually the previous hop that is being authenticated and from which the user-data constraints are enforced. As proxies are more common in SIP and may not be strongly associated with the UAC, the entity authenticating itself on an incoming TLS connection is not, generally speaking, the UAC itself.

For this reason, SIP servlet containers do not typically perform authentication based on credentials received as part of a TLS handshake. However, it is possible that in some environments it is known that the TLS connection really does identify the UAC and in such cases it is reasonable for the container to reflect this knowledge in its implementation of the declarative and programmatic security features discussed here. These security features relate to end-users, not proxies, but it is up to individual containers to determine what constitutes an authenticated message.

20.6 Server Tracking of Authentication Information

As the underlying security identities (such as users and groups) to which roles are mapped in a runtime environment are environment specific rather than application specific, it is desirable to:

1. Make login mechanisms and policies a property of the environment the servlet application is deployed in.
2. Be able to use the same authentication information to represent a principal to all applications deployed in the same container, including converged applications.
3. Require re-authentication of users only when a security policy domain boundary is crossed.

Therefore, a servlet container is required to track authentication information at the container level (rather than at the servlet application level). This allows users authenticated for one servlet application to access other resources managed by the container permitted to the same security identity.

20.7 Propagation of Security Identity in EJB^{TM} Calls

A security identity, or principal, must always be provided for use in a call to an enterprise bean. The default mode in calls to enterprise beans from servlet applications is for the security identity of a user to be propagated to the EJB^{TM} container.

In other scenarios, containers are required to allow users that are not known to the servlet container or to the EJB^{TM} container to make calls:

- SIP servlet containers are required to support access to applications by clients that have not authenticated themselves to the container.
- Application code may be the sole processor of sign on and data customization based on caller identity.

In these scenarios, a servlet application deployment descriptor may specify a `run-as` element. When it is specified, the container must propagate the caller's security identity to the EJB layer in terms of the security role name defined in the `run-as` element. The security role name must be defined for the servlet application.

For servlet containers running as part of a Java EE platform, the use of `run-as` elements must be supported both for calls to EJB components within the same Java EE application, and for calls to EJB components deployed in other Java EE applications.

20.8 Specifying Security Constraints

Security constraints are a declarative way of annotating the intended protection of applications. A SIP servlet security constraint consists of the following elements:

- resource collection (a set of servlets)
- type of authentication
- authorization constraint
- user data constraint

A resource collection is a set of servlets and SIP methods. A servlet may have one or more security constraints associated with it. Before invoking a servlet to handle an incoming request, the container must ensure that all security constraints associated with that servlet are satisfied. If this is not the case, the request must be rejected with a 401 or 407 status code.

Note: SIP servlet resource collections are identified by names of servlets being invoked instead of URL patterns as in the HTTP Servlet API.

The authentication type is an indication of whether the container should return a 401 (Unauthorized) or 407 (Proxy Authentication Required) response status code when authenticating an incoming request.

An authorization constraint is a set of security roles at least one of which users must belong to for access to resources described by the resource collection. If the user does not belong to an allowed role, the user must be denied access to the resource. If the authorization constraint defines no roles, no user is allowed access to the portion of the servlet application defined by the security constraint.

A user data constraint describes requirements for the transport layer of the client server. The requirement may be for content integrity (preventing data tampering in the communication process) or for confidentiality (preventing reading while in transit). The container must at least use TLS to respond to requests to resources marked integral or confidential. If the original request was over TCP, the container must redirect the client to the TLS port.

The security constraints can be configured in the security-constraint element of the deployment descriptor. As an alternative, applications can annotate the SIP servlet classes with `@SipSecurity` annotation as specified in section 22.3.10 Security Constraint Annotations to specify the required security constraints.

The `login-config` element allows for configuration of the authentication method that should be used, the realm name that should be used for this application, and the configuration of an identity assertion scheme.

This specification introduces the `identity-assertion` element in `login-config` to specify in the deployment descriptor the mechanism to be used for identity assertion. This element can use one of the three identity assertion mechanisms:

- **P-Asserted-Identity:** Identity MUST be asserted using the P-Asserted-Identity header as described in [RFC 3325]. This mechanism is limited to trusted domains.
- **Identity:** Identity MUST be asserted using the Identity and Identity-Info headers as described in [RFC 4474]. This mechanism provides a cryptographic approach to assure the identity of the end users that originate SIP requests, especially in an inter-domain context.
- **WebSocket**: Identity MUST be asserted using the SIP Identity associated with WebSocket connections. Refer to section 14.5 Authentication for more details.An application can also choose if the identity assertion scheme specified is REQUIRED by the application or just SUPPORTED using the element.

When P-Asserted-Identity scheme is REQUIRED by the application, the P-Asserted-Identity header MUST be present in the request. If the P-Asserted-Identity header is not present, the container MUST reject the request with a 403 response. If authorization of the Identity specified by P-Asserted-Identity header fails, the container MUST return a 403 response.

When P-Asserted-Identity scheme is SUPPORTED by the application, the container checks for the presence of the P-Asserted-Identity header in the message. If the header is not present, any other authentication mechanism configured by the user as part of the login configuration is used, for example, Digest Authentication.

When using the Identity mechanism, the Identity and Identity-Info headers are used to assert the user's identity as defined in RFC 4474. The REQUIRED or SUPPORTED behavior isthe same as the P-Asserted-Identity mechanism, except that the container MUST return a 428 error response if the required headers are not present in the request.

For example, the following shows a sample configuration for the `login-config` element:

```
<login-config>
  <auth-method>DIGEST</auth-method>
  <realm-name>example.com</realm-name>
  <identity-assertion>
    <identity-assertion-scheme>P-Asserted-Identity</identity-assertion-scheme>
    <identity-assertion-support>SUPPORTED</identity-assertion-support>
  </identity-assertion>
</login-config>
```

In this example, for every servlet that has a security constraint configured in the application, the container uses the P-Asserted-Identity header for authorization, if present. If the P-Asserted-Identity header is absent, the request is subjected to the Digest authentication mechanism configured in the element.

The `@SipLogin` annotation maps to the login-config element of the deployment descriptor, and it can be used to programmatically specify information in the login-config element. See section 22.3.3.1 @SipLogin Annotation for more details.

20.9 Default Policies

By default, authentication is not needed to access resources (servlets). Authentication is needed for requests for a resource collection only when specified by the deployment descriptor or a corresponding annotation is provided.

20.10 Authentication of Servlet Initiated Requests

This section defines the mechanism to provide authentication information on Servlet initiated requests which are challenged by the remote Proxy or UAS.

A new API introduced in this specification enables:

- the applications to set the authentication parameters while letting the container do the actual processing of the authentication headers that should be added to the request.
- the application and the container to store authentication information in an object that can be used in other requests or when additional challenges are received either from a proxy/proxies (407) or server/servers (401). Note that when forking is done, a 401 challenge may be received from multiple servers (but for the same realm).
- the container to add the required authentication headers to the request prior to sending it again.

The container MUST implement the logic to manipulate the authentication headers.

`AuthInfo` is an object that may be saved and passed to the container by the application. Only the container has knowledge of its content and format. Therefore, different implementations of the container can put different content in the object according to the implementation and according to the actual authentication mechanisms that are implemented by the container.

```
  AuthInfo SipFactory.createAuthInfo()
    Returns an empty instance of AuthInfo.
```

The following new APIs are added to support this mechanism -

```
  Iterator SipServletResponse.getChallengeRealms()
 Returns an iterator over all the realm(s) associated with the current challenge
response.
  Set SipServletResponse.getChallengeRealmSet()
  Returns an set of all the realm(s) associated with the current challenge
response.

AuthInfo.addAuthInfo(int statusCode, java.lang.String realm,
                     java.lang.String userName,
                     java.lang.String passWord)

  - statusCode is the 401/407 that was returned in the
    challenge that is being provided for in the API call.
  - realm is the realm that was returned in the challenge
    the is being provided for in this API call
```

The following adds authentication information to the `AuthInfo` object for the challenge of Type and Realm.

```
SipServletRequest.addAuthHeader(SipServletResponse challengeResponse,
                                AuthInfo authInfo)
```

The following adds the appropriate authentication header to the request. After doing this, the application can resend the request again. Note that if the request contains multiple challenges, the container can add multiple authentication headers. This is a shortcut API to be used in cases where the application does not want or needs to use the `AuthInfo` object.

```
SipServletRequest.addAuthHeader(SipServletResponse challengeResponse,
                               java.lang.String userName,
                                java.lang.String passWord)
```

20.10.1 Description of the Procedure

When receiving a challenge response (401/407) on a prior request, the application can get the challenge type (status code) and the realm from the challenge response. The application can then provide authentication parameters that are added to the `AuthInfo` object. After adding the parameters to the `AuthInfo` object, it is possible to ask the container to add the appropriate header to the request and after that the application can send the request again. If the 2xx response to a request that was successfully authenticated included the Authentication-Info header, the container MAY make use of it while sending subsequent requests.

21 Deployment Descriptor

This chapter specifies the requirements for SIP servlet container support of deployment descriptors. The deployment descriptor conveys the elements and configuration information of a servlet application between application developers, application assemblers, and deployers.

21.1 Differences from the HTTP Servlet Deployment Descriptor

The SIP Servlet deployment descriptor is similar to the HTTP Servlet deployment descriptor. The main differences are:

- The root element is `sip-app` rather than `web-app`.
- Incoming requests are mapped to a defined main servlet rather than the HTTP specific URL mapping.
- The following elements (together with child elements) have no meaning for SIP and are not present in the deployment descriptor: `form-login-config`, `mime-mapping`, `welcome-file-list`, `error-page`, `taglib`, and `jsp-file`.
- Resource collections are defined to be a named set of servlets rather than a set of URL patterns.
- The notion of filters introduced in version 2.3 of the Servlet API is not supported. The application composition model of the SIP Servlet API allows multiple applications to execute on a single request and, while this is not exactly the same as filters, it does meet many of the same requirements.

A SIP Servlet application that does not use annotations (22.3 Annotations and Resource Injection) must have a deployment descriptor that conforms to the XML XSD. This deployment descriptor exists as a file sip.xml in the /WEB-INF directory of the SIP Servlet application.

21.2 Converged SIP and HTTP Applications

Applications may have both SIP and HTTP components (and potentially components related to protocols for other, as of yet unspecified, servlet APIs). When this is the case, there will be both a web.xml HTTP deployment descriptor conforming to the DTD defined by the HTTP Servlet API, and a sip.xml deployment descriptor.

A number of deployment descriptor elements apply to the application as a whole. The following rules specify how those elements are handled when they appear in both the SIP and HTTP descriptors (the rules apply equally to the case where there are deployment descriptors for other servlet APIs alongside SIP and/or HTTP):

- **`distributable:`** if this “tagging” element is present in one deployment descriptor, it must be present in the other.
- **`context-param:`** SIP and HTTP servlets belonging to the same application are presented with a single `ServletContext`. The set of initialization parameters available from the `ServletContext` is the union of parameters specified in `context-param` elements present in the SIP and HTTP deployment descriptors. A parameter may be configured in both descriptors but in that case must have the same value in both declarations.
- **`listener:`** the set of listeners of an application is the union of listeners defined in the SIP and HTTP deployment descriptors.

Also, the application `display-name` and icons should be identical and, if present in one should be present in the other.

The `app-name` element is defined only in sip.xml. It is to be taken as the logical name for the SIP Servlet Application.

21.3 Deployment Descriptor Elements

The following types of configuration and deployment information exist in the SIP servlet application deployment descriptor and MUST be supported, with the exception of the security syntax, for all servlet containers:

- `ServletContext init` parameters

- `Session` configuration
- `Servlet` definitions
- Application lifecycle listener classes
- Error handler
- Security

Also, elements exist in the SIP Servlet application deployment descriptor to support the additional requirements of servlet containers that are part of a Java EE application server. These elements allow for looking up JNDI objects (`env-entry`, `ejb-ref`, `ejb-local-ref`, `resource-ref`, `resource-env-ref`), and are not required to be supported by containers wishing to support only the servlet specification. See the schema comments for further description of these elements.

21.4 Rules for Processing the Deployment Descriptor

The following list specifies some general rules that servlet containers and developers must note concerning the processing of the deployment descriptor for a servlet application.

- Servlet containers should ignore all leading white space characters before the first non-whitespace character. Servlets should also ignore all trailing white space characters after the last non-whitespace character for PCDATA within text nodes of a deployment descriptor.
- Servlet containers and tools that manipulate servlet applications have a wide range of options for checking the validity of a SAR/WAR. This includes checking the validity of the deployment descriptor document held within. It is recommended, but not required, that servlet containers and tools validate deployment descriptors against the XML Schema document for structural correctness.
- Configuration specified in the deployment descriptor takes precedence over the configuration specified via annotations.
- Any metadata specified via an annotation that isn't already present in the descriptor is used to augment the deployment descriptor.

Additionally, it is recommended that they provide a level of semantic checking. For example, it should be checked that a role referenced in a security constraint has the same name as one of the security roles defined in the deployment descriptor.

In cases of non-conformant servlet applications, tool sand containers should inform the deployer with descriptive error messages. High-end application server vendors are encouraged to supply this kind of validity checking in the form of a tool separate from the container.

21.5 The SIP Servlet XSD

The XSD document sip-app_2_0.xsd describes the SIP Servlet deployment descriptor.

22 Java Enterprise Edition Container

22.1 Java EE

This specification depends on technologies that are part of Java EE platform (version 6 and above) for converged Java EE features. Refer to chapter Chapter 16, “Converged Container and Applications” for a description of the converged container. Following technologies are the required dependencies for the respective convergence features described in this specification.

1. Java Naming and Directory Interface (JNDI).
2. Common Annotations Specification 1.1 or above.
3. Servlet Specification 3.0 or above.
4. Contexts and Dependency Injection for Java EE 1.0 or above.
5. Concurrency Utilities for Java EE 1.0.
6. EJB 3.1 and above.

Note that a converged SIP and Java EE application may utilize more technologies available in the Java EE platform. The above list of technologies are referred by different chapters of this specification as dependencies.

22.2 Java SE

Java SE 6.0 is the minimum version of the underlying Java platform with which SIP Servlet containers compliant with this specification must be built.

Even though the API may be revised to make use of new Java SE 6.0 constructs it must remain fully backwards compatible for the applications written for the 1.0 version of this specification.

22.3 Annotations and Resource Injection

The Java Metadata specification (JSR-175) provides a means for an application to provide configuration and dependency information in the Java code. This metadata, or annotations, are used to provide a faster and easier way for application development.

This section describes annotations and resource injection in a SIP Servlet compliant container, reflecting the work done in the Java EE through the Common Annotations for the Java Platform (JSR-250) and the Servlet specification.

The goal of this annotations work is to eliminate the need for the deployment descriptor. Here is a mapping from the deployment descriptor to the function in the annotations.

Table 22-1 Mapping from deployment descriptors to annotations

Item	Deployment descriptor	Annotation element
Icons	`small-icon` or `large-icon`	`@SipApplication` `smallIcon` or `largeIcon`
Display Name	`display-name`	`@SipApplication displayName`
Description	`description`	`@SipApplication description`
Distributable	`distributable`	`@SipApplication distributable`
Context parameters	`context-param` , `param-name` , `param-value`	Not applicable.
Listener	`listener-class`	`@SipListener`
Servlet name	`servlet-name`	`@SipServlet name`
Application name	`module-name`	`@SipApplication name`
Servlet class	`servlet-class`	`@SipServlet`
Initialization parameters	`init-param` , `param-name` , `param-value`	Not applicable, suggested to use constants.

Table 22-1 Mapping from deployment descriptors to annotations

Startup order	`load-on-startup`	`@SipServlet loadOnStartup`
Proxy timeouts	`proxy-timeout`	`@SipApplication proxyTimeout`
Session timeouts	`session-timeout`	`@SipApplication sessionTimeout`
Resources	`resource-*`	`@Resource , @Resources`
Security Roles	`security-role *`	`@DeclareRoles`
EJBs	`ejb-ref *`	`@EJB`
Run as	`run-as *`	`@RunAs`
Web Services	Not applicable	`@WebServiceRef`

22.3.1 Servlet alignment

Reflecting on the work done in the Servlet specification, this specification will require the components defined in that specification be supported. At the time of this writing, the annotations defined in the Servlet specification included `@Resource`, `@Resources`, `@PostConstruct`, `@EJB`, `@WebServiceRef`, `@DeclareRoles`, and `@RunAs`. Those definitions in the Servet specification remain the same with the following additional classes required to be injected.

Table 22-2 Interfaces and Classes which Require Annotation Support

Servlet	`javax.servlet.sip.SipServlet`
Listeners	`javax.servlet.sip.SipApplicationSessionListener`
	`javax.servlet.sip.SipApplicationSessionActivationListener`
	`javax.servlet.sip.AttributeStoreListener`
	`javax.servlet.sip.SipSessionListener`
	`javax.servlet.sip.SipSessionActivationListener`
	`javax.servlet.sip.SipErrorListener`
	`javax.servlet.sip.TimerListener`

With this specification, annotations defined as part of Servlet specification and some common Java EE annotations as defined in JSR 250 are also included. For a description of annotations listed but not described here please refer to Servlet specification and JSR 250. The complete list of annotations required to be supported is -

- `@RunAs`
- `@DeclaresRole`
- `@Resource`
- `@Resources`
- `@EJB`
- `@WebServiceRef`
- `@PostConstruct`
- `@PreDestroy`
- `@SipServlet`
- `@SipApplication`
- `@SipListener`
- `@SipApplicationKey`

Resource Injection is supported for classes whose lifecycle is controlled by the container. A non Java EE compliant implementation of this specification MUST support SIP specific annotations but it is not required to support the Java EE specific annotations.

22.3.2 @SipServlet Annotation

The `SipServlet` annotation allows for the `SipServlet` metadata to be declared without having to create the deployment descriptor. Certain values from the deployment descriptor are not needed since these are declared in the annotation in the source file of the servlet itself. First from the deployment descriptor, `servlet-class` is not needed since the annotation is declared in the class. Also, `init-parm` is not useful since it could just as easily be a static constant in the source file where the annotation exists.

The first element is the `name`. The `name` element is equivalent to the `servlet-name` element in the deployment descriptor. If the name is not provided, the servlet name used will be the short name of the class annotated.

The second element is the `applicationName` that this servlet is associated with. The application will contain the proxy settings, session settings, listeners, and some basic configuration. The `applicationName` element is an optional element. If this element is specified, it must match the name of the application. For more details about the application names, see section 9.6 Application Names

The `description` element in this annotation contains the declarative data of a servlet and can help the consumer of this application understand better what the SipServlet is and what it does. This element is optional and an empty string will be used in its place if one is not provided.

The `loadOnStartup` element of the annotation is similar to the `load-on-startup` element of the deployment descriptor. This element specifies the starting order of the servlet application within the system. If the value is a negative number, the container may choose when to start up this servlet. If the value is zero or a positive number, the container must start the servlets beginning with the lowest number. If servlets have the same `loadOnStartup` value, the container may choose which one to start first. This element defaults to a negative number, allowing the container to choose when to startup this servlet if no other value is given.

For a servlet defined via the `@SipServlet` annotation, to override values via the descriptor, the name of the servlet in the descriptor MUST match the name of the servlet specified via the annotation (explicitly specified or the default name, if one is not specified via the annotation). If the name of the servlet declared via the annotation does not match the name of the servlet declared in the deployment descriptor, the annotation specifies a new servlet declaration in addition to the other declarations in deployment descriptor, even if the servlet class remains same.

Here is the definition of the `@SipServlet` annotation:

```
package javax.servlet.sip.annotation;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
@Target({TYPE})
@Retention(RUNTIME)
@Inherited
public @interface SipServlet {
      String name() default "";
      String applicationName() default "";
      String description() default "";
      int loadOnStartup() default -1;

}
```

The first example is a basic annotated class to become a `SIP Servlet POJO`. Here, because no elements are specified, the name and display name of the servlet will be "`Weather`"

```
@SipServlet
public class Weather {

  @Invite
  public void handleInviteRequest(SipServletRequest request) {
    //...
  }
```

Here in the second example, the servlet name is "`WeatherService`" and the display name defaults to "`WeatherService`".

```
@SipServlet (name = "WeatherService")
public class Weather {

  @Invite
  public void handleInviteRequest(SipServletRequest request) {
    //...
  }
```

The class annotated by the `@SipServlet` annotation may also override all methods in `javax.servlet.sip.SipServlet` instead of following SipServlet POJO model. For example, if the class wanted to perform some action on INVITE methods, it would do the following:

```
package com.example;
import javax.servlet.sip.annotation.SipServlet;
import javax.servlet.sip.SipServletRequest;
import javax.servlet.sip.SipServlet;
@SipServlet (name = "WeatherService",
     description = "Provides weather information",
     loadOnStartup=1)
public class Weather extends SipServlet {
      public void doInvite(SipServletRequest req){
           //perform action
```

22.3.3 @SipApplication Annotation

The `@SipApplication` annotation is used to create an application level annotation for a collection of `SipServlet`s and to maintain the application level configuration which was provided by the DD. The SIP application is a logical entity which contains a set of servlets and listeners with some common configuration. This annotation provides that configuration.

This annotation may be specified either in the package level (in package-info.java) or on a class in the .war or .sar file. If there is more than one instance of this annotation in a .war or .sar file, it must result in a deployment error.

The first element is the `name`. This is an optional element and when not specified, the name of the application is deduced as per section 9.6 Application Names. This is the name the listeners and servlets reference when adding themselves to this logical application.

The second element is the display name of the application. The `displayName` element defaults to the application name unless specified through its own element in the annotation. This element is equivalent to the `display-name` element in the deployment descriptor.

The `largeIcon` and `smallIcon` elements in this annotation allow for specific icons to be used for this application. These are equivalent to the `large-icon` and `small-icon` elements in the deployment descriptor. Each takes a simple string specifying the location of the image relative to the root path of the archive this class exists within. These elements are optional.

The `description` element in this annotation is equivalent to the `description` element in the deployment descriptor. This can help the consumer of this application understand better what the application is and what it does. Often, tools that consume applications will provide the descriptions to better understand what the application does. This element is optional and a empty string will be used in its place if one is not provided.

The `distributable` element indicates to the container whether this application is developed to properly function in a distributed environment. This element set equal to true is the equivalent to the `distributable` attribute in the deployment descriptor. Just as the descriptor without the `distributable` element defaults the application to not be distributable, the distributable element in the annotation defaults to false and must be set to true if the servlet developer wishes the application to function in a distributed environment.

The `proxyTimeout` element is equivalent to the `proxy-timeout` element of the deployment descriptor. The `proxyTimeout` should specify, in whole seconds, the default timeout for all proxy operations in this application. The container may override this value as a result of its own local policy.

The `sessionTimeout` element is equivalent to the `session-timeout` element in the deployment descriptor. The `sessionTimeout` should specify, in whole minutes, the default session timeout for all application sessions created in this servlet. `SipSession`s have no timeout independent of that of the containing application session. The lifetime of a `SipSession` is tied to that of the parent application session. If the timeout is zero or a negative number, the container must ensure the default behavior of the sessions is to never time out.

An application may use the `disableCancelHandling` element to turn off the processing of CANCEL request by the container. Details of CANCEL processing by the container is explained in section 12.2.6 Receiving CANCEL for proxies and in section 13.2.3 Receiving CANCEL for user agents.

An application may use `establishDialogOnSubscribeTransaction` element to support RFC 3265 mode of establishing dialog on SUBSCRIBE transaction. Details of dialog establishment on SUBSCRIBE sessions is explained in section 8.2.3.1 Extensions Creating Dialogs.

The `version` element is equivalent to the `version` element in the deployment descriptor. It is used to designate the `version` of the sip application.

The `enableDerivedSessionAttributeCloning` element is equivalent to the `enable-derivedsession-attribute-cloning` element in the deployment descriptor. It is used to enable cloning the attributes when a derived session is created as specified in section 8.2.3.2 Derived SipSessions.

The `mainServlet` element of the annotation specifies which servlet is designated as the main servlet of the application. The concept of main servlet is described in 19 Mapping Requests To Servlets of this specification. This corresponds to `main-servlet` element of the deployment descriptor.

```
package javax.servlet.sip.annotation;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
@Target({PACKAGE,TYPE})
@Retention(RUNTIME)
public @interface SipApplication {
      String name() default "";
      String displayName() default "";
      String smallIcon() default "";
      String largeIcon() default "";
      String description() default "";
      boolean distributable() default false;
      int proxyTimeout() default 180; // seconds
      int sessionTimeout() default 3; // minutes
    String mainServlet() default "";
}
```

To use the `@SipApplication` annotation, it may be specified in a `package-info.java` file or in a class. An example of a package-info.java is shown below:

```
@javax.servlet.sip.annotation.SipApplication(
     name="WeatherApplication",
    sessionTimeout=30,
    distributable=true)
package com.example;
```

22.3.3.1 @SipLogin Annotation

The `@SipLogin` annotation allows the application developer to specify the authentication method that should be used, the realm name that should be used for this application and the configuration of an identity assertion scheme. The `@SipLogin` annotation maps to the `login-config` element in the deployment descriptor. This annotation is used within the `@SipApplication` annotation as `loginConfig` element.

```
@SipApplication(
        loginConfig=@SipLogin(
                authMethod = "DIGEST",
                realmName = "example.com",
                identityAssertion = SipLogin.IdentityAssertion.P_ASSERTED_IDENTITY,
                identityAssertionSupport = SipLogin.IdentityAssertionSupport.SUPPORTED
        )
)
package com.example;
```

In the above example, for every servlet that has a security constraint configured in the application, the container would use the P-Asserted-Identity header for authorization, if present. If P-Asserted-Identity header is absent then the request will be subjected to the Digest authentication mechanism configured in the element.

For more information on specifying authentication constraints, refer section 20.8 Specifying Security Constraints.

22.3.4 @SipListener Annotation

The `@SipListener` annotation allows the application developer to specify a listener without declaring it in the deployment descriptor of the application. The `@SipListener` annotation provides an alternative to the `<listener>` deployment descriptor element. The listener type is inferred from the interfaces implemented by the target class.

```
package javax.servlet.sip.annotation;

import static java.lang.annotation.ElementType.TYPE;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Target;

@Target({TYPE})
@Retention(RUNTIME)
@Inherited
public @interface SipListener {
  String applicationName() default "";
   String description() default "";
}
```

The class annotated by the `@SipListener` must implement at least one of the listener interfaces described in this specification. If `applicationName` element is specified, it must match the name of the application. For more details about the application names, see section 9.6 Application Names.

```
      package com.example;
      import javax.servlet.sip.annotation.SipListener;
      import javax.servlet.sip.SipApplicationSessionListener;
      import javax.servlet.sip.SipApplicationSessionEvent;

      @SipListener
      public class MyApplicationSessionListener implements
SipApplicationSessionListener {
        public void sessionDestroyed(SipApplicationSessionEvent ev) {.}
        public void sessionCreated (SipApplicationSessionEvent ev){.}
        public void sessionExpired (SipApplicationSessionEvent ev){.}
      }
```

22.3.5 @SipApplicationKey Annotation

The `@SipApplicationKey` annotation is used when the application wants to associate the incoming request (and the corresponding session) with a certain `SipApplicationSession`.

A method this annotation is attached to MUST be a public and static method, MUST return a `String` and MUST accept the incoming request as the single argument. If a method annotated with `@SipApplicationKey` is not of this signature, the container MUST fail deployment of the application.

If a given SIP application (SAR/WAR) contains more than one `@SipApplicationKey` annotated method with `SipServletRequest` as parameter, the container MUST fail the deployment.

Similarly, if a given SIP application (SAR/WAR) contains more than one `@SipApplicationKey` annotated method with `HttpServletRequest` as paramater, the container MUST fail the deployment.

Following is the definition of `@SipApplicationKey` annotation.

```
package javax.servlet.sip.annotation;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
@Target({METHOD})
@Retention(RUNTIME)
@Inherited
public @interface SipApplicationKey {
 String applicationName() default "";
}
```

22.3.5.1 @SipApplicationKey for SipServletRequests

For sip requests the `String` returned by this method is to be used as a key to associate the incoming request to an existing `SipApplicationSession` by the container as specified in 18.10.2 Session Key Based Targeting Mechanism. The container should treat a "null" return or an invalid session id as a failure to obtain a key from the application. It is recommended that the container create a new `SipApplicationSession` for the incoming request in such a case. The annotated method MUST not modify the `SipServletRequest` passed in. An example of a method annotated with `@SipApplicationKey` is shown below.

```
@javax.servlet.sip.annotation.SipApplication
package com.example;
import javax.servlet.sip.annotation.SipApplicationKey;
import javax.servlet.sip.SipServletRequest;
public class WeatherMapper {
    @SipApplicationKey
    public static String sessionKey(SipServletRequest req){
        return hash(req.getRequestURI +
                    getDomain(req.getFrom());
```

```
                  }
        }
```

22.3.5.2 @SipApplicationKey for HttpServletRequests

For http requests, the string returned by this method is to be used as a key to associate the incoming request to an existing `SipApplicationSession` by the container as specified in section 16.5 Association of HTTP Session With SipApplicationSession. If the method returns null, an empty string or throw an exception, then container would continue the normal processing of the request without creating a new `SipApplicationSession`.

22.3.6 Annotation for SipFactory Injection

The `@Resource` annotation defined in the Common Annotations for Java Platform (JSR 250) is to be used for `SipFactory` injection.

When this annotation is applied to a field of type `SipFactory` in a converged application on a converged container, the container MUST inject an instance of `SipFactory` into the field when the application is initialized. This annotation can be used in place of the existing `ServletContext` lookup for the `SipFactory` from within a Servlet.

```
SipFactory sf =
  (SipFactory) getServletContext().getAttribute(SIP_FACTORY);
```

Usage above and

```
@Resource
private SipFactory sf;
```

are equivalent. For converged containers, the injected `SipFactory` would appear as `sip/<appname>/SipFactory` in the application scoped JNDI tree, where the `appname` is the name of the application as identified by the container (see 9.6 Application Names).

In this example, this annotation can be used in any SIP Servlet or a Java EE application component deployed on the converged container in the same EAR archive file. The container must inject an instance of `SipFactory` into the field ("`sf`" in this example) at the time of application initialization. For the purposes of associating a `Servlet`, with, say the responses received to the request created using the `SipFactory`, the container must use the application's main servlet as defined in 19.2 Servlet Selection. This can be changed to another servlet through the `setHandler` mechanism in the `SipSession`.

Note that containers MAY allow the `@Resource` annotation to be present outside of SIP applications. In such cases, the `name` element of the `@Resource` annotation MUST identify the JNDI name of the desired factory to inject. Otherwise, an invocation of `SipFactory.createApplicationSession()` would be unable to determine the intended SIP application.

22.3.7 Annotation for SipSessionsUtil Injection

The `@Resource` annotation defined in the Common Annotations for Java Platform (JSR 250) is to be used to inject an instance of the `SipSessionsUtil` utility class for application session instance lookup.

This annotation can be used in place of the following `ServletContext` based lookup for the `SipSessionsUtil`.

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

And the annotation based access -

```
@Resource
SipSessionsUtil s;
```

are equivalent. The injected `SipSessionsUtil` appears as `sip/<appname>/SipSessionsUtil` in the application scoped JNDI tree, where the `appname` is the name of the application as identified by the container (see 9.6 Application Names). When multiple applications are packaged in the same EAR, access to `SipSessionsUtil` for a specific application is possible by specifying the JNDI `name` element of the `@Resource` annotation in a manner similar to 22.3.6 Annotation for SipFactory Injection. The `SipSessionsUtil` is described in chapter 16 Converged Container and Applications.

22.3.8 Annotation for DnsResolver Injection

The `@Resource` annotation defined in the Common Annotations for Java Platform (JSR 250) is to be used to inject an instance of the `DnsResolver` utility class for DNS Enum lookup of telephone numbers.

This annotation can be used in place of the following `ServletContext` based lookup for the `DnsResolver`.

```
DnsResolver s =
  (DnsResolver)
getServletContext().getAttribute("javax.servlet.sip.DnsResolver");
```

And the annotation based access -

```
@Resource
DnsResolver resolver;
```

are equivalent. The injected `DnsResolver` appears as `sip/DnsResolver` in the application scoped JNDI tree.

Since Application Routers may want to access the `DnsResolver`, it is also available on global JNDI tree (java:global). The `DnsResolver` is described in section 5.2.3 Resolving Telephone Numbers to SipURI.

22.3.9 Annotation for TimerService Injection

The `@Resource` annotation also can be used to inject an instance of the `TimerService` for scheduling timers.

This annotation can replace the following `ServletContext` based lookup of the `TimerService`.

```
TimerService t =
  (TimerService)
getServletContext().getAttribute(TIMER_SERVICE);
```

And the annotation based access -

```
@Resource
TimerService t;
```

are equivalent. The injected `TimerService` appears as `sip/<appname>/TimerService` in the application scoped JNDI tree, where the appname is the name of the application as identified by the container (see 9.6 Application Names). The `TimerService` is described in 11 Timer Service.

22.3.10 Security Constraint Annotations

Security constraints are a declarative way of annotating the intended protection of applications. A detailed explanation of security constraints can be found in section 20.8 Specifying Security Constraints. Instead of security-constraint element of the deployment descriptor, the annotations specified below can be used configure security constrains on applications.

22.3.10.1 @SipSecurity Annotation

This annotation is used on a Servlet implementation class to specify security constraints to be enforced by the container on SIP protocol messages. The SIP servlet container will enforce these constraints on the annotated SIP servlet. The `@SipSecurity` annotation provides an alternative mechanism for defining access control constraints equivalent to those that could otherwise have been expressed declaratively via security-constraint elements in the portable deployment descriptor. Note that if both security-constraint of the deployment descriptor and the @SipSecurity annotation is present for the same servlet, then the configuration in the deployment descriptor will take precedence.

```
@Inherited
@Documented
@Target({TYPE})
@Retention(RUNTIME)
public @interface SipSecurity {
  SipConstraint value() default @SipConstraint;
  SipMethodConstraint[] sipMethodConstraints() default {};
}
```

Table 22-3 The SipSecurity Annotation

Element	Description
value	the SipConstraint that defines the protection to be applied to all SIP methods that are NOT represented in the array returned by sipMethodConstraints.
`sipMethodConstraints`	the array of SIP method specific constraints

22.3.10.2 @SipConstraint Annotation

The `@SipConstraint` annotation is used within the `@SipSecurity` annotation to represent the security constraint to be applied to all SIP protocol methods for which a corresponding `@SipMethodConstraint` does NOT occur within the `@SipSecurity` annotation.

```
@Documented
@Target({})
@Retention(RUNTIME)
```

```
public @interface SipConstraint {
  SipSecurity.EmptyRoleSemantic value() default PERMIT;
  String[] rolesAllowed() default {};
  SipSecurity.TransportGuarantee transportGuarantee() default NONE;
  boolean proxyAuthentication() default false;
}
```

Table 22-4 The SipConstraint annotation

Element	Description
`value`	The default authorization semantic that applies (only) when rolesAllowed returns an-empty array
`rolesAllowed`	An array containing the names of the authorized roles
`transportGuarantee`	The data protection requirements that must be satisfied by the connections on which requests arrive
`proxyAuthentication`	A boolean indicating whether the container should use proxy authentication challenges.

22.3.10.3 @SipMethodConstraint Annotation

The `@SipMethodConstraint` annotation is used within the `@SipSecurity` annotation to represent security constraints on specific SIP protocol messages.

```
@Documented
@Target({})
@Retention(RUNTIME)
public @interface SipMethodConstraint {
  String value();
  SipSecurity.EmptyRoleSemantic emptyRoleSemantic() default PERMIT;
  TransportGuarantee transportGuarantee() default TransportGuarantee.NONE;
  boolean proxyAuthentication() default false;
  String[] rolesAllowed() default {};
}
```

Table 22-5 The SipMethodConstaint Annotation

Element	Description
`value`	The SIP protocol method name
`emptyRoleSemantic`	The default authorization semantic that applies (only) when rolesAllowed returns an empty array
`transportGuarantee`	The data protection requirements that must be satisfied by the connections on which requests arrive
`proxyAuthentication`	A boolean indicating whether the container should use proxy authentication challenges.
`rolesAllowed`	An array containing the names of the authorized roles

22.3.10.4 Access Control Details

The @SipSecurity annotation may be specified on (that is, targeted to) a SIP Servlet implementation class, and its value is inherited by subclasses according to the rules defined for the @Inherited meta-annotation. At most one instance of the @SipSecurity annotation may occur on a SIP Servlet implementation class.

When one or more `@SipMethodConstraint` annotations are defined within a `@SipSecurity` annotation, each `@SipMethodConstraint` defines the `security-constraint` that applies to the SIP protocol method identified within the `@SipMethodConstraint`. The encompassing @SipSecurity annotation defines the `@SipConstraint` that applies to all SIP protocol methods other than those for which a corresponding `@SipMethodConstraint` is defined within the `@SipSecurity` annotation.

When a class has not been annotated with the `@SipSecurity` annotation, the access policy that is applied to a servlet mapped from that class is established by the applicable `security-constraint` elements, if any, in the corresponding portable deployment descriptor.

22.3.10.5 Examples

The following examples demonstrate the use of the `SipSecurity` annotation.

Listing 22-1 for all SIP methods, no auth-constraint

```
@SipSecurity
@SipServlet
public class SipPOJO {
}
```

Listing 22-2 for all SIP methods, no auth-constraint, proxy authentication required

```
@ServletSecurity(@SipConstraint(proxyAuthentication = true))
@SipServlet
public class SipPOJO {
}
```

Listing 22-3 for all SIP methods, all access denied

```
@ServletSecurity(@SipConstraint(EmptyRoleSemantic.DENY))
@SipServlet
public class SipPOJO {
}
```

Listing 22-4 for all SIP methods, auth-constraint requiring membership in Role R1

```
@ServletSecurity(@SipConstraint(rolesAllowed = "R1")))
@SipServlet
public class SipPOJO {
}
```

Listing 22-5 for All SIP methods except REGISTER and INVITE, no constraints; for methods REGISTER and INVITE, auth-constraint requiring membership in Role R1; for REGISTER, confidential transport required

```
@SipSecurity((sipMethodConstraints = {
@SipMethodConstraint(value = "INVITE", rolesAllowed = "R1"),
@SipMethodConstraint(value = "REGISTER", rolesAllowed = "R1",
transportGuarantee = TransportGuarantee.CONFIDENTIAL)
})
@SipServlet
public class SipPOJO {
}
```

Listing 22-6 for all SIP methods except MESSAGE auth-constraint requiring membership in Role R1; for MESSAGE, no constraints

```
@SipSecurity(value = @SipConstraint(rolesAllowed = "R1"),
sipMethodConstraints = @SipMethodConstraint("MESSAGE"))
@SipServlet
public class SipPOJO {
}
```

Listing 22-7 for all HTTP methods except OPTIONS, auth-constraint requiring membership in Role R1; for OPTIONS, all access denied

```
@ServletSecurity(value = @HttpConstraint(rolesAllowed = "R1"),
httpMethodConstraints = @HttpMethodConstraint(value="OPTIONS",
emptyRoleSemantic = EmptyRoleSemantic.DENY))
@SipServlet
public class SipPOJO {
}
```

22.4 Annotation Parsing

The SIP servlet container parses Java class files in order to check for presence of the annotations described in the previous section. For .war and .sar files, the WEB-INF/classes as well as WEB-INF/lib directories are scanned. Containers may optionally process annotations for classes found elsewhere in the application's classpath.

22.5 Context and Dependency Injection (CDI)

The Contexts and Dependency Injection (CDI) specification defines a set of contextual services, provided by Java EE containers, aimed at simplifying the creation of applications that use both web tier and business tier technologies. This technology along with *The Dependency Injection for Java (DI)* specification (JSR 330) defines a standard set of annotations (and one interface) for use on injectable classes.

The SIP servlet container is required to support CDI like any other Java EE container mentioned in the CDI specification. These capabilities include, but not limited to, Managed Beans and related capabilities, Dependency Injection, Transparent Creation, Destruction and Scoping to a particular context, Interceptor Bindings, Events and Portable Extensions. For a more complete list of capabilities refer the CDI specification.

22.5.1 SIP Servlet Managed Beans

As per Table EE.5-1 of the Java EE Platform Specification, Servlets and Event Listeners are component classes. Similarly SIP Servlet classes (including Servlet POJOs) and any Event Listeners that are specified declaratively (either using annotations or by an entry in the deployment descriptor) are component classes. These component classes qualify as CDI managed beans. However non-contextual instances of these classes are created by SIP servlet container as specified in section titled *"Support for Dependency Injection"* of the Java EE specification.

22.5.2 CDI Built-In Beans

Section 3.7 of the CDI specification details the requirements on the built-in beans that need to be supplied by the Java EE and Servlet Containers. SIP servlet container is required to provide `javax.transaction.UserTransaction`, `java.security.Principal`, `javax.servlet.ServletContext` as built-in beans.

22.5.3 SIP Specific CDI Beans

A SIP servlet container is required to provide the following built-in beans, all of which have the qualifier `@Default`:

- a bean with bean type `javax.servlet.sip.SipFactory` allowing injection of a reference to the `SipFactory`. This enable an application to use `@Inject private SipFactory sf` instead of `@Resource private SipFactory sf;`
- a bean with bean type `javax.servlet.sip.SipSessionsUtil` allowing injection of a reference to the `SipSessionsUtil`. This enable an application to use `@Inject private SipSessionsUtil ssu` instead of `@Resource private SipSessionsUtil ssu;`
- a bean with bean type `javax.servlet.sip.TimerService` allowing injection of a reference to the `TimerService`. This enable an application to use `@Inject private TimerService timer` instead of `@Resource private TimerService timer;`
- a bean with type `javax.servlet.sip.SipApplicationSession` allowing injection of a reference to the `SipApplicationSession`.
- a bean with bean type `javax.servlet.sip.DnsResolver` allowing injection of a reference to the `DnsResolver`. This enable an application to use `@Inject private DnsResolver resolver` instead of `@Resource private DnsResolver resolver;`

22.5.3.1 Specifying Application Name with built-in beans

As specified in sections 22.3.6 Annotation for SipFactory Injection, 22.3.7 Annotation for SipSessionsUtil Injection and 22.3.9 Annotation for TimerService Injection of this specification, the SipFactory instances scoped to the application for converged containers. Thus the injection points where these built in beans injected may be further specify `@javax.servlet.sip.annotation.ApplicationName` to indicate the name of the application. Thus an application may use the following to inject a `SipFactory` specific to an application Foo.
`@ApplicationName("Foo") Inject private SipFactory sf`

22.5.4 CDI Built-in Scope Types

A SIP servlet container is required to support `@ApplicationScoped` and `@Dependent` built-in scope types. More information on these built-in scopes are explained in section 2.4.1 and 6.7 of the CDI specification.

The *application context* is provided by a built-in context object for the built-in scope type `@ApplicationScoped`. For SIP servlets, the application scope is active during

- during the `service()` method of any SIP servlet in the sip or converged application or when any `EventListener` specified declaratively is invoked.
- when container invoke methods in any Sip Servlet POJO.
- when annotated methods `@SipApplicationKey` is invoked.
- during the invocation of any asynchronous tasks started by an application using concurrency utilities.
- during invocation of a `TimerListener`
- when the disposer method or `@PreDestroy` callback of any bean with any normal scope other than `@ApplicationScoped` is called, and
- during `@PostConstruct` callback of any bean.

22.5.5 SIP Specific Built-in Scope Types

Apart from the built-in scopes specified in section 22.5.4 CDI Built-in Scope Types a SIP servlet container supports the following built-in scopes.

- `@SipApplicationSessionScoped`
- `@SipInvocationScoped`

These context objects (container provided implementation of javax.enterprise.context.spi.Context) will be active when SIP servlet objects are invoked as specified in the following sections.

22.5.5.1 SIP Application Session Context lifecycle

The *sip application session context* is provided by a built-in context object for the built-in scope type `@SipApplicationSessionScoped`. The application session scope is active as described below.

- During any `service()/doXXX()` method of any SIP servlet
- Whenever container invokes a method of a SIP servlet POJO.
- Whenever any method of a SIP Listener is invoked.
- Whenever a contextual task or a method of a contextual object which has a SIP application session as a context is executed. For more information about concurrency utilities refer section 17.3.2 Concurrency Utilities

The application session context is shared during any number of invocations of the above mentioned scenarios that occur for the same `SipApplicationSession`. The application session context is destroyed when the `SipApplicationSession` becomes invalid after all listeners have been called, and at the very end of any application invocation (e.g. servlet service method) if the invalidation was initiated from the application.

Note that `SipApplicationScoped` is a passivating scope. Please refer to section 6.6.4 of CDI specification for more information on passivating scopes.

An event with the `SipApplicationSession` as payload and with qualifier `@Initialized(SipApplicationSessionScoped.class)` is fired when the application session context is initialized and an event with qualifier `@Destroyed(SipApplicationSessionScoped.class)` is fired when the application session context is destroyed.

When a task, which is not a CDI managed bean is submitted or scheduled as specified in section 17.3.2 Concurrency Utilities, contextual instances of the beans cannot be injected to the task. In such situations, application can programmatically lookup the contextual instances of the beans under the sip application session context using `<T> T getManagedBean(Class<T> type, Annotation... qualifiers)` method.

22.5.5.2 Invocation Context lifecycle

The *sip invocation context* is provided by a built-in context object for the built-in scope type `@SipInvocationScoped`. The invocation context is active as described below.

- During any `service()/doXXX()` method of any SIP servlet.
- Whenever container invokes a method of a SIP servlet POJO.
- Whenever any method of a SIP Listener is invoked.
- Whenever a contextual task or a method of a contextual object which has a SIP application session as a context is executed. For more information about concurrency utilities refer section 17.3.2 Concurrency Utilities

The invocation context is destroyed at the end of the service method, when the invoked SIP servlet POJO method returns or at the end of the SIP listener method.

An event with the qualifier `@Initialized(SipInvocationScoped.class)` is fired when the invocation context is initialized and an event with qualifier `@Destroyed(SipInvocationScoped.class)` is fired when the invocation context is destroyed. The payload of the event will be the instance of SIP Servlet, SIP Servlet POJO, SIP Listener, Contextual Task or Contextual Object depending on where the invocation context is active.

22.6 Enterprise Java Beans Packaged in a SIP application

EJB specification require that an enterprise java bean class with a component defining annotation defines an enterprise bean component when packaged within WEB-INF/classes directory or in a .jar file within WEB-INF/lib of a web application. An enterprise bean can also be defined via WEB-INF/ejb-jar.xml. This specification extends that requirement to SIP and converged applications that are packaged as .sar or .war applications. Related requirements specified in EJB specification on class visibility, class loading, component environment etc. are also applicable to SIP and converged applications. Refer to *EJB specification* for more details on the requirements.

22.7 Application Component Environment Namespaces

The application component's naming environment is composed of four logical namespaces, representing naming environments with different scopes. The four namespaces are

- `java:comp` namespace, for use by a single component;
- `java:module` namespace, for use by all components in a module;
- `java:app` namespace, for use by all components in an application;
- `java:global` namespace, for use by all applications.

All these four namespaces are required to be supported by a SIP servlet container provider. For more information refer *Java EE specification*.

A Change Log

A.1 Changes since v1.1

A.1.1 Changes in the v2.0 Proposed Final Draft

The following shows the changes made to this specification since the v2.0 public draft, in the order of appearance in the specification:

1. Edited the specification for language correctness such as grammar.
2. Updated Chapter 4, "SIP Servlet POJOs" with examples and clarifications.
3. Corrected section 5.1.3 The Contact Header Field and section 6.4.2 System Headers to allow modification of display name.
4. Updated section 6.5 Transport Level Information to clarify the behavior of locally generated messages.
5. Updated section 8.2.3.2 Derived SipSessions to remove the `ForkingContext.setAttributeCloningEnabled` method. Updated section 8.2.5 Dialog Termination to clarify `ForkingContext.terminateDialogs()` method's behavior.
6. Updated section 12.2.3 Sending Responses to clarify that virtual branches apply only to initial requests.
7. Clarified container behavior when CANCEL handling is turned off. See section 13.2.3 Receiving CANCEL.

8. Recommended new APIs defined in SIP Servlet 2.0 to implement B2BUAs instead of `B2BUAHelper` class. See section 15.2 B2BUA Helper Class.

9. Added a new execution context property named `javax.servlet.sip.ApplicationSession.create` and added constants to `SipServlet` class for all execution context properties. Added clarification about sharing objects between tasks. See section 17.3.2.2 Specifying Application Session Programmatically.

10. Corrected section 18.6 Transport Information to apply the behavior of `getRemoteAddr` and `getRemotePort` to requests and responses.

11. Updated Chapter 22, "Java Enterprise Edition Container" and related content all over the specification to support Java EE 6 and above.

12. Removed `support-locally-generated` flag from section C.1 The DAR Configuration File.

13. Added `internally-routed` and `routing-directive` to section D.1 Object Model.

A.1.2 Changes in the v2.0 Public Draft

The following shows the changes made to this specification since the v2.0 early draft, in the order of appearance in the specification:

1. Updated RFC support. RFC 3261 was updated by new RFCs such as 6026and 6665. These updates are made throughout the specification, including the Preface chapter.

2. Added a list of unsupported methods in `ServletContext`.See section 3.8 Unsupported ServletContext methods.

3. Added ability to resolve telephone numbers to SipURI.See section 5.2.3 Resolving Telephone Numbers to SipURI.

4. Added methods that return Java Collections in addition to the Iterators provided in the previous versions of the specification.

5. Relaxed the rules to modify system headers in section 6.4.2 System Headers.

6. Added a method in the `SipServletRequest` object to decide if the request-uri is addressed internally to the container. See section 6.5 Transport Level Information.

7. Added a method in the `SipServletMessage` object to decide whether the message originated from the container. See section 6.5 Transport Level Information.

8. Changed 100rel extension support (Reliable Provisional Responses) to mandatory. See section 6.8.1 Reliable Provisional Responses.

9. Added Attribute Stores.Attribute Stores are a general-purpose parent interface to all of the interfaces in the SIP Servlet standard that allows applications to set attributes. See chapter 7 Attributes.
10. Enhanced the SIP Session state machine to handle derived sessions. See Section 8.2.1 Relationship to SIP Dialogs.
11. Changed the SUBSCRIBE dialog establishment as per RFC 6665.See- section 8.2.3.1 Extensions Creating Dialogs.
12. Added better support for Forking, especially for derived session management.See- section 8.2.6 Forking and SipSessions.
13. Disabled the cloning of session attributes when a derived session is created. It can be re-enabled by the application. Section 8.2.3.2 Derived SipSessions.
14. Clarified that the dialog terminates only when all of the usages in the dialog are terminated. See section 8.2.4.1.2 Invalidate When Ready Mechanism.
15. Added SIP Session Keep-Alive support as per RFC 6048. See section 8.2.7 Session Keep Alive.
16. Added the ability to access Pending Messages from Session. See section 8.4 Accessing Messages from Session.
17. Added the `InviteBranch` interface to encapsulate a branch of transaction on a forked INVITE request. See section 8.5 InviteBranch.
18. Added rules for classloading and visibility of SIP Servlet classes in section 9.10 Servlet Application Classloader.
19. Added the proxy branch timeout callback to section 12.2.1 Proxy Branches.
20. Clarified handling of 2XX INVITE responses to a proxy asper RFC 6026. See section 12.2.4.1 Handling 2xx Responses to INVITE.
21. Added support for Max-Breadth as per RFC 5393. See section 12.2.11 Max-Breadth Check.
22. Added ability to write B2BUAs without using the B2BUAHelper class. This enables cross-protocol B2Bs. See section 15.1 What is a B2BUA.
23. Modified rules for detecting a SIP archive based on a SIP Servlet-specific annotation in the archive. See section 16.1.1 SIP and Java EE Converged Application.
24. Enhanced the ways to access SIP application sessions by using index keys.See section 16.3 Accessing SipApplicationSession.

25. Added Association of an `HttpSession` with a `SipApplicationSession` by using the `@SipApplicationKey` annotation. See section 16.5 Association of HTTP Session With SipApplicationSession.

26. Added the ability to retrieve outbound interfaces as `InetAddress` objects. See section 17.2 Multi-homed Host Support.

27. Added the ability to access active application session.See section 17.3.2.8 Accessing active application session.

28. Added the ROUTE_FINAL route modifier in the application composition process.Seesection 18.3 Application Router Behavior and section 18.4.1 Procedure for Routing an Initial Request.

29. Relaxed the restriction that Application Router cannot return internal routes when the route modifier is ROUTE_BACK. See section 18.3 Application Router Behavior.

30. Added that Application Router must not modify stateinfo after it is returned to the container. See section 18.3 Application Router Behavior.

31. Clarified that the container does not respond with 481 if the dialog corresponding to the Join/Replace header is not found. See section 18.10.4 Join and Replaces Targeting Mechanism.

32. Added the servlet mapping rules from SIP Servlet 1.0 to the specification. See section 19.2.1 Servlet Mapping Rules.

33. Specified rules for the deployment descriptor overriding annotations in section 21.4 Rules for Processing the Deployment Descriptor.

34. Updated the SIP Servlet deployment descriptor schema to sip-app_2_0.xsd. See section 21.5 The SIP Servlet XSD.

35. Added support for Java 7. See section 22.2 Java SE.

36. Removed the requirement for specifying the application name for a `@SipServlet` annotation. See section22.3.2 @SipServlet Annotation.

37. Added the ability to specify the `@SipApplication` annotation in classes apart from package-info.java. See section 22.3.3 @SipApplication Annotation.

38. Added the ability to disable the container's CANCEL handlingr.See section 22.3.3 @SipApplication Annotation.

39. Added the ability to specify the application version in the `@SipApplication` annotation. See section 22.3.3 @SipApplication Annotation.

40. Added the ability to specify the authentication method for the application by using annotations. See section 22.3.3.1 @SipLogin Annotation.
41. Added the ability to specify security constraints by using annotations. See section 22.3.10 Security Constraint Annotations.
42. Defined enhanced default application router configuration rules using JSON and SIP request object model. See section C.1 The DAR Configuration File.
43. Defined the SIP request object model in Appendix D SIP Request Object Model.

A.1.3 Changes in the v2.0 Early Draft

The following shows the changes made to this specification since the v1.1, in the order of appearance in the specification:

1. Defined the `SipServletContext` interface to retrieve SIP-specific container interation (e.g, getting an instance of `SipFactory` and getting an Outbound interface list).See section 3.1 SipServletContext.
2. Added the ability to programmatically add or configure servlets and configure Sip Application. See section 3.7 Programmatically adding and configuring SIP Servlets.
3. Added SIP Servlet POJOs. Added an extensible POJO model for writing SIP Servlets without depending on `GenericServlet`. See Chapter 4, "SIP Servlet POJOs".
4. Added the ability to terminate dialogs in a SIP session by sending appropriate SIP messages. See section 8.2.5 Dialog Termination.
5. Added the ability to use Java EE application names as SIP application names. It also supports looking up the application name and module name.See section 9.6 Application Names.
6. Added support for WebSockets as a transport as SIP Servlets. It also simplifies integrated converged SIP/WebSocket and Web application development. See Chapter 14, "SIP Over WebSockets".
7. Added the ability to develop portable thread safe SIP Servlet applications.See section 17.3 SIP Servlet Concurrency.
8. Added support for SIP outbound RFC (5626). Defined Flow and related interfaces to support connection management. See section 17.4 Managing Client Initiated Connections.
9. Made `applicationName` in the `@SipApplication` annotation optional to support Java EE module names. See section 22.3.3 @SipApplication Annotation.

10. Added Context and Dependency Injection support for SIP Servlet applications. Includes support for built-in beans, dependency injection, portable extension, and event framework. See section 22.5 Context and Dependency Injection (CDI).

11. Added the ability to package Enterprise Java Beans in a SIP Servlet application.See section 22.6 Enterprise Java Beans Packaged in a SIP application.

12. Added Portable JNDI name space support.See section 22.7 Application Component Environment Namespaces.

A.2 Changes since v1.0

Changes in this specification since v1.0 in the order of appearance in the specification -

1. Servlet Life Cycle Listener - Added to remove any possibility of race conditions during initialization of the servlet. Also defined how servlet initialization relate to application deployment. 2.1.1 Servlet Life Cycle Listener

2. Clear definition of Initial Request under various condition. Appendix B Definition of Initial Request

3. `Parameterable` interface - Support for SIP headers that can have parameters by providing easy accessors and mutators. 5.1.1 The Parameterable Interface

4. Ability to add Contact header parameters and set user part. 5.1.3 The Contact Header Field

5. Change to committed state of `SipServletMessage`. 6.2 Implicit Transaction State

6. Because of evolution of servlet spec 2.4, `SipServletResponse.setCharacterEncoding()` which extends `SipServletMessage` does not throw the `UnsupportedEncodingException`.

7. Specification for container to remove its own Route header and making the popped Route header available to applications. 6.7.3 Popped Route Header

8. Changes in lifetime and accessibility of `SipServletMessage`. 6.9 Accessibility of SIP Servlet Messages

9. The behavior of `SipServletMessage.getAcceptLanguage()` and `SipServletMessage.getAcceptLanguages()` has changed when no preferred locale is specified by the client. This is so that a caller of these methods can distinguish the case where no preferred locale is specified by the client from one where a preferred locale is specified. If no preferred locale is specified by the client, `getAcceptLanguage` returns null and `getAcceptLanguages` returns an empty `Iterator`. 6.11.1 Indicating Preferred Language

10. New `getSipSession(Id)` method on `SipApplicationSession` to access the `SipSession` by its Id. 8.1.1 Protocol Sessions
11. Addition of a new mechanism for `SipApplicationSession` invalidation called Invalidate When Ready mechanism. 8.1.2.2.2 Invalidate When Ready Mechanism
12. Addition of a new mechanism for `SipSession` invalidation called Invalidate When Ready mechanism. 8.2.4.1.2 Invalidate When Ready Mechanism
13. Distributed containers to support not only `Serializable` objects but also instances of `SipServletMessage` as attributes. Also extent of serializable closure clarified. 8.6.3 Distributed Environments.
14. Equivalence of .sar and .war archive formats for SIP Servlet Containers. 9.7 Servlet Application Archive File
15. New listeners for `SipApplicationSession` attributes. 10.1 SIP Servlet Event Types and Listener Interfaces
16. New listener for `SipApplicationSession` activation. 10.1 SIP Servlet Event Types and Listener Interfaces
17. New listener for `SipServlet` lifetime to receive initialization callback `servletInitialized()`. 10.1 SIP Servlet Event Types and Listener Interfaces
18. `SipSession` attribute listeners can be any class defined in the deployment descriptor. 10.1 SIP Servlet Event Types and Listener Interfaces
19. Support for RFC3327 (Path header) by extending the `Proxy` object. 12.5 Path Header and Path Parameters
20. Deprecation of transaction stateless proxy. 12.1 Parameters
21. Support for explicit `Proxy` branch creation and manipulation. 12.2.1 Proxy Branches
22. A general purpose `proxyTimeout` parameter applicable for both sequential and parallel proxy, deprecation of `sequential-search-timeout` parameter (not feature), optional override of `proxyTimeout` values for branches. 12.1 Parameters
23. Change in Sequential search timeout behavior. 12.2.4.3 Sequential Search Timeout
24. Clarification that a 503 error response to be sent to the servlet if the sending of the message fails after the `send()` returns. 13.1.4 Sending a Request as a UAC
25. Specification of Transaction timeout handling. 13.1.6 Transaction Timeout

26. Removal of the stateless record routing section. Record routing applications by very nature are stateful.
27. Support for creating PRACK request by providing `createPrack()` method. 13.1.8 Sending PRACK and 6.8.1 Reliable Provisional Responses
28. Introduction of a B2BUA helper functionality in specification. The helper simplifies the writing of B2BUA applications by abstracting features like Session linkage, Request access on Sessions, Session cloning etc. 15 Back To Back User Agents
29. SIP/HTTP and in general SIP/Java EE convergence. The features introduced in this specification bridge the gap between SIP Servlet applications and other Java EE components. 16 Converged Container and Applications
30. A clear definition of a converged application scope and features available within that scope. 16.1 Converged Application
31. Support for multihomed hosts. 17.2 Multi-homed Host Support
32. Detailed specification of Application selection and composition model. 18 Application Selection And Composition Model
33. Specification of `encodeURI` mechanism in context of application composition. 18.10.1 Session Targeting and Application Selection
34. The new Session Key based session targeting mechanism. 18.10.2 Session Key Based Targeting Mechanism
35. Introduction of a designated "main" servlet in the application. 19.2 Servlet Selection
36. Specification that only one servlet to be invoked in case of multiple servlets in mapping element. 19.2 Servlet Selection
37. Modified Servlet triggering mechanism obviating the mapping rules. 19.2 Servlet Selection
38. Mechanism for authentication of application initiated request. 20.10 Authentication of Servlet Initiated Requests
39. Java 5 support. 22.2 Java SE
40. Annotations for SIP Servlet container. 22.3 Annotations and Resource Injection
41. New method for obtaining the application name from a `SipApplicationSession.` 9.6 Application Names

A.2.1 Backward Compatibility considerations

Although applications written to v1.0 of this specification (JSR 116) shall work without any change, container implementors must be aware of a few changes. In extreme cases, applications may require some changes.

1. v1.0 style applications deployed on v1.1 or above containers must still be invoked through Application Router. Some v1.0 container implementations may have relied on servlet mapping rules to select applications. While servlet mapping rules shall still be consulted for servlet selection, application selection resides solely with Application Router.
2. The contact header now allows applications to set user-part and to add parameters. Container implementations that used user-part or some named parameters should make appropriate adjustments.
3. The Invalidate When Ready mechanism for session invalidation is enabled by default and impacts the session lifetime as compared to v1.0. See 8.2.4.2 Important Semantics.
4. `SipApplicationSession.setExpires()` no longer throws `IllegalArgumentException` on 0 or negative arguments. Instead, it sets the session to never expire, just as it sets through the deployment descriptor.
5. For v1.0 style applications with multiple servlets and their servlet-mapping rules, one and only servlet selected [first matched going through the mapping] shall be invoked on initial request.
6. Because of evolution of servlet spec 2.4, `SipServletResponse.setCharacterEncoding()`, which extends both `SipServletMessage` and `ServletResponse`, no longer throws `UnsupportedEncodingException` because it inherits a more generic method from `ServletResponse`. This is a binary compatible change, meaning that the compiled applications shall run on the v1.1 containers without change but it is not a source compatible change if the application is explicitly catching the `UnsupportedEncodingException` on `SipServletResponse.setCharacterEncoding()`.
7. In absence of the `SipServletRequest.getPoppedRoute()` API, some implementations of JSR 116 popped the Route header after the request visited the servlet. In these implementations, the applications may still have access to the Route header through existing methods such as getHeader("Route") . The applications using the old mechanism must change to use the new `SipServletRequest.getPoppedRoute()` and `SipServletRequest.getInitialPoppedRoute()` API.

A.2.2 Changes in the API since v1.0

Note: New APIs are clearly marked in API docs with "@since 1.1"

- Overall, the API was updated to use Java 5. Where possible, the API was made generic and more type safe.
- Unchecked exceptions thrown from the API are clearly specified.
- All methods updated to indicate behavior on different/invalid inputs based on underlying domain objects
- `equals()` and `toString()` in various classes are updated to be inline with the domain object behavior

Table 22-6 New classesfor application router support

`SipApplicationRouter`	application router interface
`SipApplicationRouterInfo`	application routing information
`SipApplicationRoutingDirective`	application routing directive
`SipApplicationRoutingRegion`	application routing region
`SipApplicationRoutingRegionType`	enum for routing region types
`SipRouteModifier`	enum used to influence the route value

Table 22-7 New classes for application convergence

`SipSessionsUtil`	session management for converged applications
`ConvergedHttpSession`	extension to `HttpSession` for converged applications

Table 22-8 Miscellaneous new classes

`B2buaHelper`	support for B2BUA applications
`Parameterable`	represents SIP header field values with parameters
`ProxyBranch`	proxy branch information
`SipApplicationSessionAttributeListener`	get notifications of changes to the attribute lists of application sessions

Table 22-8 Miscellaneous new classes

`SipApplicationSessionActivationListener`	new listener to support application session activation and passivation events
`SipApplicationSessionBindingEvent`	event for attribute binding/unbinding on `SipApplicationSession`
`SipApplicationSessionBindingListener`	listener for binding events
`SipServletContextEvent`	SIP Servlet-specific context event
`SipServletListener`	listener for post `init()` event
`SipServletMessage.HeaderForm`	enum for header forms
`SipSession.State`	enum for session states
`UAMode`	enum for different UA modes: UAC or UAS

Table 22-9 Changed or enhanced classes

`Address`	added that it now implements `Parameterable`
`Proxy`	added new support for creating proxy branches
	added new support for setting outbound interfaces
	added new Path header support
	added Proxy timeout
	deprecated methods `setStateful` and `getStateful`
	deprecated the sequential search attribute
`URI`	added new methods for setting and getting URI parameters
`SipURI`	moved parameter-related methods to `URI`
`TelURL`	added support for RFC 3966;also added new `setPhoneNumber` method

Table 22-9 Changed or enhanced classes

`SipSession`	added new support for `B2buaHelper` class
	added new `isValid` method
	added a method to query session state
	added new methods that are required for invalidation when ready mechanism
	added new method to support outbound interfaces
	added new method to support application composition
`SipServletMessage`	added new methods to support `Parameterable` header types
	added ability to specify default use of compact or long header names in message
	added `pushRoute(Address uri)`
	`added that getAcceptLanguage` now returns null and `getAcceptLanguages` now returns an empty `Iterator` if no preferred locale was specified by the UA
`SipServletRequest`	added support for new `B2buaHelper` class
	added new method to retrieve previously popped route header
	added new methods to support new application composition
	added new method to support the Path header
`SipServletResponse`	added new `createPrack` method
	added new predefined response code constants
`SipServlet`	added new OUTBOUND_INTERFACES and SESSIONS context attributes
	added new methods: • `doPrack` - for SIP PRACK requests • `doUpdate` - for SIP UPDATE requests • `doRefer` - for SIP REFER requests • `doPublish` - for SIP PUBLISH requests
`SipFactory`	added new method to parse `Parameterable` header types

Table 22-9 Changed or enhanced classes

`SipApplicationSession`	added new `isValid` method
	added new `encodeURL` method to support converged applications
	added new `getExpirationTime()`
	`setExpires()` now accepts 0 or negative arguments
	`added that getSessions(protocol)` now throws `IllegalArgumentException` if the protocol is not understood
	added `getSipSession(id)` and `getSession(String id, String protocol)`
	deprecated the `encodeURI()` method
	added new methods that are required for invalidation when ready mechanism
`ServletParseException`	added support for nested exceptions
`TooManyHopsException`	added support for nested exceptions

B Definition of Initial Request

Since the determination that a request is an initial request determines when the application selection process starts, a well defined procedure for making this determination is necessary. This appendix provides a precise specification for an initial request.

Compliant containers use the following procedure to determine whether a request is an initial request:

1. Request Detection. Upon receipt of a SIP message, determine whether the message is a SIP request. If it is not a SIP request, the procedure stops. The message is not an initial request.
2. Ongoing Transaction Detection. Employ methods of Section 17.2.3 in RFC 3261 to determine whether the request matches an existing transaction. If it does, the procedure stops. The request is not an initial request.
3. Examine Request Method. If it is a CANCEL, BYE, PRACK, ACK, UPDATE, or INFO request, the procedure stops. The request is not an initial request.
4. Existing Dialog Detection. If the request has a tag in the To header field, the container computes the dialog identifier (as specified in section 12 of RFC 3261) corresponding to the request and compares it with existing dialogs. If it matches an existing dialog, the procedure stops. The request is not an initial request.

   The request is a subsequent request and must be routed to the application path associated with the existing dialog. If the request has a tag in the To header field, but the dialog identifier does not match any existing dialogs, the container must reject the request with a 481 (Call/Transaction Does Not Exist). When this occurs, RFC 3261 says either the UAS has crashed or the request was routed incorrectly. In the latter case, the mis-routed request is best handled by rejecting the request. For the SIP Servlet environment, a UAS crash may mean either an application crashed or the container itself crashed. In either case, it is impossible to route the request as a subsequent request and it is inappropriate to route it as an initial request. Therefore, the only viable approach is to reject the request.

5. Detection of Requests Sent to Encoded URIs. Requests may be sent to a container instance addressed to a URI by calling the `encodeURI()` method of `SipApplicationSession` managed by this container instance. When a container receives this request, the procedure stops. This request is not an initial request. Refer to section 18.10.1 Session Targeting and Application Selection for more information about how a request sent to an encoded URI is handled by the container.

6. Initial Request. The request is an initial request, and the application invocation process starts. In this case and this case only, `SipServletRequest.isInitial()` MUST return true.

B.1 Retried Requests

The preceding procedure intentionally treats "retried" requests (as specified in Section 8.1.3.5 of RFC 3261) as initial requests. The fact that the CSeq value in requests do not match the CSeq value in the original request ensures (even in the presence of race conditions where the original transaction is not yet gone) that the request would not be detected as a merged request. There are three good reasons for this approach:

1. Inconsistent Treatment of Retried Requests. Section 8.1.3.5 of RFC 3261 merely specifies that the UAC SHOULD use the same values for Call-ID, To, and From in the re-tried request. An UA could be non-compliant with RFC 3261 and use new values for Call-ID, To, and From headers. In this case, the request is determined to be an initial request, and the application(s) that received the first request do not receive the retried request as a subsequent request.

2. Incorrect Application Handling of Retried Requests. Some retried requests for 4XX responses in 8.1.3.5 could result in a set of applications being invoked that is different from the set invoked for the first request. Examples of 4xx responses where this is possible are: 415 (Unsupported Media Type) response, 416 (Unsupported URI Scheme), and 420 (Bad Extension). Treating a retried request as a subsequent request for these 4xx response codes would thus route the request to an application or servlet where either the application router would not have chosen the application and/or the triggering rule for the servlet does not match the retried request. This is undesirable.

3. Untractable Container Implementations. The determination of such retried requests requires that From tags, Call-IDs, and CSeq values for terminated transactions be nonetheless "remembered" for some period of time. The server transaction associated with a first request eventually reaches the Terminated state because the 4xx final response is sent to the UAC and that response is subsequently acknowledged. Normally, when a transaction reaches the Terminated state, the container implementation would be free to release the resources associated with keeping track of the transaction. In fact, RFC 3261 dictates that the transaction be destroyed immediately upon the Terminated state being reached. However, because there is no specified time limit on how long a

> UAC may wait before sending a "retried" request, the container must keep terminated transactions. Although a time limit is configurable, a retried request received after the time limit expires would be treated as an initial request. In any case, keeping terminated transactions around would impose a significant burden on container implementations.

The container treats retried requests as initial requests. Unfortunately, this means that some applications are burdened with having to maintain the state required to detect such requests. This seems particularly true for applications that return the 401 (Unauthorized) or 407 (Proxy Authentication Required) expecting a retried request with the proper credentials included.

B.2 REGISTER Requests

REGISTER requests are also treated as initial requests, even when a REGISTER request refreshes an existing registration binding.

C Default Application Router

An Application Router component is essential for the functioning of the container.so the following application router logic SHOULD be available with every container compliant with this specification. The container implementations MAY choose to provide a much richer Application Router component. For the purpose of this discussion, the Application Router defined in this appendix is called the Default Application Router (DAR).

The Application Router and the Container have a simple contract defined by the `SipApplicationRouter` interface.

The DAR MUST implement all of the methods of that interface as described in this document.

C.1 The DAR Configuration File

The DAR uses a simple configuration text file that follows JSON-based application composition rules.

- The configuration text file MUST be made available to the DAR, Further, the location/content of this file MUST be accessible from a hierarchical URI, which itself is supplied as system property "`javax.servlet.sip.ar.dar.configuration`".
- The configuration file consists of an array of chains, each containing a criteria followed by an array of applications that form the application chain in JSON form. The criteria is expressed using a JSON encoding of rule language specified in Appending D SIP Request Object Model.

```
{
  "chains" : [
    {
     "description" : "Foo Application Chain",
     "criteria" : {
       "and" :{
          "equal" : {
             "request.method" : "INVITE"
          },
          "contains" : {
             "ignore-case :"true"
             "request.from.uri.host" : "oracle.com"
          }
       }
     },
     "applications" : [
       { "name" : "OriginatingCallWaiting",
         "subscriber": "request.from",
         "region": "ORIGINATING",
         "route-modifier": "NO_ROUTE"},
       {"name" : "CallForwarding",
         "subscriber":"request.to",
         "region":"TERMINATING",
         "route-modifier":"NO_ROUTE"}
     ]
  } ]
}
```

The above example shows a JSON configuration. See below for another example of a criteria element (other elements are left out for clarity).

```
  "criteria" : {
    "and" :[
       {"equal" : {
          "request.method" : "INVITE" }},
       {"equal" : {
          "request.to.uri.port" : 5060 }},
       {"contains" : {
          "ignore-case :"true",
          "request.from.uri.host" : "oracle.com"
```

```
        }}
      ]
    }
```

- More detailed explanations of each JSON element are shown below.
  - **chains**: Array of application chains. Each element in the array represents one application chain.
  - **description**: Description of the application chain.
  - **criteria**: The criteria contains JSON encoded rules that are based on the object model specified in Appendix D SIP Request Object Model. When this predicate evaluates to true, the chain is chosen for handling the request.
  - **applications**: An array of applications in this chain. This element contains the data for populating the `SipApplicationRouterInfo` object. Each application contains the following elements:
    - **name**: Name of the application as known to the container.
    - **subscriber**: The SIP header which forms the identity of the subscriber that DAR returns. This is specified according to the object model in Appendix D SIP Request Object Model. Alternatively, it can return any string.
    - **region**: The routing regionthat can be one of the strings "ORIGINATING", "TERMINATING", or "NEUTRAL"
    - **routes**: An array of SIP URIs indicating the routes as returned by the Application Router. It can be an empty string.
    - **route-modifier**: A route modifier that can be one of the strings "ROUTE", “ROUTE_FINAL”, "ROUTE_BACK", or "NO_ROUTE"
- The properties file is first read by the DAR when the `init()` is first called on the DAR. The arguments passed in the `init()` are ignored.
- The properties file is refreshed each time `applicationDeployed()` or `applicationUndeployed()` is called. Similar to `init()`, the argument of these two invocations are ignored. The callbacks act just as a trigger to read the refreshed file.

In this example, the DAR invokes two applications on an INVITE request with the host part of the From header as “oracle.com”. The applications are identified by their names as defined in the application deployment descriptors. The subscriber identity returned is the URI from the From and To header respectively for the two applications. The DAR does not return any route to the container and maintains the invocation state in the stateInfo as the index of the last application in the list.

C.2 The DAR Operation

The following method is the key interaction point between the Container and the Application Router:

```
SipApplicationRouterInfo getNextApplication
    (SipServletRequest initialRequest,
      SipApplicationRoutingRegion region,
      SipApplicationRoutingDirective directive,
       SipTargetedRequestInfo targetedRequestInfo,
      Serializable stateInfo);
```

This method is invoked when an initial request is received by the container. When this method is invoked on DAR, it uses the initial request content to evaluate the criteria of the chains in the order in which the chains are specified. It chooses the chain that evaluates the criteria to true. Next, it creates the object `SipApplicationRouterInfo` from the `applications` element of the selected chain starting from the first in the array. The `stateInfo` could contain the index of the last application returned, so that on next invocation of `getNextApplication` the DAR proceeds to the next application in the array. The order of declaration of applications becomes the priority order of invocation.

This is a minimalist Application Router with no processing logic besides the declaration of the application order. It is expected that in real world deployments, the Application Router shall play an extremely important role in application orchestration and composition. It is likely to make use of complex rules and diverse data repositories. The DAR is intended to be a very simple implementation that is available as part of the reference implementation, and could be used instead of a real world Application Router.

C.3 Backward Compatibility

The default application router specified in the 1.1 version of this specification used a simple string format in a Java properties file for configuring application chains. SIP Servlet implementations are required to support that properties file based configuration for maintaining backward compatibility. However, application developers are encouraged to use the configuration model explained in this section of the specification.

D SIP Request Object Model

For purposes of the rule language used in this specification, an object model is defined for SIP requests. This model defines a number of object types along with their properties. An object property is either undefined, another object, or a primitive value like a string or an integer. The model used here closely follows the interface definitions of the API itself.

D.1 Object Model

The model of the SIP Servlet API rule language consists of the following object types and associated properties:

SipServletRequest (request):

Represents the request itself. It is the entry point for rule matching. It has the following properties:

- **method**: the request method, a string
- **uri**: the request URI; for example a `SipURI` or a `TelURL`
- **from**: an Address representing the value of the From header
- **to**: an Address representing the value of the To header
- **top-route**: an Address representing the top Route header, if any
- **internally-routed**: A boolean ("true" or "false") equivalent to `SipServletMessage.isInternallyRouted()`

- **routing-directive:** A string representing `SipApplicationRoutingDirective` associated with the request. It is of NEW, CONTINUE or REVERSE.

URI:

- **scheme**: the URI scheme

SipURI (extends URI):

- **scheme**: a literal string – either “sip” or “sips”
- **user**: the “user” part of the SIP/SIPS URI
- **host**: the “host” part of the SIP/SIPS URI. This may be a domain name or a dotted decimal IP address
- **port**: the URI port number in decimal format; if absent, the default value is used (5060 for UDP and TCP, 5061 for TLS)
- **tel**: if the “user” parameter is not “phone”, this variable is undefined. Otherwise, its value is the telephone number contained in the “user” part of the SIP/SIPS URI with visual separators stripped. This variable is always matched case insensitively (the telephone numbers may contain the symbols ‘A’, ‘B’, ‘C’, and ‘D’).
- **param.*name***: value of the named parameter within a SIP/SIPS URI; *name* must be a valid SIP/SIPS URI parameter name.

TelURL (extends URI):

- **scheme**: always the literal string “tel”
- **tel**: the tel URL subscriber name with visual separators stripped off
- **param.*name***: value of the named parameter within a tel URL; *name* must be a valid tel URL parameter name.

Address:

- **uri** the URI object; see `URI`, `SipURI`, `TelURL` types above
- **display-name**: the display-name portion of the From or To header

Unless otherwise noted, variable values are case sensitive, meaning that the operators listed below by default take case into account when matching on those variables

D.1.1 An Example

TABLE 1 shows the value of all defined variables for the following SIP requests:

```
INVITE sip:watson@boston.bell-tel.com SIP/2.0
From: A. Bell <sip:a.g.bell@bell-tel.com>;tag=3pcc
To: T. Watson <sip:watson@bell-tel.com>....
```

TABLE 1. Variables and Corresponding Values

variable	value
request.method	“INVITE”
request.uri	“sip:watson@boston.bell-tel.com”
request.uri.scheme	“sip”
request.uri.user	“watson”
request.uri.host	“boston.bell-tel.com”
request.uri.port	5060
request.uri.tel	*undefined*
request.from	“A. Bell <sip:a.g.bell@bell-tel.com>;tag=3pcc”
request.from.uri	“sip:a.g.bell@bell-tel.com”
request.from.uri.scheme	“sip”
request.from.uri.user	“a.g.bell”
request.from.uri.host	“bell-tel.com”
request.from.uri.port	5060
request.from.display-name	“A. Bell”
request.to	“T. Watson <sip:watson@bell-tel.com>“
request.to.uri	“sip:watson@bell-tel.com“
request.to.uri.scheme	“sip“

variable	value
request.to.uri.user	“watson“
request.to.uri.host	“bell-tel.com“
request.to.uri.port	5060
request.to.display-name	“T. Watson“
request.top-route	*undefined*

D.2 Conditions

A condition is either an operator or a logical connector. The following operators are defined:

- **equal**: compares the value of a variable with a literal value and evaluates to true if the variable is defined and its value equals that of the literal. Otherwise, the result is false.
- **exists**: takes a variable name and evaluates to true if the variable is defined, and false otherwise.
- **contains**: evaluates to true if the value of the variable specified as the first argument contains the literal string specified as the second argument.
- **subdomain-of**: given a variable denoting a domain name (SIP/SIPS URI *host*) or telephone subscriber (*tel* property of SIP or Tel URLs) and a literal value, this operator returns true if the variable denotes a subdomain of the domain given by the literal value. Domain names are matched according to the DNS definition of what constitutes a subdomain; for example, the domain names "example.com" and “research.example.com“ are both subdomains of "example.com". IP addresses may be given as arguments to this operator; however, they only match exactly. In the case of the *tel* variables, the subdomain-of operator evaluates to true if the telephone number denoted by the first argument has a prefix that matches the literal value given in the second argument; for example, the telephone number "1 212 555 1212" would be considered a subdomain of "1212555".

Additionally, the following logical connectors are supported:

- **and:** contains a number of conditions and evaluates to true if and only if all contained conditions evaluate to true
- **or:** contains a number of conditions and evaluates to true if and only if at least one contained condition evaluates to true
- **not:** negates the value of the contained condition.

The *equal* and *contains* operators optionally ignore character case when making comparisons. The default is case-sensitive matching.

E References

[3pcc]	J. Rosenberg, J. Peterson, H. Schulzrinne and G. Camarillo, RFC 3725 - Best Current Practices for Third Party Call Control (3pcc) in the Session Initiation Protocol (SIP)
[CPL]	J. Lennox, X. Wu and H. Schulzrinne, RFC 3380 - CPL: A Language for User Control of Internet Telephony Services
[IM]	J. Rosenberg et al., RFC 3428 - Session Initiation Protocol (SIP) Extension for Instant Messaging.
[JAF]	B. Calder and B. Shannon, “JavaBeans Activation Framework Specification”, May 27, 1999.
[JLS]	“The Java Programming Language Specification”, http://java.sun.com/docs/books/jls.
[JavaMail]	Sun Microsystems, “JavaMail API Design Specification v1.2”, September 2000.
[JAXP]	R. Mordani, J. D. Davidson, and S. Boag, “Java API for XML Processing”, February 6, 2001.
[privacy]	C. Jennings, J. Peterson, and M. Watson, RFC 3325 - Private Extensions to the Session Initiation Protocol (SIP) for Asserted Identity within Trusted Networks.
[refer]	R. Sparks, RFC 3515 - The Session Initiation Protocol (SIP) Refer Method.
[RFC 1738]	T. Berners-Lee, L. Masinter, and M. McCahill, “Uniform Resource Locators (URL)”, RFC 1738, December 1994.
[RFC 2278]	N. Freed and J. Postel, “IANA Charset Registration Procedures”, RFC 2278, January 1998.
[RFC 2327]	M. Handley and V. Jacobson, “SDP: Session Description Protocol”, RFC 2327, April 1998.

[3pcc]	J. Rosenberg, J. Peterson, H. Schulzrinne and G. Camarillo, RFC 3725 - Best Current Practices for Third Party Call Control (3pcc) in the Session Initiation Protocol (SIP)
[RFC 2976]	S. Donovan, “The SIP INFO Method”, RFC 2976, October 2000.
[RFC 3261]	J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, “SIP: Session Initiation Protocol“, RFC 3261, June 2002.
[RFC 3262]	J. Rosenberg and H. Schulzrinne, “Reliability of Provisional Responses in the Session Initiation Protocol (SIP)”, RFC 3262, June 2002.
[RFC 3265]	A. B.Roach, “Session Initiation Protocol (SIP)-Specific Event Notification“, RFC 3265, June 2002.
[SERL]	R. W. Steenfeldt and H. Smith, “SIP Service Execution Rule Language Framework and Requirements“, Internet Engineering Task Force, May 2001. Work in progress.
[Servlet API]	D. Coward, “Java Servlet Specification, Version 2.3”, September, 2001.
[simple]	Rosenberg et al., RFC 3856 - A Presence Event Package for the Session Initiation Protocol (SIP)
[timer]	S. Donovan and J. Rosenberg, RFC 4028 - Session Timers in the Session Initiation Protocol (SIP)
[DFC1998]	"Distributed feature composition: A virtual architecture for telecommunications services" Michael Jackson and Pamela Zave. IEEE Transactions on Software Engineering XXIV(10):831-847, October 1998.
[CN2004]	"Component coordination: A telecommunication case study" Pamela Zave and Healfdene H. Goguen and Thomas M. Smith. Computer Networks XXXXV(5):645-664, August 2004.
[SE2005]	"Experience with component-based development of a telecommunication service" Gregory W. Bond and Eric Cheung and Healfdene H. Goguen and Karrie J. Hanson and Don Henderson and Gerald M. Karam and K. Hal Purdy and Thomas M. Smith and Pamela Zave. Proceedings of the Eighth International Symposium on Component-Based Software Engineering, May 2005.
[JSR250]	R. Mordani, Common Annotations for the JavaTM Platform
[RFC3327]	D. Willis and B. Hoeneisen, Session Initiation Protocol (SIP) Extension Header Field for Registering Non-Adjacent Contacts.
[RFC3966]	H. Schulzrinne, The tel URI for Telephone Numbers.

[3pcc]	J. Rosenberg, J. Peterson, H. Schulzrinne and G. Camarillo, RFC 3725 - Best Current Practices for Third Party Call Control (3pcc) in the Session Initiation Protocol (SIP)
[sipping-config-framework]	D. Petrie, A Framework for Session Initiation Protocol User Agent Profile Delivery.
[RFC 5626]	C. Jennings, R. Mahy, F. Audet , Managing Client-Initiated Connections in the Session Initiation Protocol (SIP)
[SIP-WEBSOCKET-DRAFT]	I. Baz Castillo, J. Millan Villegas, V. Pascual, The WebSocket Protocol as a Transport for the Session Initiation Protocol (SIP)
[RFC 3326]	H. Schulzrinne, D. Oran and G. Camarillo, The Reason Header Field for the Session Initiation Protocol (SIP).
[SIP Servlet API]	A. Kristensen, “SIP Servlet API, Version 1.0”, February, 2003.

F Glossary

Address-of-Record

An address-of-record (AOR) is a SIP or SIPS URI that points to a domain with a location service that can map the URI to another URI where the user might be available. Typically, the location service is populated through registrations. An AOR is frequently thought of as the “public address” of the user.

Application developer

The producer of a SIP application. Application developers create a set of servlet classes, supporting libraries, and files (such as images and compressed archive files) for the servlet application. The application developer is typically an application domain expert. The developer must be aware of the servlet environment and its consequences when programming, including concurrency considerations, and create the SIP application accordingly.

Application assembler

Takes the output of the application developer and ensures that it is a deployable unit. Thus, the input of the application assembler is the servlet classes, JSP pages, HTML pages, and other supporting libraries and files for the servlet application. The application assembler generates a servlet application archive or a servlet application in an open directory structure.

Application path

A list of application instances to invoke for incoming messages belonging to a particular dialog. This list is constructed as a side effect of the handling of the initial request. It consists of the following:

- the application acting as a UAC if the initial request was created within the container
- the application acting as an UAS if an application responded to the initial request

- all applications that proxied with record-routing enabled

The application path is a logical concept and may or may not be explicitly represented by implementations.

Application Router

Application Router is a component that is essential for the container's functioning. The contract between the container and Application Router is an interface defined in this specification. On initial requests, the container consults the Application Router to determine the application to invoke. After any application completes processing a request, the container again consults Application Router to determine the next application to invoke. Thus, Application Router determines the application invocation order.

Application session

Application sessions represent application instances. Application sessions act as a store for application data and provide access to contained protocol sessions.

Back-To-Back User Agent

A back-to-back user agent (B2BUA) is a logical entity that receives a request and processes it as an user agent server (UAS). To determine how the request should be answered, it acts as an user agent client (UAC) and generates requests. Unlike a proxy server, it maintains dialog state and must participate in all requests sent on the dialogs it established. Since it is a concatenation of an UAC and UAS, no explicit definitions are needed for its behavior.

Call

A call is an informal term that refers to communication between peers generally set up for the purposes of a multimedia conversation.

Call leg

Call leg is another name for a dialog [RFC 2543]; it is not used in the revised SIP specification [RFC 3261].

Call Stateful

A proxy is called stateful if it retains state for a dialog from the initiating INVITE request to the terminating BYE request. A call stateful proxy is always transaction stateful, but the converse is not necessarily true.

Client

A client is any network element that sends SIP requests and receives SIP responses. Clients may or may not interact directly with a human user. User agent clients and proxies are clients.

Committed message

A SIP message object that was fully processed according to this specification and that should not be further modified. See 6.2 Implicit Transaction State.

Conference

A multimedia session that contains multiple participants.

Default servlet

The first servlet listed in the deployment descriptor.

Deployer

The deployer takes one or more servlet application archive files or other directory structures provided by an application developer and deploys the application into a specific operational environment. The operational environment includes a specific servlet container and SIP server. The deployer must resolve all of the external dependencies declared by the developer. To perform his role, the deployer uses tools provided by the servlet container provider. The deployer is an expert in a specific operational environment. For example, the deployer is responsible for mapping the security roles defined by the application developer to the user groups and accounts that exist in the operational environment where the servlet application is deployed.

Dialog

A dialog is a peer-to-peer SIP relationship between two UAs that persists for some time. A dialog is established by SIP messages, such as a 2xx response to an INVITE request. A dialog is identified by a call identifier, a local tag, and a remote tag. A dialog was formerly known as a call leg in RFC 2543. The baseline SIP specification defines only INVITE–BYE as a mechanism of establishing and terminating dialogs but allows extensions to define other methods capable of initiating dialogs. The SUBSCRIBE/NOTIFY methods defined by the event framework are an example of this [RFC 3265].

Downstream

Refers to the direction that requests flow from the user agent client to user agent server.

Final response

A response that terminates a SIP transaction as opposed to a provisional response that does not. All 2xx, 3xx, 4xx, 5xx, and 6xx responses are final.

Header

A header is a component of a SIP message that conveys information about the message. It is structured as a sequence of header fields.

Header field

A header field is a component of the SIP message header. It consists of one or more header field values separated by a comma or having the same header field name.

Header field value

A header field value is a singular value, which can be one of many in a header field.

Home Domain

The domain providing service to a SIP user. Typically, this is the domain present in the URI in the address-of-record of a registration.

Informational Response

Same as a provisional response.

Initial request

A request that is dispatched to applications based on rule matching rather than on an existing application path. Compare with subsequent request.

Initiator, calling party, caller

The party initiating a session (and dialog) with an INVITE request. A caller retains this role from the time he sends the initial INVITE that established a dialog until the termination of that dialog.

Invitation

An INVITE request.

Invitee, invited user, called party, callee

The party that receives an INVITE request for the purposes of establishing a new session. A callee retains this role from the time he receives the INVITE until the termination of the dialog established by that INVITE.

Location server

See Location service.

Location service

A location service is used by a SIP redirect or proxy server to obtain information about a callee's possible location(s). It contains a list of bindings of address-of-record keys to zero or more contact addresses. The bindings can be created and removed in many ways; this specification defines a REGISTER method that updates the bindings.

Loose Routing

A proxy is said to be loose routing if it follows the procedures defined in this specification for processing the Route header field. These procedures separate the request's destination (present in the Request-URI) from the set of proxies that need to be visited along the way (present in the Route header field). A proxy compliant with these mechanisms is also known as a loose router.

Message

Data sent between SIP elements as part of the protocol. SIP messages are either requests or responses.

Method

The method is the primary function that a request invokes on a server. The method is carried in the request message itself. Example methods are INVITE and BYE.

Outbound proxy

A proxy that receives requests from a client, even though it may not be the server resolved by the Request-URI. Typically, a UA is manually configured with an outbound proxy, or can learn about one through auto-configuration protocols.

Parallel search

In a parallel search, a proxy issues several requests to possible user locations upon receiving an incoming request. Rather than issuing one request and then waiting for the final response before issuing the next request , a parallel search issues requests without waiting for the result of previous requests.

Principal

A principal is an entity that can be authenticated by an authentication protocol. A principal is identified by a principal name and authenticated by using authentication data. The content and format of the principal name and the authentication data depend on the authentication protocol.

Protocol session

A common name for protocol-specific session objects. A protocol session represents a point-to-point signaling relationship and serves as a repository for application data relating to that relationship. Examples include the `SipSession` and `HttpSession` interfaces defined by the SIP and HTTP Servlet specifications, respectively. A number of protocol sessions may belong to a single application session.

Provisional response

A response used by the server to indicate progress but does not terminate a SIP transaction. 1xx responses are provisional, and other responses are considered final. Provisional responses are not sent reliably.

Proxy, proxy server

An intermediary entity that acts as both a server and a client for the purpose of making requests on behalf of other clients. A proxy server primarily plays the role of routing, which means its job is to ensure that a request is sent to another entity "closer" to the targeted user. Proxies are also useful for enforcing policy (for example, making sure a user is allowed to make a call). A proxy interprets and, if necessary, rewrites specific parts of a request message before forwarding it.

Recursion

A client recurses on a 3xx response when it generates a new request to one or more of the URIs in the Contact header field in the response.

Redirect server

A redirect server is a user agent server that generates 3xx responses to requests it receives, directing the client to contact an alternate set of URIs.

Registrar

A registrar is a server that accepts REGISTER requests and places the information it receives in those requests into the location service for the domain it handles.

Regular transaction

A regular transaction is any transaction with a method other than INVITE, ACK, or CANCEL.

Request

A SIP message sent from a client to a server for the purpose of invoking a particular operation.

Response

A SIP message sent from a server to a client, for indicating the status of a request sent from the client to the server.

Ringback

Ringback is the signaling tone produced by the calling party's application, indicating that a called party is being alerted (ringing).

Role (development)

The actions and responsibilities taken by various parties during the development, deployment, and running of a servlet application. In some scenarios, a single party may perform several roles; in others, each role may be performed by a different party.

Role (security)

An abstract notion used by an application developer in an application that can be mapped by the Deployer to a user, or group of users, in a security policy domain.

Route set

A route set is a collection of ordered SIP or SIPS URI that represent a list of proxies that must be traversed when sending a particular request. A route set can be learned, through headers like Record-Route, or it can be configured.

Security policy domain

The scope over which security policies are defined and enforced by a security administrator of the security service. A security policy domain is also referred to as a realm.

Security technology domain

The scope over which the same security mechanism, such as Kerberos, is used to enforce a security policy. Multiple security policy domains can exist within a single technology domain.

Server

A server is a network element that receives requests to service them and sends back responses to those requests. Examples of servers are proxies, user agent servers, redirect servers, and registrars.

Sequential Search

In a sequential search, a proxy server attempts each contact address in sequence, proceeding to the next one only after the previous one generated a final response or timed out. A 2xx or 6xx class final response always terminates a sequential search.

Servlet application archive

A single file that contains all of the components of a servlet application. This archive file is created by using standard JAR tools, which allow any or all of the application components to be signed. Servlet application archive files are identified by the .sar extension. The SAR file layout is derived from the Web application archive (.war) file format but may contain servlets and deployment descriptors pertaining to different protocols, such as SIP and HTTP. With this specification, either SAR or WAR file formats can be used to package SIP Servlet applications.

Servlet Container Provider

A vendor that provides the runtime environment, namely the servlet container and possibly the SIP server, in which a servlet application runs as well as the tools necessary to deploy servlet applications. The expertise of the container provider is in HTTP-level programming. Since this specification does not specify the interface between the SIP server and the servlet container, it is left to the container provider to split the implementation of the required functionality between the container and the server.

Servlet definition

A unique name associated with a fully qualified class name of a class implementing the `Servlet` interface. A set of initialization parameters can be associated with a servlet definition.

Servlet mapping

A servlet definition that associates a servlet container with a URL path pattern. All requests to that path pattern are handled by the servlet associated with the servlet definition.

Session

From the SDP specification, “A multimedia session is a set of multimedia senders and receivers and the data streams flowing from senders to receivers. A multimedia conference is an example of a multimedia session.” [RFC 2327] (A session as defined for SDP can comprise one or more RTP sessions.) As defined, a callee can be invited several times, by different calls, to the same session. If SDP is used, a session is defined by the concatenation of the user name, session id, network type, address type, and address elements in the origin field.

SIP application

A collection of SIP Servlets and static resources that might include voice prompts, grammars, VoiceXML scripts, and other data. A SIP application may be packaged into an archive or exist in an open directory structure. All compatible servlet containers must accept a SIP application and perform a deployment of its contents into their runtime. This may mean that a container can run the application directly from a servlet application archive file, or it may mean that it moves the contents of a servlet application into the appropriate locations for the container. SIP applications are to the SIP Servlet API as web applications are to the HTTP Servlet API.

SIP transaction

A SIP transaction occurs between a client and a server. It comprises all messages from the first request sent from the client to the server up to a final (non-1xx) response sent from the server to the client. If the request is INVITE and the final response is a non-2xx, the transaction also includes an ACK to the response. The ACK for a 2xx response to an INVITE request is a separate transaction.

SIP/web application, distributable

A SIP or Web application that can be deployed in a servlet container distributed across multiple Java virtual machines running on the same host or different hosts. The deployment descriptor for such an application uses the distributable element.

Stateful Proxy

A logical entity that maintains the client and server transaction state machines defined by this specification during the processing of a request. It is also known as a transaction stateful proxy. The behavior of a stateful proxy is further defined in 12 Proxying. A (transaction) stateful proxy is not the same as a call stateful proxy.

Stateless Proxy

A logical entity that does not maintain the client or server transaction state machines defined in this specification when it processes requests. A stateless proxy forwards every request it receives downstream and every response it receives upstream. This specification deprecates the use of SIP Servlet applications as a transaction stateless proxy.

Strict routing

A proxy is said to be strict routing if it follows the Route processing rules of RFC 2543 and many prior Internet Draft versions of RFC 3261. That rule requires proxies to destroy the contents of the Request-URI when a Route header field is present. Strict routing behavior is not used in RCF 3261, in favor of a loose routing behavior. Proxies that perform strict routing are also known as strict routers.

Subsequent request

A request that is dispatched to applications.A request is a subsequent request if it uses an existing application path that was created while processing an earlier initial request establishing the dialog.

System Administrator

The person who configures and administers the servlet container. The administrator oversees the well-being of deployed applications at runtime. This specification does not define the contracts for system management and administration. The administrator typically uses runtime monitoring and management tools provided by the container provider and server vendors to accomplish these tasks.

System headers

Headers that are managed by the SIP Servlet container.SIP Servlets must not attempt to modify system headers directly via calls to `setHeader` or `addHeader`. This includes Call-ID, From, To, CSeq, Via, Record-Route, Route, and Contact headers when used to specify a session signaling address. For example, in INVITEs and 200 responses to INVITEs. System headers are discussed in section 6.4.2 System Headers.

TLS

Transport Layer Security. A layer four means of protecting TCP connections by providing integrity, confidentiality, replay protection, and authentication.

Uniform resource locator (URL)

A compact string representation of information for location and access of resources via the Internet [RFC 1738]. SIP and SIPS URIs are syntactically similar to mailto URLs. That is, they are typically of the form sip:user@host, where user is a user name or a telephone number and host is either a fully qualified domain name or a numeric IP address. SIP URIs are used within SIP messages to indicate the originator (From), current destination (Request-URI), and final recipient (To) of a SIP request, and to specify redirection addresses (Contact). When used as a hyper link (for example in a Web page), a SIP URI indicates that the specified user or service can be contacted using SIP.

Upstream

It refers to the direction that responses flow from the user agent server back to the user agent client.

URL-encoded

A character string encoded according to RFC 1738 [RFC 1738, section 2.2].

User agent client (UAC)

A user agent client is a logical entity that creates a new request and then uses the client transaction state machinery to send it. The role of the UAC lasts only for the duration of that transaction. In other words, if a piece of software initiates a request, it acts as an UAC for the duration of that transaction. If it receives a request later, it assumes the role of an user agent server for the processing of that transaction.

User agent server (UAS)

A user agent server is a logical entity that generates a response to a SIP request. The response accepts, rejects, or redirects the request. This role lasts only for the duration of that transaction. In other words, if a piece of software responds to a request, it acts as a UAS for the duration of that transaction. If it generates a request later, it assumes the role of a user agent client for the processing of that transaction.

User agent (UA)

A logical entity that can act as both a user agent client and user agent server.

Web application

A collection of servlets, JSP pages, HTML documents, and other web resources that might include image files, compressed archives, and other data. A web application may be packaged into an archive or exist in an open directory structure. All compatible servlet containers must accept a web application and perform a deployment of its contents into their runtime. This may mean that a container can run the application directly from a web application archive file, or it may mean that it moves the contents of a web application into the appropriate locations for that particular container.