SIP Servlet API
Version 1.0

Anders Kristensen (specification lead)
akristensen@dynamicsoft.com

Please send all comments to
sipservlet-interest@java.sun.com

February 4, 2003

SIP Servlet API Specification ("Specification")
Version: 1.0
Status: Final Release
Specification Lead: dynamicsoft Inc. ("Specification Lead")
Release: February 4, 2003

Copyright 2003 dynamicsoft Inc.
All rights reserved.

NOTICE
The Specification is protected by copyright and the information described therein may be protected by one or more U.S. patents, foreign patents, or pending applications. Except as provided under the following license, no part of the Specification may be reproduced in any form by any means without the prior written authorization of the Specification Lead and its licensors, if any. Any use of the Specification and the information described therein will be governed by the terms and conditions of this license and the Export Control Guidelines as set forth in the Terms of Use on the Sun website. By viewing, downloading or otherwise copying the Specification, you agree that you have read, understood, and will comply with all of the terms and conditions set forth herein.

The Specification Lead hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense), under the Specification Lead's intellectual property rights that are essential to practice the Specification, to internally practice the Specification for the purpose of designing and developing your Java applets and applications intended to run on the Java platform or creating a clean room implementation of the Specification that: (i) includes a complete implementation of the current version of the Specification, without subsetting or supersetting; (ii) implements all of the interfaces and functionality of the Specification without subsetting or supersetting; (iii) includes a complete implementation of any optional components (as defined by the Specification) which you choose to implement, without subsetting or supersetting; (iv) implements all of the interfaces and functionality of such optional components, without subsetting or supersetting; (v) does not add any additional packages, classes or interfaces to the "java.*" or "javax.*" packages or subpackages or other packages defined by the Specification; (vi) satisfies all testing requirements available from the Specification Lead relating to the most recently published version of the Specification six (6) months prior to any release of the clean room implementation or upgrade thereto; (vii) does not derive from any of the Specification Lead's source code or binary code materials; and (viii) does not include any of the Specification Lead's source code or binary code materials without an appropriate and separate license from the Specification Lead. The Specification contains the proprietary information of the Specification Lead and may only be used in accordance with the license terms set forth herein. This license will terminate immediately without notice from the Specification Lead if you fail to comply with any provision of this license. Upon termination or expiration of this license, you must cease use of or destroy the Specification.

TRADEMARKS
No right, title, or interest in or to any trademarks, service marks, or trade names of Sun or Sun's licensors, the Specification Lead or the Specification Lead's licensors is granted hereunder. Sun, Sun Microsystems, the Sun logo, Java, JAIN, and the Java Coffee Cup logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

DISCLAIMER OF WARRANTIES
THE SPECIFICATION IS PROVIDED "AS IS" AND IS EXPERIMENTAL AND MAY CONTAIN DEFECTS OR DEFICIENCIES WHICH CANNOT OR WILL NOT BE CORRECTED BY THE SPECIFICATION LEAD. THE SPECIFICATION LEAD MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER

EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE OR THAT ANY PRACTICE OR IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADE SECRETS OR OTHER RIGHTS. This document does not represent any commitment to release or implement any portion of the Specification in any product.

THE SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION THEREIN; THESE CHANGES WILL BE INCORPORATED INTO NEW VERSIONS OF THE SPECIFICATION, IF ANY. THE SPECIFICATION LEAD MAY MAKE IMPROVEMENTS AND/OR CHANGES TO THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THE SPECIFICATION AT ANY TIME. Any use of such changes in the Specification will be governed by the then-current license for the applicable version of the Specification.

LIMITATION OF LIABILITY
TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL THE SPECIFICATION LEAD OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO ANY FURNISHING, PRACTICING, MODIFYING OR ANY USE OF THE SPECIFICATION, EVEN IF THE SPECIFICATION LEAD AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will indemnify, hold harmless, and defend the Specification Lead and its licensors from any claims arising or resulting from: (i) your use of the Specification; (ii) the use or distribution of your Java application, applet and/or clean room implementation; and/or (iii) any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND
If this Software is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

REPORT
You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your evaluation of the Specification ("Feedback"). To the extent that you provide the Specification Lead with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant the Specification Lead a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.

Contents

Preface xi

Who should read this Document xi
Typographic Conventions xi
Providing Feedback xi
Acknowledgements xi

1 Overview 1

1.1 Goals of the SIP Servlet API 1
1.2 What is a SIP Servlet? 1
1.3 What is a SIP Servlet Container? 2
1.4 Relationship with the HTTP Servlet API 2
1.4.1 Differences from the HTTP Servlet API 2
1.4.2 Converged Applications 4
1.5 Examples 4
1.5.1 A Location Service 4
1.5.2 A Multi-Protocol Servlet Application 5

2 Request Routing and Application Composition 9

2.1 Initial Requests 9
2.2 Subsequent Requests 9
2.2.1 Message Context 9
2.2.2 Application Path 10
2.3 Application Path Cleanup 11
2.4 Application Composition 11
2.4.1 Cascaded Services Model 11
2.4.2 The Role of Proxying 12
2.4.3 Loop Detection 12
2.4.4 Limitations of the Model 13
2.5 An Example 13

3 The Servlet Interface 15
3.1 Servlet Life Cycle 15
3.2 Processing SIP Messages 15
3.3 SIP Specific Request Handling Methods 16
3.3.1 Receiving Requests 17
3.4 SIP Specific Response Handling Methods 18
3.5 Number of Instances 18

4 Servlet Context 19
4.1 The SipFactory 19
4.2 Extensions Supported 19
4.3 Context Path 19

5 Addressing 21
5.1 The Address Interface 21
5.1.1 The From and To Header Fields 22
5.1.2 The Contact Header Field 22
5.2 URIs 23
5.2.1 SipURI 23
5.2.2 TelURL 24

6 Requests and Responses 25
6.1 The SipServletMessage Interface 25
6.2 Implicit Transaction State 25
6.3 Access to Message Content 26
6.3.1 Character Encoding 27
6.4 Headers 27
6.4.1 Address Header Fields 28
6.4.2 System Headers 28
6.4.3 TLS Attributes 29
6.5 Transport Level Information 29
6.6 Requests 29
6.6.1 Parameters 29
6.6.2 Attributes 30
6.7 Responses 30
6.7.1 Reliable Provisional Responses 31
6.7.2 Buffering 32
6.8 Internationalization 32
6.8.1 Indicating Preferred Language 32

6.8.2 Indicating Language of Message Content ... 33

7 Acting as a User Agent ... 35

7.1 Client Functions ... 35
7.1.1 Creating Initial Requests ... 35
7.1.2 Creating Subsequent Requests ... 37
7.1.3 Pushing Route Header Field Values ... 37
7.1.4 Sending a Request as a UAC ... 37
7.1.5 Receiving Responses ... 38
7.1.6 Sending ACK ... 38
7.1.7 Sending CANCEL ... 38
7.2 Server Functions ... 39
7.2.1 Sending Responses ... 39
7.2.2 Receiving ACK ... 39
7.2.3 Receiving CANCEL ... 39
7.3 Back-To-Back User Agents ... 40

8 Proxying ... 41

8.1 Parameters ... 41
8.2 Operation ... 42
8.2.1 Pushing Route headers ... 43
8.2.2 Sending Responses ... 43
8.2.3 Receiving Responses ... 43
8.2.4 Sending CANCEL ... 45
8.2.5 Receiving CANCEL ... 45
8.2.6 Sending ACK ... 46
8.2.7 Receiving ACK ... 46
8.2.8 Handling Subsequent Requests ... 46
8.2.9 Max-Forwards Check ... 47
8.3 Proxying and Sessions ... 47
8.3.1 Stateless Record-Routing ... 47
8.4 Record-Route Parameters ... 47

9 Timer Service ... 49

9.1 TimerService ... 49
9.2 ServletTimer ... 50
9.3 TimerListener ... 50

10 Sessions 51

10.1 SipApplicationSession 51
10.1.1 Protocol Session Iterators 52
10.1.2 SipApplicationSession Lifetime 52
10.2 SipSession 53
10.2.1 Relationship to SIP Dialogs 53
10.2.2 Maintaining Dialog State in the SipSession 54
10.2.3 Creation of SipSessions 55
10.2.4 SipSession Lifetime 57
10.2.5 The RequestDispatcher Interface 57
10.2.6 The SipSession Handler 58
10.2.7 Binding Attributes into a SipSession 58
10.3 Last Accessed Times 59
10.4 Important Session Semantics 59
10.4.1 Threading Issues 59
10.4.2 Distributed Environments 59

11 Mapping Requests to Servlets 61

11.1 Triggering Rules 61
11.2 SIP Request Object Model 61
11.3 Conditions 62
11.4 XML Encoding 63
11.5 An Example 64

12 SIP Servlet Applications 67

12.1 Relationship with HTTP Servlet Applications 67
12.2 Relationship to ServletContext 67
12.3 Elements of a SIP Application 67
12.4 Deployment Hierarchies 67
12.5 Directory Structure 68
12.5.1 Example Application Directory Structure 68
12.6 Servlet Application Archive File 69
12.7 SIP Application Configuration Descriptor 69
12.8 Dependencies On Extensions 69
12.9 Servlet Application Classloader 70
12.10 Replacing a Servlet Application 70
12.11 Servlet Application Environment 70

13 Application Listeners and Events 73
13.1 SIP Servlet Event Types and Listener Interfaces 73

14 Security 75
14.1 Introduction 75
14.2 Declarative Security 75
14.3 Programmatic Security 76
14.4 Roles 77
14.5 Authentication 77
14.6 Server Tracking of Authentication Information 78
14.7 Propagation of Security Identity in EJBTM Calls 78
14.8 Specifying Security Constraints 78
14.9 Default Policies 79

15 Deployment Descriptor 81
15.1 Differences from the HTTP Servlet Deployment Descriptor 81
15.2 Converged SIP and HTTP Applications 81
15.3 Deployment Descriptor Elements 82
15.4 Rules for Processing the Deployment Descriptor 82
15.4.1 Deployment Descriptor DOCTYPE 83
15.5 DTD 83
15.6 Examples 96
15.6.1 A Basic Example 96

A References 99

B Glossary 101

Preface

This specification defines version 1.0 of the SIP Servlet API. The specification requires Java 2, version 1.2 and support for SIP as defined in [RFC 3261].

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.

Typographic Conventions

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

Providing Feedback

Feedback on this specification is welcomed. Please e-mail your comments to `sipservlet-interest@mailman.dynamicsoft.com`.

Acknowledgements

This specification was developed under the Java Community Process 2.0 as JSR 116. The JSR 116 expert group consists of the following members: Danny Coward (Sun Microsystems), Mick O'Doherty (Nortel Networks), Christophe Gourraud (Ericsson), Anders Kristensen (dynamicsoft), Mikko Lonnfors (Nokia), Wei Lu (IBM), Igor Miladinovic (Vienna University), Sean Olson (individual), Marc Petit-Huguenin (8x8), Jean-Pierre Pinal (Siemens), Johannes Stadler (Vienna University), James Steadman (Ubiquity), Kindy Sylla (Ericsson).

Additionally, Ajay Deo, Peter Mataga, Kelvin Porter, Jonathan Rosenberg, and Prasad Sripathi of dynamicsoft made numerable contributions to this specification.

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].

The definition of the *tel* properties of sip, sips, and tel URLs and the *subdomain-of* expression in the SIP Servlet API rule language were taken from CPL [CPL].

1 Overview

The Session Initiation Protocol (SIP) is used to establish, modify, and tear down multimedia IP sessions including IP telephony, presence, and instant messaging. An important aspect of any communication infrastructure is programmability and the purpose of the SIP Servlet API is to standardize the platform for delivering SIP based services. The term *platform* is used here to include 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 “non-essential” 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 media types, for example, Web, telephony, and presence.
- **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 execute 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 which is managed by a SIP servlet container and which 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 via that server's native extension API. Servlet containers can also be built into or possibly installed into 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 2 Platform Standard Edition 1.2 (J2SE) or Java 2 Platform Enterprise Edition 1.3 (J2EE) environment, these restrictions should be placed using the permission architecture defined by the Java 2 Platform. For example, high-end application servers may limit the creation of a `Thread` object, to insure that other components of the container 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 as package `javax.servlet` and an HTTP specific part in package `javax.servlet.http`. This specification refers to the generic part 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`. As such, 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 and so the two protocols have much in common. Both are request-response protocols and messages have very similar structure and formats. 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, 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 is crucial in this context.

- 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.

It follows that, in addition to the ability inherited from the Servlet API of responding to incoming requests, the SIP Servlet API must support the following capabilities:

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

1.4.1.1 Use of Servlet.service()

In order to allow for these features, the SIP Servlet API uses the original Servlet API interfaces in a manner that differs from 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

Another important difference is the fact that the SIP Servlet API event model is asynchronous rather than synchronous as in the HTTP API. This means that applications are not obliged to respond to incoming requests in the upcall reporting the event. They may initiate some other action, return control to the container, and then respond to the request at some later point in time. The container relies on timeout of the *application instance* as a whole in order 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 B2BUA not to hog threads while waiting for a potentially long-lived transaction to complete.

1.4.1.3 Application Composition

While 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 described further in the chapter 2.

1.4.2 Converged Applications

While the SIP Servlet API can certainly be implemented independently of the HTTP Servlet API, it is expected that many interesting services will combine multiple modes of communication, for example, telephony, Web, email, presence and instant messaging. For this reason, it is an important goal that it be possible to implement a servlet container that supports both the SIP and HTTP Servlet APIs at the same time and in a manner that allows applications to include both SIP and HTTP components. This specification defines requirements of such *converged containers* throughout the text.

A combined SIP/HTTP servlet application will contain deployment descriptors pertaining to both the SIP and HTTP parts and will share state through the notion of *application session* objects representing instances of the application (see chapter 10).

1.5 Examples

1.5.1 A Location Service

Routing is an integral part of SIP and is a common function of SIP services. This example is a location service that performs a database lookup on the request URI of incoming requests and proxies the request to the set of destination addresses associated with that URI. The steps performed by the application and container are as follows:

1. Alice makes a call to sip:bob@example.com. The INVITE is received by the servlet container which invokes the location service.
2. The location service determines, using non-SIP means, that the callee (Bob) is registered with two locations, identified by, say, two SIP URIs.
3. The service proxies to those two destinations in parallel, without record-routing, and in unsupervised mode.
4. One of the destinations return 200 (OK) and the other branch is cancelled by the container.
5. The 200 is forwarded upstream to Alice and the call setup is completed as per usual.

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

1.5.2 A Multi-Protocol Servlet Application

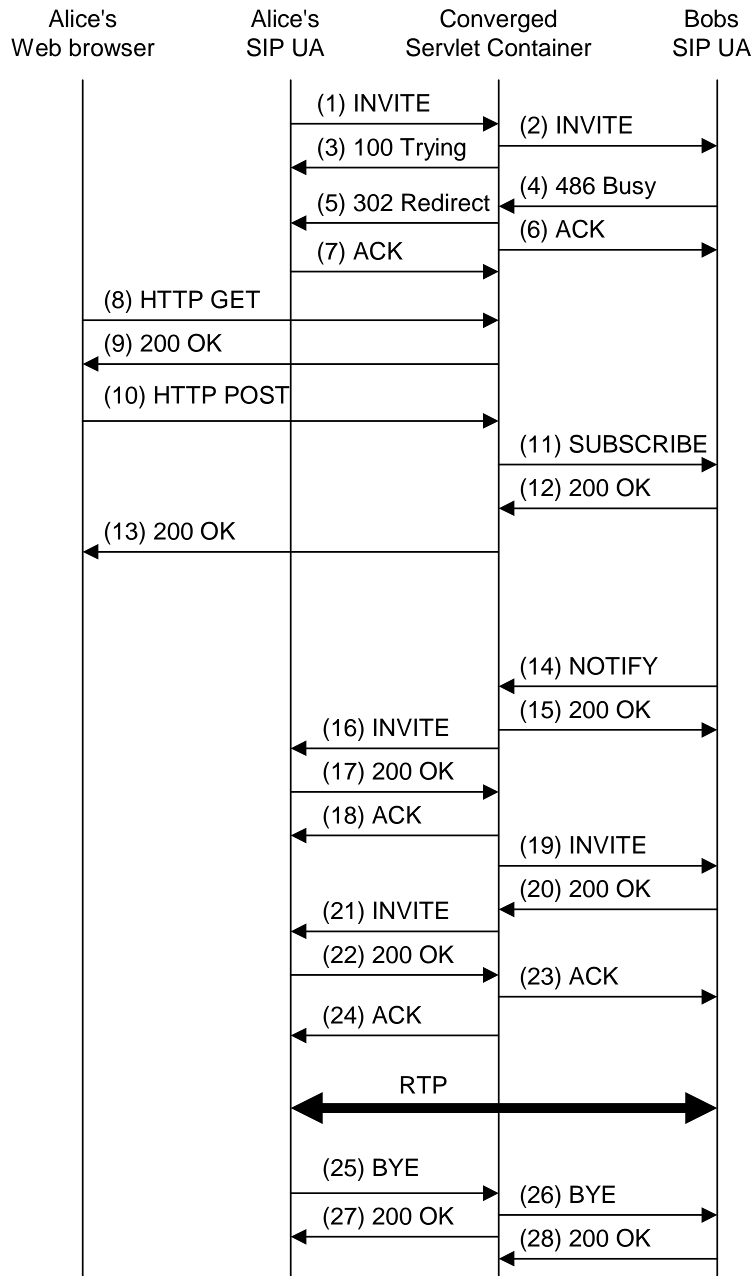
The following is an example of a converged application. It consists of both SIP and HTTP components with SIP being used both in its capacity to establish media sessions and as a presence subscription and notification protocol [simple].

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 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 allowing the caller to ask for a call to be set up the moment the callee becomes available. The caller may then elect to have the application establish a phone call when the callee becomes available again. The application knows about the availability of the callee by subscribing to his or her presence status. When a notification is received that the callee has become available, the application establishes a session between the original caller and callee using third party call control techniques.

Figure 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. It also sends a 100 (Trying) informational response upstream to stop retransmissions.
4. Bob's phone returns a 486 (Busy Here) 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 to Bobs phone as per the SIP specification.
7. Alice's SIP UA sends an ACK 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 an HTML page containing a form allowing Alice 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 gasps at the sight of this amazing application and hits the *Submit* button on the HTML page thus 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 will allow the service to know when Bob becomes available again.
12. Bob's UA accepts the subscription.
13. The HTTP CSBNA servlet receives the posted form and returns a Web page to Alice which she can subsequently use to modify or cancel the scheduled call.
14. At some later point in time, Bob hangs up and 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 will unsubscribe from Bob's status at this point—this is not shown in the diagram.)

FIGURE 1. **Call Flow for the CSBNA Application.**



16–24.The application establishes a call between Alice and Bob using the third party call control mechanisms described in [3pcc][1]. In this case, a call is established and media exchanged over RTP.

25–28.The call is terminated.

In addition to illustrating the functioning of a converged container, this example 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 10.

---

1. Here the general purpose recommended 3pcc call flow is used to set up the call. However, in this particular case, the application does have enough information that it may be reasonable to use a simpler call flow that avoids the re-INVITE in step 21. First, from the initial failed INVITE it knows the session description used by Alice's SIP UA, and second it has reason to believe that Bob is available to answer the call immediately.

2 Request Routing and Application Composition

An important function of SIP servlet containers is to route requests to applications. When determining which applications to invoke and in which order, containers distinguish between *initial* and *subsequent* requests. Initial requests are routed based on the set of rules of all applications deployed to the container, whereas subsequent requests are routed based on the path taken by a previous initial request.

2.1 Initial Requests

Each application has a set of rules associated with it, specifying under which circumstances, i.e. for which requests, the application is interested in executing. Application rules are specified declaratively and are carried in the deployment descriptor (the rule language is defined in chapter 11).

An initial request is a request for which the container has no a priori internal routing knowledge. It may or may not be a request capable of establishing a dialog. The container matches initial requests against the set of rules with which the container is configured and invokes applications corresponding to triggered rules.

Responses always follow the reverse of the path taken by the corresponding request. This is true for both initial and subsequent requests.

2.2 Subsequent Requests

If an initial request results in a SIP dialog being established and at least one application is on the signaling path, be it as a UA or as 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 which are on the signaling path. Requests that are routed internally in a container based on such state are called *subsequent requests* in this specification.

Correct routing of subsequent requests can be achieved in several ways and it is up to implementations to choose one. One approach is to explicitly store dialog related routing information internally and look it up based on a dialog ID computed for incoming requests. Another is to push routing information, e.g. in the form of session IDs, to endpoints in Record-Route headers and retrieve it back from Route headers of subsequent requests.

2.2.1 Message Context

When passed to applications, SIP servlet request and response objects are always associated with a `SipSession` object which, in turn, belongs to a `SipApplicationSession`. This specification refers to the combination of application, application session, and `SipSession` as the *mes-*

sage 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 application session and `SipSession` objects associated with them, for performance reasons, containers may choose to defer creation of session objects when applications cannot observe the difference.

2.2.2 Application Path

Unlike initial requests, subsequent requests are not dispatched to applications based on configured rules, but rather follow the path, or more precisely a subset of the path, of the corresponding initial request, or the reverse path if the subsequent request originated from the callee as opposed to the caller.

Basically, an initial request which establish a SIP dialog also establish an *application path* within the container, along which subsequent requests in the dialog are routed. This path consists of message contexts for applications acting either as UA or as a record-routing proxy for the initial request and specifies the context in which subsequent requests are processed. The application path is a logical concept and as such may or may not be explicitly represented within containers.

A subsequent request is passed to each of the applications on the application path in turn. The application path is traversed in a direction that depends on whether the subsequent request is received from the caller or the callee. Subsequent requests from the caller are processed by applications in the same order as was the initial request. Subsequent requests received from the callee follow the reverse path, that is, they are passed to the last application first, then to the last but one application, up to the first application, before being proxied towards the caller (if the first application on the path is not itself the caller). Proxying applications are invoked for subsequent requests only if they record-routed when proxying the initial request, see section 8.1.

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 will then be routed based on this application path, and will be passed first to C and then to A.

NOTE: Routing subsequent requests based on the path established by the initial request is consistent with the cascaded services model discussed in section 2.4.1.

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 8.2.8.

It is worth noting that a `SipSession` can belong to at most one application path. This is because initial requests are processed in the context of new `SipSessions` 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 will create *derived* `SipSessions` for the second and subsequent paths being created, see section 10.2.3.

2.3 Application Path Cleanup

When a `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.

It is possible for a container to receive a request belonging to a dialog it *used* to have internal routing information for but where this routing information has been purged. In this case it is up to container policy whether to reject the request, proxy it without application involvement, or whether to treat it is an initial request, that is, to dispatch it to applications based on the rule set.

2.4 Application Composition

SIP application servers typically host many different services simultaneously. Since more than one application may be interested in processing the same initial request, multiple rules may be satisfied for any given request, and it is desirable to allow multiple applications to trigger on the same request. A typical example from traditional telephony is a call-screening and a call-forwarding service as these would both execute on incoming INVITEs destined for subscribers to these services.

It is clear that precautions must be taken by containers to ensure that applications will not interfere adversely with one another and that the application server behaves in accordance with the SIP specification. This section specifies what conditions for application composition must be satisfied by any compliant SIP servlet container.

2.4.1 Cascaded Services Model

The model of application composition adopted in the SIP Servlet API is that of the *cascaded services model* [SERL]. This model states that:

Triggering of service applications on the same host, shall be performed in the same sequence as if triggering had occurred on different hosts.

This principle guarantees that each invoked service has a consistent view of the context in which it is executing—application developers can write SIP services *as if* they were the only application processing the incoming request. This means services can execute independently of one another, which greatly simplifies application development.

The simplest realization of this strategy is for the SIP servlet container to always select just one matching rule when determining which application to invoke for incoming requests. In this case there is no application composition but the behavior is consistent with the cascaded services model and so is a valid implementation.

2.4.2 The Role of Proxying

For SIP, the requirement that multiple applications execute as if they were located on separate physical hosts implies that application composition takes place when a request sent by one application, either as a UAC or as a proxy, is routed by the container to another application. Generally speaking, applications don't know, and doesn't need to know, whether a request they send or proxy is going to be sent directly to the outside world by the container or to another application with a matching rule. This is an extremely important property of the SIP Servlet API as it offers a well-defined framework within which application composition can take place.

If, for example, two applications, A and B, are deployed with rules matching an incoming INVITE, the container may choose to first invoke application A. If A then proxies the INVITE, the container may choose to invoke B (provided B's rule still matches, see below). B is then free to respond or proxy the request independently of A. Regardless of how B choose to handle the request, A is guaranteed to be presented with a consistent view of the transaction, for example, A can expect to receive a final response in accordance with the SIP and SIP Servlet API specifications regardless. Implementations are free to employ any mechanism of their choice for deciding which of several matching rules to trigger next.

It follows that a forking proxy may cause multiple invocations of the same or different applications logically downstream on the same host. For example, depending on the rule configuration, an outbound call screening application might be invoked once for each INVITE proxied by a forking find-me application.

Another implication of the "composition-by-proxying" model is that, as previously mentioned, responses are processed in reverse order of requests, that is, they are passed upstream from the last application to handle the request logically upstream towards the application first invoked to handle the corresponding request.

2.4.3 Loop Detection

The requirement that a triggering rule always be true when the associated servlet is invoked may cause some implementations to re-evaluate rules when a request is proxied. For such servers there is a possibility that loops in invoked service logic may occur, that is, A proxies to B which proxies back to A, etc. It is obviously important that such loops be detected and handled somehow. 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. Section 8.2.9 discuss the issue of when to generate 483 (Too Many Hops) error responses on basis of the Max-Forwards header.

2.4.4 Limitations of the Model

The Servlet API defines the logical roles of application developer, assembler, and deployer, and there is an implicit understanding that applications deployed to a servlet container are independent. For this reason, and because it is simple, well-defined, clean, and easily explained, the cascaded services model is a good fit to the SIP Servlet API.

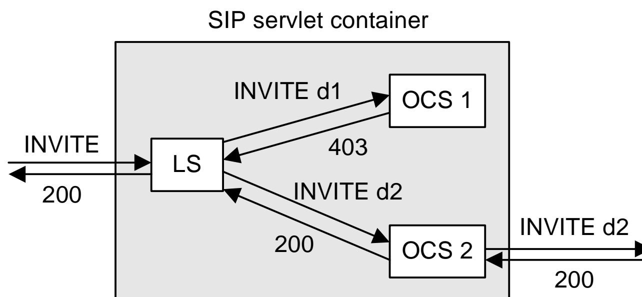
However, in some cases services are indeed written by the same person or organization, and they may not be independent. In such cases of more closely coupled services, it may be desirable to have other models for application composition. Future versions of the SIP Servlet API may define such alternative models.

2.5 An Example

For a simple illustration of application composition and request routing, consider the location service (LS) of section 1.5.1 and an originating call screening application (OCS). This latter application screens outgoing calls based on the destination address. For purpose of illustration, it is assumed that the OCS application record-routes.

Figure 2 shows the routing of an initial INVITE request. The following step-by-step walkthrough of the callflow indicates session context as a triplet (*app*, *as*, *ss*) where *app* is the application name, *as* specifies an application session, and *ss* specifies a `SipSession`.

FIGURE 2. Example of Application Composition.



1. The container receives an INVITE request. The INVITE does not belong to an existing SIP dialog and so the container routes it to applications based on its rule set.

2. The LS rule matches and this application is chosen by the container. The LS application is invoked in the context of (LS, as1, ss1).
3. The LS performs a database lookup and proxies to two destinations, *d1* and *d2*.
4. Both proxied requests go back to the container which again performs rule matching. This time the container choose to trigger the OCS rule for both proxied requests. The INVITE d1 request is processed in context (OCS, as2, ss2) and INVITE d2 in context (OCS, as3, ss3) – the two OCS invocations are associated with different session objects and will see different request objects with request URIs *d1* and *d2*, respectively.
5. The first instance of the OCS application finds that *d1* is on the screening list and rejects the request with a 403 (Forbidden) response.
6. The 403 response is passed upstream along the *transaction path* in the reverse direction of the request. It is received by the LS application where it contributes to finding the best response. It is not passed to the application at this point.
7. The second instance of the OCS application finds that *d2* should not be screened and so proxies the request (without changing the request URI).
8. No more rules match the INVITE d2 request, so the container proxies it toward its destination, *d2*, that is, outside the application server.
9. The container receives a 200 (OK) response for the INVITE d2 branch. Again, the response follows the transaction path in reverse, i.e. the container passes it first to the OCS 2 application in context (OCS, as3, ss3) and then to the LS application in context (LS, as1, ss1).
10. The 200 is the best response received by the LS proxying application and so the application is passed the 200 response. The container then forwards the response upstream towards the caller.
11. Since the 200 response establish a dialog, an application path is created, meaning the container ensures that it will be able to route subsequent requests in that dialog to applications on the signaling path. In this case the application path consists of the single item (OCS, as3, ss3).
12. 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 technical reasons only and are simply dropped.)
13. The container passes the ACK to the OCS application in context (OCS, as3, ss3) and when the upcall returns, proxies the ACK outside the application server according to standard SIP routing rules.
14. Same procedure is followed for the BYE when one is received.

This is a classical example of feature interaction. The order in which the applications execute is significant as call screening could be circumvented if applied before the location service. Containers will typically provide administrators with some mechanism for prioritizing applications.

Note that, logically, each application has it's 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 a its own response context [RFC 3261, chapter16]. This specification then "augments" the RFC 3261 defined state machines with additional rules, for example, that 100 responses, ACKs for non-2xx's, and responses for CANCELs are ignored.

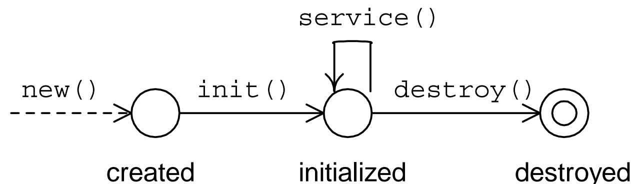
3 The Servlet Interface

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 will extend `SipServlet` to implement their servlets.

3.1 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 3.

FIGURE 3. Servlet lifecycle



Very briefly, the servlet container loads the servlet class, instantiates it and invokes `init()` on it, passing along configuration information in the form of a `ServletConfig` object. Having been successfully initialized, the servlet is available for service, and 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.

There is no explicit provision for dynamically changing rules or other configuration information, or for updating the application code itself. Implementations may support such operations by deploying a separate application with the new configuration and/or code and destroying the old application.

3.2 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;
```

If the event is a request the response argument is null, and vice versa, if the event is a response, the request argument is null. When invoked to process a SIP event, the arguments must implement the `SipServletRequest` and `SipServletResponse` interfaces[1]. 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 dispatches further as described in the following sections.

3.3 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 `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

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 exten-

1. Analogous to `HttpServletRequest` and `HttpServletResponse` in the HTTP Servlet API.

sions. PRACK is defined in RFC 3262 and is discussed in section 6.7.1. The SUBSCRIBE and NOTIFY methods are defined in the SIP event notification framework [RFC 3265] upon which the SIP presence framework is defined [simple]. The MESSAGE method supports instant messaging [RFC 3428], and the INFO method is a general purpose mid-dialog signaling transport mechanism [RFC 2976].

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 section 8.2.8. If the request is initial, it is rejected with status code 500; otherwise the method does nothing (if the application proxied the initial request, the container will proxy the subsequent request when the method call returns). A servlet will typically override 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 containers 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 will not generate a response of its own if a servlet returns control to the container without having responded to an incoming request[1].

Applications wishing to handle SIP methods unknown to `SipServlet.doRequest` can override this 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);
    }
}
```

3.3.1 Receiving Requests

SIP servlets are invoked to process incoming requests in the following cases:

- It's an *initial* request, it matches a rule associated with the servlet, and the container chooses to trigger this rule.
- It's a *subsequent* request in a dialog for which the application is a UA or in which it is a proxy and it record-routed on the initial request.
- It's an ACK for a 2xx response to an INVITE which the application either responded to as a UAS or proxied with record-routing enabled.

1. This is different from the HTTP Servlet API where the container *will* generate a response upon return of control from the servlet.

- It's a CANCEL for an INVITE the application has received but not yet generated a final response for.

In all cases the servlet is invoked by the container 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, the servlet container must generate a 500 response to that request. The header or body of the response may contain additional information that can be of use in identifying the cause of the problem.

3.4 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
- `doSuccessResponse` for handling SIP 2xx responses
- `doRedirectResponse` for handling SIP 3xx responses
- `doErrorResponse` for handling SIP 4xx, 5xx, and 6xx responses

Chapters 7 and 8 describe, in detail, the rules surrounding the invocation of these methods.

3.5 Number of Instances

Refer to [Servlet API, section 2.2].

NOTE: The number of servlet instances created by a container is different from what is called *application instances* in this specification. This latter term effectively refers to an application session together with its contained protocol sessions as described in chapter 10. Servlet objects are independent of any particular application instance and will typically process requests belonging to many different application instances.

---

4 Servlet Context

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

4.1 The SipFactory

The `SipFactory` interface is used by servlets to create instances of various interfaces:

- **requests:** the `createRequest` methods create instances of the `SipServletRequest` interface and is used by UAC applications when creating requests in a new dialog. When creating subsequent requests in an *existing* dialog, `SipSession.createRequest` is used instead. Section 7.1.1 discusses requirements of the `createRequest` methods.
- **address objects:** ability to create `URI`, `SipURI`, and `Address` instances.
- **application sessions:** ability to create new application sessions. This is meant primarily to be used during startup when the application is invoked to initialize itself, and should be used sparingly.

All servlet containers must make an instance of the `javax.servlet.sip.SipFactory` interface available to servlets via the context attribute of the same name, `javax.servlet.sip.SipFactory`.

4.2 Extensions Supported

SIP servlet containers must make an immutable instance of the `java.util.List` interface available as a `ServletContext` parameter with name `javax.servlet.sip.supported`. This `List` contains the `String` names of SIP extensions supported by the container. This can be used by applications to determine whether the container supports a particular extension. For an example use see section 6.7.1.

4.3 Context Path

The Servlet API defines the notion of a *context path*. This is a URL path prefix with which a web application is associated. All requests with an HTTP URL starting with the context path of a web application will be routed to the corresponding servlet context. As SIP URIs do not have a notion of paths, the following `ServletContext` methods have no meaning for SIP-only servlet applications/containers and must return null:

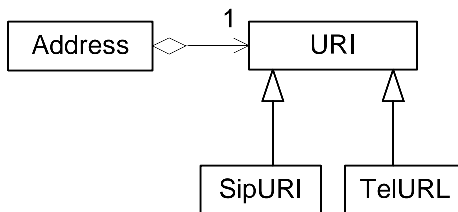
```
ServletContext getContext(String uripath);
String getRealPath(String path);
RequestDispatcher getRequestDispatcher(String path);
```

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 and resource loading proceeds according to the Java Servlet Specification [Servlet API].

5 Addressing

Addressing plays a role in many SIP functions and so, in addition to the request URI, many header fields are defined to carry one or more addresses, for example, From, To, and Contact. The format of these addresses is generally them same. They consists of a URI with an optional display name and an optional set of parameters. Figure 4 shows the addressing abstractions of the SIP Servlet API and the relationship between them.

FIGURE 4. Addressing abstractions



The `SipFactory` interface described in section 4.1 is used to construct instances of these interfaces.

5.1 The Address Interface

The `Address` interface is used to represent values of header fields which 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 which operate on any address header field (see section 6.4.1). This includes the RFC 3261 defined header fields listed above as well as extension headers such as P-Asserted-Identity, P-Preferred-Identity [RFC 3325], Path [RFC 3327] and Refer-To [refer].

It is highly likely that future SIP extensions will define more such header fields and so 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 `Strings`.

SIP Servlet containers will typically have built in knowledge of certain headers but are required to 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 section 6.4.1), 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, for example, 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 From and To Header Fields

The From and To header fields contain the addresses of the UAC and UAS respectively. Section 7.1.1.1 discuss how the container guarantees the integrity of From and To headers through cloning and immutability.

5.1.2 The Contact Header Field

The Contact header field is another address header which warrants extra comments.

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 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 will always have 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 in these cases. Containers should throw an `IllegalArgumentException` on application attempts to set the Contact header field in these cases.

The other use 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. These are the uses where 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 wildcard Contact value and `SipFactory.createAddress` will return a wildcard `Address` given a value of "*". Note that wildcard `Address` objects are legal *only* in Contact header fields.

5.2 URIs

SIP entities are addressed by URI. 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. Particular implementations may know how to handle other URI schemes, e.g. tel URLs [RFC 2806].

Implementations are required to 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 is able to construct `Address` objects containing `URIs` 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 are required to 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 or 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.

As an example, the following SIP URI:

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

contains a transport parameter with value "tcp" and a Subject header with value "SIP Servlets".

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're 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

Represents tel URLs as defined in [RFC 2806]. The tel URL scheme is used to represent addresses of terminals in the telephone network, i.e. 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.

6 Requests and Responses

This chapter describes the structure of SIP servlet message objects. Chapters 7 and 8 describe operational aspects of message processing.

6.1 The SipServletMessage Interface

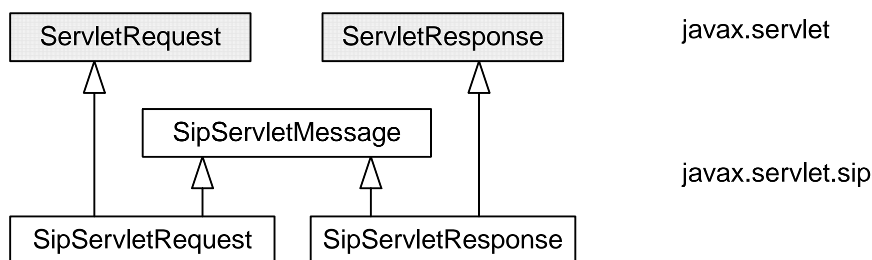
The generic Servlet API is defined with an implicit assumption that servlets receive requests from clients, inspects various aspects of the corresponding `ServletRequest` object, and generate a response by setting various attributes of a `ServletResponse` object. As HTTP servlets reside only on origin 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 need to be more symmetric, that is, requests must be writable as well as readable, and likewise, responses must be readable as well as writable.

The `SipServletMessage` interface defines a number of methods which are common to `SipServletRequest` and `SipServletResponse`, for example setters and getters for message headers and content.

Figure 5 illustrates how the SIP request and response interfaces extend the generic `javax.servlet` interfaces the same way the HTTP request and response interfaces does, but additionally implement the `SipServletMessage` interface.

FIGURE 5. Request-response Hierarchy



6.2 Implicit Transaction State

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

which would violate the transaction state machine, the container throws an `IllegalStateException`.

A `SipServletMessage` for which one of the following conditions is true is called *committed*:

- the message is an incoming request for which a final response has been generated
- the message is an outgoing request which has been sent
- the message is an incoming response received by a servlet acting as a UAC
- the message is a response which has been forwarded upstream

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 and using a chunked encoding for generated content is not practical due to the SIP requirement that a Content-Length header be included in all messages. Also, SIP applications will 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 `ServletRequest` methods `getInputStream` and `getReader` as well as `ServletResponse` methods `getOutputStream` and `getWriter` must return `null`.

The `SipServletMessage` interface defines the following setters and getters 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 wish to 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).

Which type of `Object` is 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` in order 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`, `setContentType`, and `setContentLanguage`.

For incoming messages, the character encoding is specified by the `charset` parameter of the Content-Type header field, if such a parameter is present. In order to deal with cases where the sender didn't correctly specify a character set, `setCharacterEncoding` may be used on incoming messages to force use of a particular encoding.

The default encoding is "UTF-8".

6.4 Headers

A servlet can access the headers of a SIP message 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);
Iterator getHeaderNames();
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` method allow access to all values of a specified header field by returning an iterator over `String` objects representing those values.

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 Address Header Fields

The methods listed above treat SIP header field values as strings. As discussed in section 5.1 many SIP header fields carry addresses and having the ability to access such addresses in a parsed form is both more convenient and allows for better performance than accessing those header field values as strings.

The following methods are defined in terms of the Address interface:

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

Again, if a header field has multiple values in a message, the getAddressHeader method returns the first value. The getAddressHeaders method allow access to all values of the specified header field returning an iterator over Address objects.

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

Address objects may belong to more than one SipServletMessage at a time. In this case modification of the Address will result in modification of the value of the underlying header field for both messages. This sort of aliasing can result in bugs when not intended. Application writers may practice defensive programming by cloning Address objects to avoid sharing.

6.4.2 System Headers

The term *system header* is used to refer to those 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, as well as Contact when used to confer a session signaling address, that is, in messages that other than REGISTER requests and responses and 3xx and 485 responses. There is no need for services to set these 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.

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 has been 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 protocol end-to-end. It may well 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.

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();
```

These methods have meaning for incoming messages only, in which case they return the IP address/port number on which the message was received locally (`getLocalAddr`, `getLocalPort`), or the IP address/port number of the sender of the message (`getRemoteAddr`, `getRemotePort`), as well as the transport protocol used, e.g. “UDP”, “TCP”, “TLS”, or “SCTP”.

6.6 Requests

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

6.6.1 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 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 itself:

- For initial requests where the application is invoked because one of its rules matched, 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 specified the application to be invoked, the parameters are those of the SIP or SIPS URI in that Route header.
- For subsequent requests in a dialog, the parameters presented to the application are those that the application itself set on the Record-Route header for the initial request or response (see section 8.4). These will typically be the URI parameters of the top Route header field but if the upstream SIP element is a “strict router” they may be returned in the request URI (see RFC 3261). It’s the containers responsibility to recognize whether the upstream element is a strict router and determine the right parameter set accordingly.

6.6.2 Attributes

Attributes are objects associated with a request. Attributes may be set by the container to express information that otherwise could not be expressed via the API, or may be set by a servlet to communicate information to another servlet (via `RequestDispatcher`). Attributes are accessed with the following methods of the `ServletRequest` interface:

```
Object getAttribute(String name);
Enumeration getAttributeNames();
void setAttribute(String name, Object o);
void removeAttribute(String name);
```

Only one attribute value may be associated with an attribute name.

Attribute names beginning with the prefix `javax.servlet.sip.` are reserved for definition by this specification. It is suggested that all attributes placed into the attribute set be named in accordance with the reverse package name convention suggested by the Java Programming Language Specification for package naming [JLS].

6.7 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 representing the request that was sent on the branch on which the response was received — `getProxy().getOriginalRequest()` can be used to obtain the original request object (see also sections 8.2.3.2 and 10.2.3).

6.7.1 Reliable Provisional Responses

Provisional responses (responses with a status codes in the 100-199 range) are not sent reliably in baseline SIP. However, an extension has been defined that 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 interworking.

Support for the *100rel* extension is optional in SIP servlet containers ("100rel" is the name of the option tag defined for the provisional response extension). If implemented, the container must include the string "100rel" in the list of supported extensions available to applications through the `ServletContext`, see section 4.2. Applications can determine at runtime whether the container supports the 100rel extension by testing whether this `String` is present in the "supported" list.

When wishing to send a 1xx reliably, the application invokes `SipServletResponse.sendReliably()`. If this method call is successful, the container will send the response reliably. A `Rel100Exception` is thrown if the response is not a 1xx other than 100, if the request is not an INVITE, if the UAC didn't indicate support for the 100rel extension in a Supported or Required header, or if the container itself doesn't support the extension.

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

If no PRACK is received for a reliable provisional response within the time specified by RFC 3262, the container will inform 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.

6.7.2 Buffering

The Servlet API defines a number of `ServletResponse` methods related to buffering of 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 an issue of 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 section 6.2.

6.8 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` contains methods related to internationalization, these assume that servlets only respond to incoming requests and are insufficient for the SIP Servlet API.

6.8.1 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, warnings, etc. This information can be communicated from the UA 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, while `addAcceptLanguage` adds another (least preferred) locale to the list of acceptable locales.

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

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

The `getAcceptLanguage` method will return the preferred locale that the client will accept content in. See section 14.4 of RFC 2616 (HTTP/1.1) for more information about how the Accept-Language header must interpreted to determine the preferred language of the client.

The `getLocales` method will return 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.

6.8.2 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, will set 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 using the following `SipServletMessage` method:

```
Locale getContentLanguage();
```

7 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 various actions are allowed. As mentioned in section 6.2, when an application attempts to perform an action that would violate the SIP state machine, an `IllegalStateException` is thrown by the container.

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

7.1 Client Functions

The following specify how applications can send requests and receive responses.

7.1.1 Creating Initial Requests

When initiating a request a UAC first creates a `SipServletRequest` object and then sends it, possibly after having modified it. Creation of the request object is done by invoking the overloaded `createRequest` method on a `SipFactory` if the request is an initial request, that is, if it doesn't belong to an existing SIP dialog.

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 section 4.1
- 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 ServletException;
```

The returned `SipServletRequest` exists in a new `SipSession` which belongs to the specified `ApplicationSession`. The handler for the newly created `SipSession` is the applications *default servlet*, see section 10.2.6. 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. Sections 7.1.6 and 7.1.7 discuss how and when applications generate those requests. The `SipFactory` methods listed above 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 section 7.3.

7.1.1.1 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.

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.

7.1.2 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. Again, the application may modify the request before invoking it by invoking the send method on the request object.

7.1.3 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 will visit the set of proxies identified in those Route values before being delivered to the destination. This is what is known as a pre-loaded 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 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 section 8.2.1.

7.1.4 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 is obliged to present the application with a final responses. 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. In this particular case, a 404 response would be appropriate.

7.1.5 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 section 10.2.6). 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 will be further dispatched based on the value of the status code.

7.1.5.1 Handling Multiple Dialogs

Due to forking at downstream proxies it is possible that multiple 2xx responses will 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 section 10.2.3. The cloned session object will contain the same application data but its `createRequest` method will create requests belonging to that second or subsequent dialog, that is, with a To tag specific to that dialog.

7.1.6 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, and so SIP servlet containers are responsible for generating ACKs for non-2xx responses, while 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 containers 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.

7.1.7 Sending CANCEL

A UAC can cancel an INVITE request in progress by 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. Note that 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 containers responsibility to delay sending of the CANCEL until a provisional response has been received.

7.2 Server Functions

By definition, a SIP servlet that responds to an incoming request with a final response becomes a UAS for the corresponding transaction.

7.2.1 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.

When proxying a request, applications do not (usually) generate a final response of their own. However, they may still generate provisional responses until the transaction has terminated. This works using the mechanism described above.

7.2.2 Receiving ACK

Applications are notified of incoming ACKs for 2xx responses to INVITEs which 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 section 13.1.

7.2.3 Receiving CANCEL

When a CANCEL is received for a request which 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) and to the CANCEL with a 200 OK final response, and it 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. Neither should it respond to the CANCEL notification.

Clearly, there is a race condition between the container generating the 487 response and the SIP servlet generating its own response. This should be handled using standard Java mechanisms for resolving race conditions. If the application wins, it will not be 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`.

7.3 Back-To-Back User Agents

A back-to-back user agent (b2bua) is a SIP element which acts as an endpoint for two or more dialogs and forwards requests and responses between those two dialogs in some fashion. 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 mediates the signaling between those 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, albeit in a small way, by the SIP Servlet API. 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 following version of `Sipfactory.createRequest`:

```
SipServletRequest createRequest(
    SipServletRequest origRequest,
    boolean sameCallId);
```

This method creates a request identical to the one provided as the first argument with the exceptions that

- The From header field of the new request has a new tag chosen by the container.
- The To header field of the new request has no tag.
- The Call-ID is optionally created afresh, depending on the value of the *sameCallId* argument.
- Record-Route and Via header fields are not copied. As usual, the container will add its own Via header field to the request when it's actually sent outside the application server.
- For non-REGISTER requests, the Contact header field is not copied but is populated by the container as usual.

Note in particular that the Route header field, if present in *origRequest*, *is* copied to the new request.

This method is included for convenience and performance. As for other `SipFactory.createRequest` methods, the returned request belongs to a new `SipSession`.

8 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 whereas HTTP servlets execute on origin servers only and are concerned only with responding to incoming requests, 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`. When proxying statefully, there is at most one `Proxy` object per SIP transaction, meaning that `SipServletMessage.getProxy()` will return the same `Proxy` instance whenever invoked on the original request object or on any other message belonging to that transaction.

8.1 Parameters

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

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

recordRoute: flag controlling whether the application stays on the signaling path for this dialog or not. This should be set to true if the application wishes 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.

stateful: whether the proxying operation should be transaction stateless or stateful. The default value is true, meaning stateful. The ability to proxy without maintaining transaction state potentially yields better performance. However, as this has no semantic implications for applications, containers are free to consider this parameter a hint.

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.

sequentialSearchTimeout: the time the container waits for a final response before it cancels the branch and proxies to the next destination in the target set. The default value is the value

of the `sequential-search-timeout` element of the deployment descriptor or a container specific value, if this is not specified.

Not all combinations of these parameters are allowed. In particular, if any of *recurse*, *parallel*, and *supervised* are true, proxying cannot be stateless.

The *recordRoute* and *stateful* flags may only be set before the first call to `Proxy.proxyTo`. Any attempt to set them afterwards results in an `IllegalStateException` being thrown. The *recurse*, *parallel*, and *supervised* flags as well as the *sequentialSearchTimeout* parameter may be modified by applications while a proxying operation is in progress.

8.2 Operation

An application performs a proxying operation by obtaining a `Proxy` object from a `SipServletRequest`. When creating the `Proxy` object for an incoming INVITE request, the server also sends a 100 provisional response upstream *if* the application did not already sent a 1xx response itself. This is done to stop retransmissions from the client.

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 and modifying or replacing the body. The application then invokes `proxyTo()` to initiate the proxying operation. This method is overloaded to take either a single `URI` or a `List` of `URIs` specifying downstream destination(s). Compliant servlet engines are required to be able to handle SIP and SIPS URIs and may know how to handle other URI schemes, for example, Tel URIs.

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 proxied request will have the request URI replaced 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 will be proxied either to other applications with matching rules or towards the external destination specified by the request URI.

8.2.1 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 URIs on 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 sending 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].

8.2.2 Sending Responses

Applications may generate informational responses of their own before or during a proxying operation. This is done the same way it's done when acting as a UAS.

Once a request has been proxied, the application will not usually generate a final response of its own. However, there are cases where being able to do so is useful and so it's allowed but with some additional constraints.

As long as no final response has been 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:

- 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 *will* perform 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), then that 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.

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

8.2.3 Receiving Responses

For requests proxied statelessly, responses are always forwarded upstream without any application involvement whatsoever.

When in stateful mode, the servlet container is responsible for automatically forwarding the following responses received for a proxying operation upstream:

- all informational responses other than 100
- the *best* response received when final responses have been received from all destinations.
- all 2xx responses

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 2xx or 6xx response is received, the server CANCELs all outstanding branches and will not create new branches.

When an application is invoked to handle the best final response received and this is not a 2xx, the application may add addresses for further destinations through the `Proxy.proxyTo` method. The effect is that a new branch is created for each additional destination, as if the method had been invoked before a (tentative) best answer had been 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 either because

1. the application proxied to more destinations in the notification for a final response
2. one or more 2xx retransmissions were received, see section 8.2.3.2 below
3. multiple destinations returned 2xx responses.

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

8.2.3.1 Handling 2xx Responses to INVITE

As mentioned in section 7.2.1, 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]. Ordinarily, SIP proxies handle 2xx retransmissions by forwarding them statelessly upstream based on the Via header field. In the case of an application server, we have the additional complication that one or more of the proxying applications (there may be more than one due to application composition) may wish to modify the 2xx response before it’s forwarded upstream. In this case it’s necessary to ensure that 2xx retransmissions are modified in the exact same way as the 2xx first received for the INVITE.

This can be achieved in several different ways and it is left to implementations to choose one. One solution is to *not* terminate INVITE client and server transactions when first seeing a 2xx for an INVITE. This enables the application server to handle 2xx retransmissions in a manner similar to

the original 2xx, that is, invoke applications as necessary and then forward it upstream. For this reason, applications proxying INVITEs must be prepared to receive retransmissions of 2xx responses and must modify such retransmissions the same way they did the 2xx originally received.

8.2.3.2 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, and so 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` 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` method.

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

8.2.4 Sending CANCEL

An INVITE proxy operation can be aborted by invoking `Proxy.cancel()`. This will cause the container to terminate all outstanding branches by sending CANCEL requests to the corresponding proxied INVITEs. 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 cancelling a branch can be thought of as a way of speeding up the generation of a final response. The application will still be invoked to handle responses (assuming the *supervised* flag is true).

Calls to `Proxy.cancel()` have the effect of cancelling all branches currently in progress and clearing the proxy transaction's current *target set*. If the application subsequently calls `proxyTo` with additional targets, the proxy transaction will create branches for those targets as usual.

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

There is no provision for cancelling individual branches.

The comments made in section 7.1.7 regarding the container having to delay the sending of a CANCEL until a provisional response has been received apply to proxy transaction branches, also.

8.2.5 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 cancelled, 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 cancelled 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 section 7.2.3.

8.2.6 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, and so SIP servlets should never generate ACKs for proxied requests.

8.2.7 Receiving ACK

ACKs for non-2xx final responses are just 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 (for example to modify addresses in a session description carried in the body of the ACK). In this case, as when proxying the original request, the ACK is proxied downstream in its modified form.

8.2.8 Handling Subsequent Requests

As discussed in chapter 2, 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 rule matching.

When proxying with the *recordRoute* flag being 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, though, and may modify it in the upcall before the server proxies it downstream. Applications should not attempt to explicitly proxy subsequent requests. 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 upcall notifying the application of the subsequent request. After having performed the upcall the container will proxy 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.

8.2.9 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`, and if Max-Forwards is 0 a `TooManyHopsException` is thrown. If unhandled by the application, this will be caught by the container which must then generate a 483 (Too many hops) error response.

8.3 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 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. Section 10.2.3 discusses under which circumstances a container must associate an incoming response or even a subsequent request with a `SipSession` cloned from that of an existing dialog.

8.3.1 Stateless Record-Routing

Some applications may wish to stay on the signaling path of a dialog without being dialog stateful. If a proxy application record-routes but does not retrieve either application session or `SipSession`, the container may insert a Record-Route header field but avoid creation of session objects. The container is still required to invoke the same application(s) for subsequent requests received in that dialog (due to application composition there may be more than one). It will have to insert state into the Record-Route header field to identify those applications.

An application which does stateless record-routing cannot at a later time create session objects, for example, when receiving responses or subsequent requests. If a proxy application knows it *may* need to create a session object on responses for an initial request, it must force creation of the session object before proxying, as otherwise it is too late for the container to change the state pushed to the endpoint in the Record-Route header field.

8.4 Record-Route Parameters

When statelessly record-routing, it is often convenient for a SIP proxy to push state to the dialog endpoints, the user agents, and 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, will contribute to the route set of the user agents and will be 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 with in 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 by modified by the application. These Record-Route URI components will be populated by the container and may or may not have valid values at the time an application proxies a request.

Parameters added to Record-Route header field values in this manner can be retrieved from subsequent requests using the `getParameter` method of `SipServletRequest`, discussed in section 6.6.1.

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 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, and also, 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 would have to encode 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 size data.

This version of the SIP Servlet API does not allow proxy applications to push *different* state to the two endpoints of a dialog. The UAS will copy the Record-Route header field of requests unchanged into 2xx responses and 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 state received in subsequent requests from the caller to differ from that received in subsequent requests from the callee. Such a feature would potentially be useful but may have an adverse effect on performance and security as the Record-Route header in responses cannot be protected end-to-end if rewritten by proxies.

9 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` which is used when creating timers, `ServletTimer` which represents timers and is passed to callbacks, and `TimerListener` which is the callback interface implemented by the application and invoked by the container on timer expiration.

9.1 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` provides 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 may 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 associate with that session.

The meaning of the arguments to the `createTimer` method is:

- `appSession` – the application session that the new `ServletTimer` will be associated with
- `delay` – delay in milliseconds before the `ServletTimer` is to expire the first time
- `period` – time 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 approximately regular intervals but with fixed-delay execution 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 if the server is shut down and subsequently restarted
- `info` – application information to be delivered along with the timer expiration. This is useful for determining the significance of a timer expiration in applications that sets multiple timers per application session.

9.2 ServletTimer

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

The `ServletTimer` interface provides the following methods:

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

Note that when a `SipApplicationSession` is terminated all timers belonging to that application session are cancelled.

9.3 TimerListener

Applications are notified of expiration of timers through the `TimerListener` interface. This interface has 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 in chapter 13). 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>
```

There may be at most one `TimerListener` implementation declared in the deployment descriptor.

10 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.

This implies that a single 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.

10.1 SipApplicationSession

The application session serves two purposes: it provides 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.serv-`

`let.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`.

10.1.1 Protocol Session Iterators

Two `SipApplicationSession` methods allow servlets to obtain `Iterator` objects over 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 `SipSessions`, and “HTTP” to get all `HttpSessions`.

10.1.2 SipApplicationSession Lifetime

Containers manage session data and so need a mechanism for allowing session objects to become garbage. When an application session is destroyed, all contained protocol sessions are destroyed with it. An application session is destroyed either because it times out or because a servlet explicitly invalidates it by invoking the `invalidate` method. For reasons of performance, it is recommended that applications explicitly invalidate session 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.

The timer ensures that application sessions will eventually become garbage regardless of whether an application explicitly invalidates it or not. Servlets can register for application session timeout notifications (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.

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

10.2 SipSession

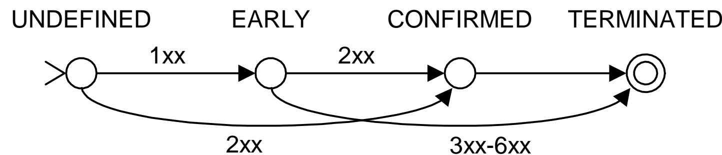
`SipSession` objects represent point-to-point SIP relationships, either as established dialogs or in the stage before a dialog is actually established. The `SipSession` can be obtained from a `SipServletMessage` by calling the `getSession` method.

10.2.1 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.

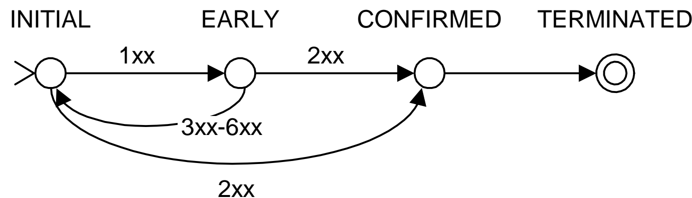
The SIP dialog state machine is shown in Figure 6.

FIGURE 6. The SIP dialog state machine.



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

FIGURE 7. The `SipSession` state machine.



The `SipSession` state machine is shown in Figure 7. It differs from the SIP dialog state machine in the following ways:

- The *initial* state is a valid state. Multiple requests can be generated in the initial state and the rules defined in section 10.2.2.2 specify how dialog state such as local CSeq is updated in this case.

- A non-2xx final response received in the early state doesn't terminate a SipSession, but rather transitions it back to the initial state, from where more requests may be generated.

The standard SIP rules governing when a second or subsequent response cause a single request to establish multiple dialogs hold unmodified for SipSessions. 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. This second SipSession will be *derived* from the one in which the INVITE was generated as described in section 10.2.3.2 below. Both SipSessions will then represent dialogs in the confirmed state.

The initial state is introduced to allow a UAC to generate multiple requests to be generated 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.

Note that the "pseudo dialog" semantics presented here are 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 en 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).

10.2.2 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 SipSessions 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.

10.2.2.1 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. 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.

10.2.2.2 When in the INITIAL SipSession State

When a UAC or UAS transitions from the early state to the initial state in Figure 7, 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 characteristica:

- 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.

10.2.3 Creation of SipSessions

A big difference between dialogs and SipSessions 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 SipSessions.

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 section 10.2.3.2.

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 section 10.2.3.2 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.

10.2.3.1 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 3265]. In the event framework, a 2xx response to an initial SUBSCRIBE may establish a dialog with subsequent NOTIFY requests establishing additional dialogs. In fact, one or more NOTIFY requests may arrive at the caller and intervening proxies *before* the 2xx for the SUBSCRIBE in which case those NOTIFYs establish dialogs.

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 a dialog.

10.2.3.2 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` is the same as that of the original—in particular, the values are not cloned.

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.

10.2.4 SipSession Lifetime

Containers are not required to track dialog state and do not invalidate a SipSession when the corresponding dialog is terminated. SipSession instances are destroyed, and thus purged from the containers memory, when either the parent application session times out or the SipSession is explicitly invalidated by the application via the invalidate method.

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

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

When a SipSession terminates, either because the parent application session timed out or because the SipSession was explicitly invalidated, the container may purge all knowledge of that SipSession from its memory. In this case, if a subsequent request belonging to the corresponding dialog is received, the container is free to handle it in any one of the following ways:

1. reject the request
2. route the request without application involvement
3. if the application path is empty, treat the request as an *initial* request, i.e. match against existing rules and dispatch to applications accordingly, in the process possibly establishing a new application path

Some implementations may not always be able to tell whether a subsequent request for which it has purged all dialog state is indeed a subsequent request and not a new initial request with a preloaded Route. In this case the container has no choice but to treat it as a new initial request and perform rule matching.

It is quite possible that different application instances on the same application path have different lifetimes. Containers may apply either option 1 or 2 to subsequent requests received on a dialog corresponding to a *partially invalidated* application path, that is, a path for which at least one, but not all, application instances have been invalidated. When receiving a request that belongs to a partially invalidated application path, the container first invokes all applications up to the first invalidated instance. Then, in case of option 1, the request is rejected, and in case of option 2, the next valid application instance is invoked or, if the application path is exhausted, the request is proxied according to the rules of section 8.2.8.

10.2.5 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.

10.2.6 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 *default servlet* to be the handler for the newly created `SipSession`. The `SipSession` handler will be invoked to handle responses for all requests as well as new incoming requests. The application default servlet is defined to be the first one listed in the SIP deployment descriptor.

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`, that is, it is a servlet name. Servlet names are typically associated with servlets through the deployment descriptor.

10.2.7 Binding Attributes into a SipSession

A servlet can bind an object attribute into a `SipSession` by name. Any object bound into a session is available to any other servlet that belongs to the same `ServletContext` and that handles a request identified as being a part of the same session.

Some objects may require notification when they are placed into, or removed from, a session. This information can be obtained by having the object implement the `SipSessionBindingListener` interface. This interface defines the following methods that will signal an object being bound into, or being unbound from, a session.

- `valueBound`
- `valueUnbound`

The `valueBound` method must be called before the object is made available via the `getAttribute` method of the `SipSession` interface. The `valueUnbound` method must be called

after the object is no longer available via the `getAttribute` method of the `SipSession` interface.

10.3 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 request. 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.

10.4 Important Session Semantics

10.4.1 Threading Issues

Multiple servlets executing request threads may have active access to a single session object at the same time. The developer has the responsibility for synchronizing access to session resources as appropriate.

10.4.2 Distributed Environments

Within an application marked as distributable, all requests that are part of a session can only be handled on a single VM at any one time. In addition the container must be able to handle all objects placed into instances of the `SipSession` class using the `setAttribute` or `putValue` methods approriately.

- The container must accept objects that implement the `Serializable` interface.
- The container may choose to support storage of other objects in the `SipSession`, such as references to Enterprise JavaBeans 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 session 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 `SipSessions` and their attributes. 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 their attributes will be preserved.

Containers must notify any session attributes implementing the `SipSessionActivation-Listener` 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.

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.

11 Mapping Requests to Servlets

Servlet engines maintain a set of mappings which specify the conditions under which particular servlets should be invoked. Chapter 2 describes how initial requests are matched against configured rules and matching servlet(s) are invoked. This chapter defines the rule language itself.

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.

11.1 Triggering Rules

As discussed in chapter 2, requests that do not belong to an established SIP dialog are routed by the container based on the set of rules of all deployed applications. This is true for requests received from the network as well as for requests received from applications, in case the container performs application composition.

In the present version of the SIP Servlet API, rules are simply predicates over SIP requests. Given a SIP request, a rule evaluates to true of false. A matching rule may be *triggered* by the container, in which case the corresponding servlet is invoked to process the request.

The choice of which of several matching rules to trigger given an initial request is largely left as a matter of container policy. However, the following two conditions must be observed by all containers:

1. A triggered rule must match at the time the corresponding SIP servlet is invoked.
2. 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.

The first condition guarantees the application writer that, even in the face of application composition, he/she can be assured that when a servlet is invoked to process an initial request, there is a rule that evaluates to true for the request. In this sense the rule is a *guard* for the servlet, and is part of the contract between application and container.

The intra-application prioritization of rules imposed by the second condition is needed to guarantee portability of applications across containers.

11.2 SIP Request Object Model

For purposes of the rule language, 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.

The model of the SIP Servlet API rule language consists of the following object types and associated properties:

request object: represents the request itself. This 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

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.

11.3 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 ignores character case when making comparisons. The default is case sensitive matching.

11.4 XML Encoding

A variable is denoted as a path through the request object structure. Individual objects on the path are separated by a “.”, as in “request.uri.user”.

The following XML definitions specify how the rule language defined above is encoded in the `servlet-mapping` element of the deployment descriptor (the deployment descriptor DTD is listed in its entirety in chapter 15).

```
<!ENTITY % condition "and | or | not |
                      equal | contains | exists | subdomain-of">
<!ELEMENT servlet-mapping (servlet-name, pattern)>
<!ELEMENT servlet-name (#PCDATA)>
<!ELEMENT pattern (%condition;)>
<!ELEMENT and (%condition;)+>
<!ELEMENT or (%condition;)+>
<!ELEMENT not (%condition;)>
<!ELEMENT equal (var, value)>
<!ELEMENT contains (var, value)>
<!ELEMENT exists (var)>
<!ELEMENT subdomain-of (var, value)>
<!ELEMENT var (#PCDATA)>
<!ELEMENT value (#PCDATA)>
<!ATTLIST equal ignore-case (true|false) "false">
<!ATTLIST contains ignore-case (true|false) "false">
```

11.5 An Example

Table 1 shows the value of all defined variables for the following SIP request:

```
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“
request.to.uri.user	“watson“
request.to.uri.host	“bell-tel.com“
request.to.uri.port	5060
request.to.display-name	“T. Watson“

and the following sample rule evaluates 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 (as the From display name does contain the string "bell" when disregarding case).

12 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. 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.

12.1 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, libraries etc. of SIP servlet applications exists underneath 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.

12.2 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, also: all servlets within an application see the same `ServletContext` instance.

12.3 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 which ties all of the above elements together.

12.4 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.

12.5 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 actually 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 node is not part of the public document tree of the application. 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 using the `getResource()` and `getResourceAsStream()` method calls on the `ServletContext`. Any application specific configuration information that the developer needs access to from servlet code yet does not wish to be exposed 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 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 class loader.
- 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 class loader 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.

12.5.1 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
```

In this latter case there are separate SIP and HTTP deployment descriptors: `sip.xml` and `web.xml`, respectively. Resources not under WEB-INF/ will be part of the content served from the HTTP part of the converged application.

12.6 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 into such a form, a META-INF directory will be present, which 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 the `ServletContext`.

12.7 SIP Application Configuration Descriptor

The following are types of configuration and deployment information in the SIP application deployment descriptor (the DTD is presented in chapter 15):

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

12.8 Dependencies On Extensions

When a number of applications make use of code or resources, they will typically be installed as library files in the container in current implementations of servlet containers. These files are often common or standard API that can be used without portability being sacrificed. Files used only by one or a few applications will be 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 such libraries servlets in a SAR may have, in order to preserve portability.

Servlet containers are recommended to 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/MANI-FEST. MF entry in the SAR file listing extensions, if any, needed by the SAR. The format of the manifest entry should fol-

low 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://java.sun.com/j2se/1.3/docs/guide/extensions/versioning.html`.

Servlet containers should be able to recognize declared dependencies expressed in the optional manifest entry in a SAR file, or in the manifest entry of any of the library JARs under the WEB-INF/lib entry in a SAR. 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.

12.9 Servlet Application Classloader

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

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

12.10 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.

12.11 Servlet Application Environment

Java 2 Platform Enterprise Edition 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 J2EE[1], 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

1. At the time of writing this is true for HTTP servlets only, but the same mechanisms apply to SIP servlets nonetheless.

`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 J2EE environment with regards to setting up the environment are described in chapter 5 of the Java 2 Platform Enterprise Edition v1.3 specification 2. Servlet containers that are not part of a J2EE compliant implementation are encouraged, but not required, to implement the application environment functionality described in the J2EE 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.

13 Application Listeners and Events

Version 2.3 of the Java Servlet Specification 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 and explicitly allows servlets to implement listeners directly, see section below.

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

13.1 SIP Servlet Event Types and Listener Interfaces

The SIP specific events types and the listener interfaces used to monitor them are as follows.

- listener on `SipApplicationSession`:
  - `javax.servlet.sip.SipApplicationSessionListener`
    A `SipApplicationSession` has been created, destroyed, or has timed out.
- listeners on `SipSession`:
  - `javax.servlet.sip.SipSessionListener`
    A `SipSession` has been created or destroyed.
  - `javax.servlet.sip.SipSessionAttributeListener`
    Attributes on the servlet context have been added, removed or replaced.
- error listener:
  - `javax.servlet.sip.SipErrorListener`
    notification that an expected ACK or PRACK was not received.
- timer listener:
  - `javax.servlet.sip.TimerListener`
    notification that a `Timer` has fired.

Additionally, objects stored as attributes on session objects may implement the `SipSessionBindingListener` and `SipSessionActivationListener` interfaces to get notified of attribute binding and session activation events, respectively, but implementations of those listener interfaces are not listed in the deployment descriptor.

13.2 Servlets Acting as Listeners

The `ServletConfig` and `ServletContext` objects passed to servlets at initialization time provide them, among other things, with configuration information, access to resources, and the ability to perform logging. Listeners do not have the same ready access to the `ServletConfig` and `ServletContext` objects, and so cannot utilize them without the programmer making special arrangements. This is possible but cumbersome.

For this reason, SIP servlets may implement one or more listener interfaces directly. When they do so, the container will *not* instantiate the class separately for its listener function but rather, in the most common case of non-`SingleThreadModel` servlets, will arrange to use the single servlet instance as the listener also.

For example, the following class acts as both a `SipServlet` and a `TimerListener`:

```
public class MyServlet extends SipServlet
    implements TimerListener
{
    private TimerService timerService;

    public void init() {
        timerService = (TimerService)
                getServletContext().getAttribute(TIMER_SERVICE);
    }

    protected void doInvite(SipServletRequest req) {
        ...
        timerService.createTimer(...);
    }

    public void timeout(ServletTimer timer) {
        log("OK to log here...");
    }
}
```

In this case the container creates only a single instance of `MyServlet`, and this instance will serve both as a `SipServlet` and as the `TimerListener` for the application.

Containers typically create a pool of instances for servlets implementing the `SingleThread-Model` interface. If such a servlet class doubles as a listener, containers choose any one of these servlets when it wishes to invoke the corresponding listener.

Note that, for consistency, the deployment descriptor must still contain separate `servlet` and `listener` declarations when the same class implements both, even though the `listener` element will not cause a separate instance to be created.

14 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 mechanism.

This chapter describes deployment representations for security requirements. Similarly 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.

14.1 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 such an environment, a substantial number of servlet applications will 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 has not been 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.

14.2 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 the `RequestDispatcher` to invoke a static resource or servlet using a `forward`.

14.3 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`
- `isUserInRole`
- `getUserPrincipal`

The `getRemoteUser` method returns the user name the client used for authentication. The `isUserInRole` method determines if a remote user is in a specified security role. The `getUserPrincipal` method determines the principal name of the current user and returns a `java.security.Principal` object. These APIs 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 always 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 be passed 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 would be:

```
<security-role-ref>
  <role-name>FOO</role-name>
  <role-link>manager</manager>
</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.

14.4 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 has mapped 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 has mapped 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.

14.5 Authentication

A SIP user agent can authenticate a user to a SIP server using, for example, one of the following mechanisms:

- SIP basic authentication
- SIP digest authentication
- the P-Asserted-Identity header, thus trusting an upstream/downstream proxy to have authenticated the caller/callee, as specified in [RFC 3325]

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 will 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.

14.6 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, and
3. Require re-authentication of users only when a security policy domain boundary has been 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.

14.7 Propagation of Security Identity in EJB™ 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™ container.

In other scenarios, containers are required to allow users that are not known to the servlet container or to the EJB™ 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 signon and customization of data 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 security identity of the caller to the EJB layer in terms of the security role name defined in the `run-as` element. The security role name must one of the security role names defined for the servlet application.

For servlet containers running as part of a J2EE platform, the use of `run-as` elements must be supported both for calls to EJB components within the same J2EE application, and for calls to EJB components deployed in other J2EE applications.

14.8 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.

14.9 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.

15 Deployment Descriptor

This chapter specifies the SIP Servlet Specification, v1.0 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.

15.1 Differences from the HTTP Servlet Deployment Descriptor

The SIP servlet deployment descriptor is very 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 servlets based on the rule language defined in chapter 11 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`, `jsp-file`.
- Resource collections are defined to be a named set of servlets rather than a set of URL patterns as in the case of HTTP.
- 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 must have a deployment descriptor that conforms to the XML DTD presented below. As mentioned in chapter 12, this deployment descriptor exists as a file `sip.xml` in the `/WEB-INF` directory of the servlet application.

15.2 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.

15.3 Deployment Descriptor Elements

The following types of configuration and deployment information exist in the SIP servlet application deployment descriptor and are required to be supported, with the exception of the security syntax, for all servlet containers:

- `ServletContext` init parameters
- Session configuration
- Servlet definitions
- Servlet mappings
- Application lifecycle listener classes
- Error handler
- Filter definitions and filter mappings
- Security

Also, elements exist in the SIP servlet application deployment descriptor to support the additional requirements of servlet containers that are part of a J2EE 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 DTD comments for further description of these elements.

15.4 Rules for Processing the Deployment Descriptor

This chapter lists 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 whitespace characters before the first non-whitespace character, and all trailing whitespace 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. 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 DTD document for structural correctness.

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, tools, and containers should inform the developer 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.

15.4.1 Deployment Descriptor DOCTYPE

All valid SIP application deployment descriptors must contain the following DOCTYPE declaration:

```
<!DOCTYPE sip-app
   PUBLIC "-//Java Community Process//DTD SIP Application 1.0//EN"
   "http://www.jcp.org/dtd/sip-app_1_0.dtd">
```

15.5 DTD

The following DTD defines the XML grammar for a SIP servlet application deployment descriptor.

```
<!-- The sip-app element is the root of the deployment descriptor
for a SIP servlet application -->

<!ELEMENT sip-app (icon?, display-name?, description?,
distributable?, context-param*, listener*, servlet*,
servlet-mapping*, proxy-config?, session-config?,
resource-env-ref*, resource-ref*, security-constraint*,
login-config?, security-role*, env-entry*, ejb-ref*,
ejb-local-ref*)>

<!-- The icon element contains small-icon and large-icon elements
that specify the file names for small and a large GIF or JPEG icon
images used to represent the parent element in a GUI tool. -->

<!ELEMENT icon (small-icon?, large-icon?)>

<!-- The small-icon element contains the name of a file containing
a small (16 x 16) icon image. The file name is a path relative to
the root of the SAR.  The image may be either in the JPEG or GIF
format. The icon can be used by tools.

Example:
  <small-icon>/employee-service-icon16x16.jpg</small-icon>
-->

<!ELEMENT small-icon (#PCDATA)>
```

```
<!-- The large-icon element contains the name of a file containing
a large (32 x 32) icon image. The file name is a path relative to
the root of the SAR.  The image may be either in the JPEG or GIF
format. The icon can be used by tools.

Example:

  <large-icon>/employee-service-icon32x32.jpg</large-icon>
-->

<!ELEMENT large-icon (#PCDATA)>

<!-- The display-name element contains a short name that is
intended to be displayed by tools.  The display name need not be
unique.

Example:

  <display-name>Employee Self Service</display-name>
-->

<!ELEMENT display-name (#PCDATA)>

<!-- The description element is used to provide text describing the
parent element.  The description element should include any infor-
mation that SAR producer wants to provide to the consumer of SAR
(i.e., to the Deployer). Typically, the tools used by SAR consumer
will display the description when processing the parent element
that contains the description. -->

<!ELEMENT description (#PCDATA)>

<!-- The distributable element, by its presence in a servlet appli-
cation deployment descriptor, indicates that this servlet applica-
tion is programmed appropriately to be deployed into a distributed
servlet container -->

<!ELEMENT distributable EMPTY>

<!-- The context-param element contains the declaration of a serv-
let application's servlet context initialization parameters. -->

<!ELEMENT context-param (param-name, param-value, description?)>

<!-- The param-name element contains the name of a parameter. Each
parameter name must be unique in the servlet application. -->

<!ELEMENT param-name (#PCDATA)>

<!-- The param-value element contains the value of a parameter. -->

<!ELEMENT param-value (#PCDATA)>
```

```
<!-- The listener element indicates the deployment properties for
a servlet application listener bean. -->

<!ELEMENT listener (listener-class)>

<!-- The listener-class element declares a class in the applica-
tion must be registered as  a servlet application listener bean.
The value is the fully qualified classname of the listener class.
-->

<!ELEMENT listener-class (#PCDATA)>

<!-- The servlet element contains the declarative data of a serv-
let. -->

<!ELEMENT servlet (icon?, servlet-name, display-name?,
description?, servlet-class, init-param*, load-on-startup?,
run-as?, security-role-ref*)>

<!-- The servlet-name element contains the canonical name of the
servlet. Each servlet name is unique within the servlet applica-
tion. -->

<!ELEMENT servlet-name (#PCDATA)>

<!-- The servlet-class element contains the fully qualified class
name of the servlet. -->

<!ELEMENT servlet-class (#PCDATA)>

<!-- The init-param element contains a name/value pair as an ini-
tialization param of the servlet -->

<!ELEMENT init-param (param-name, param-value, description?)>

<!-- The load-on-startup element indicates that this servlet
should be loaded (instantiated and have its init() called) on the
startup  of the servlet application. The optional contents of these
element must be an integer indicating the order in which the serv-
let should be loaded. If the value is a negative integer,  or the
element is not present, the container is free to load the  servlet
whenever it chooses. If the value is a positive integer  or 0, the
container must load and initialize the servlet as the  application
is deployed. The container must guarantee that  servlets marked
with lower integers are loaded before servlets  marked with higher
integers. The container may choose the order  of loading of serv-
lets with the same load-on-start-up value. -->

<!ELEMENT load-on-startup (#PCDATA)>

<!-- The servlet-mapping element defines a mapping between a serv-
let and a pattern -->
```

```
<!ELEMENT servlet-mapping (servlet-name, pattern)>

<!-- The different types of conditions supported. -->
<!ENTITY % condition "and | or | not | equal | contains | exists |
subdomain-of">

<!-- A pattern is a condition: a predicate over the set of SIP
requests. -->
<!ELEMENT pattern (%condition;)>

<!-- An "and" condition is true if and only if all its constituent
conditions are true. -->
<!ELEMENT and (%condition;)+>

<!-- An "or" condition is true if at least one of its constituent
conditions is true. -->
<!ELEMENT or (%condition;)+>

<!-- Negates the value of the contained condition. -->
<!ELEMENT not (%condition;)>

<!-- True if the value of the variable equals the specified literal
value. -->
<!ELEMENT equal (var, value)>

<!-- True if the value of the variable contains the specified lit-
eral value. -->
<!ELEMENT contains (var, value)>

<!-- True if the specified variable exists. -->
<!ELEMENT exists (var)>

<!-- -->
<!ELEMENT subdomain-of (var, value)>

<!--
Specifies a variable.
Example: <var>request.uri.user</var>
-->
<!ELEMENT var (#PCDATA)>

<!-- Specifies a literal string value that is used to specify
rules. -->
```

```
<!ELEMENT value (#PCDATA)>

<!-- Specifies whether the "equal" test is case sensitive or not.
-->

<!ATTLIST equal ignore-case (true|false) "false">

<!-- Specifies whether the "contains" test is case sensitive or
not. -->

<!ATTLIST contains ignore-case (true|false) "false">

<!-- The proxy-config element configures proxy-related parameters.
-->

<!ELEMENT proxy-config (sequential-search-timeout?)>

<!-- The sequential-search-timeout element defines the default
timeout for sequential searches for all proxy operations performed
by this application. The specified timeout must be expressed in a
whole number of seconds. The container may override this value as a
result of its own local policy. -->

<!ELEMENT sequential-search-timeout (#PCDATA)>

<!-- The session-config element defines the session parameters for
this servlet application. -->

<!ELEMENT session-config (session-timeout?)>

<!-- The session-timeout element defines the default session time-
out interval for all application sessions created in this servlet
application. SipSessions 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. The specified
timeout must be expressed in a whole number of minutes.  If the
timeout is 0 or less, the container ensures the default behavior of
sessions is never to time out. -->

<!ELEMENT session-timeout (#PCDATA)>

<!-- The resource-env-ref element contains a declaration of a
servlet's reference to an administered object associated with a
resource in servlet's environment.  It consists of an optional
description, the resource environment reference name, and an indi-
cation of the resource environment reference type expected by
servlet code.

Example:

  <resource-env-ref>
    <resource-env-ref-name>jms/StockQueue</resource-env-ref-name>
```

```
    <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
  </resource-env-ref>
-->
```

<!ELEMENT resource-env-ref (description?, resource-env-ref-name, resource-env-ref-type)>

```
<!--  The resource-env-ref-name element specifies the name of a
resource environment reference; its value is the environment entry
name used in servlet code.  The name is a JNDI name relative to the
java:comp/env context and must be unique within a servlet applica-
tion.  -->
```

<!ELEMENT resource-env-ref-name (#PCDATA)>

```
<!--  The resource-env-ref-type element specifies the type of a
resource environment reference.  It is the fully qualified name of
a Java language class or interface. -->
```

<!ELEMENT resource-env-ref-type (#PCDATA)>

```
<!-- The resource-ref element contains a declaration of a serv-
let's reference to an external resource. It consists of an optional
description, the resource manager connection factory reference
name, the indication of the resource manager connection factory
type expected by servlet code, the type of authentication (Appli-
cation or Container), and an optional specification of the share-
ability of connections obtained from the resource (Shareable or
Unshareable).

Example:
  <resource-ref>
    <res-ref-name>jdbc/EmployeeAppDB</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
    <res-sharing-scope>Shareable</res-sharing-scope>
  </resource-ref>
-->
```

<!ELEMENT resource-ref (description?, res-ref-name, res-type, res-auth, res-sharing-scope?)>

```
<!-- The res-ref-name element specifies the name of the resource
factory reference name. -->
```

<!ELEMENT res-ref-name (#PCDATA)>

```
<!-- The res-ref-name element specifies the name of a resource man-
ager connection factory reference.  The name is a JNDI name rela-
tive to the java:comp/env context.  The name must be unique within
servlet application. -->
```

```
<!ELEMENT res-type (#PCDATA)>

<!-- The res-auth element specifies whether the servlet code signs
on programmatically to the resource manager, or whether the Con-
tainer will sign on to the resource manager on behalf of the serv-
let. In the latter case, the Container uses information that is
supplied by the Deployer.

The value of this element must be one of the two following:

  <res-auth>Application</res-auth>
  <res-auth>Container</res-auth>
-->

<!ELEMENT res-auth (#PCDATA)>

<!--  The res-sharing-scope element specifies whether connections
obtained through the given resource manager connection factory
reference can be shared. The value of this element, if specified,
must be one of the two following:

  <res-sharing-scope>Shareable</res-sharing-scope>
  <res-sharing-scope>Unshareable</res-sharing-scope>

The default value is Shareable. -->

<!ELEMENT res-sharing-scope (#PCDATA)>

<!-- The security-constraint element is used to associate security
constraints with one or more servlet resource collections -->

<!ELEMENT security-constraint (display-name?,
resource-collection+, proxy-authentication?, auth-constraint?,
user-data-constraint?)>

<!-- The resource-collection element is used to identify a subset
of the resources and SIP methods on those resources within a serv-
let application to which a security constraint applies. If no SIP
methods are specified, then the security constraint applies to all
SIP methods. -->

<!ELEMENT resource-collection (resource-name, description?,
servlet-name*, sip-method*)>

<!-- The resource-name contains the name of this resource collec-
tion -->

<!ELEMENT resource-name (#PCDATA)>

<!-- The sip-method contains an SIP method (INVITE | BYE |...) -->

<!ELEMENT sip-method (#PCDATA)>
```

```
<!-- The presence of the proxy-authentication element indicates to
the container that it must challenge the user agent with a 407
(Proxy Authentication Required) response status code when authen-
ticating an incoming request. If not present a 401 (Unauthorized)
status code is used. -->

<!ELEMENT proxy-authentication EMPTY>

<!-- The user-data-constraint element is used to indicate how data
communicated between the client and container should be protected
-->

<!ELEMENT user-data-constraint (description?,
transport-guarantee)>

<!-- The transport-guarantee element specifies that the communica-
tion between client and server should be NONE, INTEGRAL, or CONFI-
DENTIAL. NONE means that the application does not require any
transport guarantees. A value of INTEGRAL means that the applica-
tion requires that the data sent between the client and server be
sent in such a way that it can't be changed in transit. CONFIDEN-
TIAL means that the application requires that the data be transmit-
ted in a fashion that prevents other entities from observing the
contents of the transmission. In most cases, the presence of the
INTEGRAL or CONFIDENTIAL flag will indicate that the use of TLS is
required. -->

<!ELEMENT transport-guarantee (#PCDATA)>

<!-- The auth-constraint element indicates the user roles that
should be permitted access to this resource collection. The role-
name  used here must either correspond to the role-name of one of
the  security-role elements defined for this servlet application,
or be  the specially reserved role-name "*" that is a compact syn-
tax for  indicating all roles in the servlet application. If both
"*" and  rolenames appear, the container interprets this as all
roles. If no roles are defined, no user is allowed access to the
portion of the servlet application described by the containing
security-constraint. The container matches role names case sensi-
tively when determining access. -->

<!ELEMENT auth-constraint (description?, role-name*)>

<!-- The role-name element contains the name of a security role.
-->

<!ELEMENT role-name (#PCDATA)>

<!-- The login-config element is used to configure the authentica-
tion method that should be used, the realm name that should be used
```

```
for this application, and the attributes that are needed by the
form login mechanism. -->

<!ELEMENT login-config (auth-method?, realm-name?)>

<!-- The realm name element specifies the realm name to use in HTTP
Basic authorization -->

<!ELEMENT realm-name (#PCDATA)>

<!-- The auth-method element is used to configure the authentica-
tion mechanism for the SIP servlet application. As a prerequisite
to gaining  service from any SIP servlet which is protected by an
authorization constraint, a user must have authenticated using the
configured mechanism. Legal values for this element are "BASIC",
"DIGEST", or "CLIENT-CERT". -->

<!ELEMENT auth-method (#PCDATA)>

<!-- The security-role element contains the declaration of a secu-
rity role which is used in the security-constraints placed on the
servlet application. -->

<!ELEMENT security-role (description?, role-name)>

<!-- The security-role-ref element defines a mapping between the
name of  role called from a Servlet using  isUserInRole(String
name) and the name of a security role defined  for the servlet
application. For example,  to map the security role reference "FOO"
to the security role  with role-name "manager" the sytax would  be:

  <security-role-ref>
    <role-name>FOO</role-name>
    <role-link>manager</manager>
  </security-role-ref>

In this case if the servlet called by a user belonging to the "man-
ager"  security role made the API call  isUserInRole("FOO") the
result would be true. Since the role-name "*" has a special meaning
for authorization  constraints, its value is not permitted here.
-->

<!ELEMENT security-role-ref (description?, role-name, role-link)>

<!-- The role-link element is used to link a security role refer-
ence to a defined security role. The role-link element must contain
the name of one of the security roles defined in the security-role
elements. -->

<!ELEMENT role-link (#PCDATA)>
```

```
<!-- The env-entry element contains the declaration of a servlet
application's environment entry. The declaration consists of an
optional description, the name of the environment entry, and an
optional value.  If a value is not specified, one must be supplied
during deployment. -->

<!ELEMENT env-entry (description?, env-entry-name,
env-entry-value?, env-entry-type)>

<!-- The env-entry-name element contains the name of a servlet
applications's environment entry.  The name is a JNDI name relative
to the java:comp/env context.  The name must be unique within a
servlet application.

Example:
  <env-entry-name>minAmount</env-entry-name>
-->

<!ELEMENT env-entry-name (#PCDATA)>

<!-- The env-entry-value element contains the value of a compo-
nents environment entry. The value must be a String that is valid
for the constructor of the specified type that takes a single
String parameter, or for java.lang.Character, a single character.

Example:
  <env-entry-value>100.00</env-entry-value>
-->

<!ELEMENT env-entry-value (#PCDATA)>

<!-- The env-entry-type element contains the fully-qualified Java
type of the environment entry value that is expected by the compo-
nents code.

The following are the legal values of env-entry-type:

    java.lang.Boolean
    java.lang.Byte
    java.lang.Character
    java.lang.String
    java.lang.Short
    java.lang.Integer
    java.lang.Long
    java.lang.Float
    java.lang.Double
-->

<!ELEMENT env-entry-type (#PCDATA)>
```

```
<!-- The ejb-ref element is used for the declaration of a reference
to an enterprise bean's home. The declaration consists of:  - an
optional description - the EJB reference name used in the code of
the servlet that's referencing the enterprise bean - the expected
type of the referenced enterprise bean - the expected home and
remote interfaces of the referenced   enterprise bean - optional
ejb-link information, used to specify the referenced   enterprise
bean   -->

<!ELEMENT ejb-ref (description?, ejb-ref-name, ejb-ref-type, home,
remote, ejb-link?)>

<!-- The ejb-ref-name element contains the name of an EJB refer-
ence. The EJB reference is an entry in the servlet's environment
and is relative to the java:comp/env context.  The name must be
unique within the servlet application.  It is recommended that name
is prefixed with "ejb/".

Example:
  <ejb-ref-name>ejb/Payroll</ejb-ref-name>
-->

<!ELEMENT ejb-ref-name (#PCDATA)>

<!-- The ejb-ref-type element contains the expected type of the
referenced enterprise bean. The ejb-ref-type element must be one
of the following:
  <ejb-ref-type>Entity</ejb-ref-type>
  <ejb-ref-type>Session</ejb-ref-type>
-->

<!ELEMENT ejb-ref-type (#PCDATA)>

<!-- The home element contains the fully-qualified name of the
enterprise bean's home interface.

Example:
  <home>com.aardvark.payroll.PayrollHome</home>
-->

<!ELEMENT home (#PCDATA)>

<!-- The remote element contains the fully-qualified name of the
enterprise bean's remote interface.

Example:
  <remote>com.wombat.empl.EmployeeService</remote>
-->

<!ELEMENT remote (#PCDATA)>
```

```
<!-- The ejb-link element is used in the ejb-ref or ejb-local-ref
elements to specify that an EJB reference is linked to another
enterprise bean.  The value of the ejb-link element must be the
ejb-name of an enterprise bean in the same J2EE application unit.
The name in the ejb-link element may be composed of a path name
specifying the ejb-jar containing the referenced enterprise bean
with the ejb-name of the target bean appended and separated from
the path name by "#".  The path name is relative to the SAR con-
taining the servlet application that is referencing the enterprise
bean. This allows multiple enterprise beans with the same ejb-name
to be uniquely identified.

Examples:
  <ejb-link>EmployeeRecord</ejb-link>
  <ejb-link>../products/product.jar#ProductEJB</ejb-link>
-->

<!ELEMENT ejb-link (#PCDATA)>

<!-- The ejb-local-ref element is used for the declaration of a
reference to an enterprise bean's local home. The declaration con-
sists of:  - an optional description - the EJB reference name used
in the code of THE_COMPONENT   that's referencing the enterprise
bean - the expected type of the referenced enterprise bean - the
expected local home and local interfaces of the referenced   enter-
prise bean - optional ejb-link information, used to specify the
referenced   enterprise bean -->

<!ELEMENT ejb-local-ref (description?, ejb-ref-name, ejb-ref-type,
local-home, local, ejb-link?)>

<!--   The local element contains the fully-qualified name of the
enterprise bean's local interface.   Used by ejb-local-ref  -->

<!ELEMENT local (#PCDATA)>

<!--   The local-home element contains the fully-qualified name of
the enterprise bean's local home interface.   Used by ejb-local-ref
-->

<!ELEMENT local-home (#PCDATA)>

<!-- The run-as element, if defined for a servlet, overrides the
security  identity used to call an EJB by that servlet in this
servlet application. The role-name is one of the  security roles
already defined for this servlet application.  Used by: <servlet> -
->

<!ELEMENT run-as (description?, role-name)>
```

```
<!-- The ID mechanism is to allow tools to easily make tool-spe-
cific references to the elements of the deployment descriptor.
This allows tools that produce additional deployment information
(i.e information beyond the standard deployment descriptor infor-
mation) to store the non-standard information in a separate file,
and easily refer from these tools-specific files to the informa-
tion in the standard sip-app deployment descriptor. -->

<!ATTLIST sip-app id ID #IMPLIED>
<!ATTLIST icon id ID #IMPLIED>
<!ATTLIST small-icon id ID #IMPLIED>
<!ATTLIST large-icon id ID #IMPLIED>
<!ATTLIST display-name id ID #IMPLIED>
<!ATTLIST description id ID #IMPLIED>
<!ATTLIST distributable id ID #IMPLIED>
<!ATTLIST context-param id ID #IMPLIED>
<!ATTLIST param-name id ID #IMPLIED>
<!ATTLIST param-value id ID #IMPLIED>
<!ATTLIST listener id ID #IMPLIED>
<!ATTLIST listener-class id ID #IMPLIED>
<!ATTLIST servlet id ID #IMPLIED>
<!ATTLIST servlet-name id ID #IMPLIED>
<!ATTLIST servlet-class id ID #IMPLIED>
<!ATTLIST init-param id ID #IMPLIED>
<!ATTLIST load-on-startup id ID #IMPLIED>
<!ATTLIST servlet-mapping id ID #IMPLIED>
<!ATTLIST proxy-config id ID #IMPLIED>
<!ATTLIST sequential-search-timeout id ID #IMPLIED>
<!ATTLIST session-config id ID #IMPLIED>
<!ATTLIST session-timeout id ID #IMPLIED>
<!ATTLIST resource-env-ref id ID #IMPLIED>
<!ATTLIST resource-env-ref-name id ID #IMPLIED>
<!ATTLIST resource-env-ref-type id ID #IMPLIED>
<!ATTLIST resource-ref id ID #IMPLIED>
<!ATTLIST res-ref-name id ID #IMPLIED>
<!ATTLIST res-type id ID #IMPLIED>
<!ATTLIST res-auth id ID #IMPLIED>
<!ATTLIST res-sharing-scope id ID #IMPLIED>
<!ATTLIST security-constraint id ID #IMPLIED>
<!ATTLIST resource-collection id ID #IMPLIED>
<!ATTLIST resource-name id ID #IMPLIED>
<!ATTLIST sip-method id ID #IMPLIED>
<!ATTLIST proxy-authentication id ID #IMPLIED>
<!ATTLIST user-data-constraint id ID #IMPLIED>
<!ATTLIST transport-guarantee id ID #IMPLIED>
<!ATTLIST auth-constraint id ID #IMPLIED>
<!ATTLIST role-name id ID #IMPLIED>
<!ATTLIST login-config id ID #IMPLIED>
<!ATTLIST realm-name id ID #IMPLIED>
<!ATTLIST auth-method id ID #IMPLIED>
```

```
<!ATTLIST security-role id ID #IMPLIED>
<!ATTLIST security-role-ref id ID #IMPLIED>
<!ATTLIST role-link id ID #IMPLIED>
<!ATTLIST env-entry id ID #IMPLIED>
<!ATTLIST env-entry-name id ID #IMPLIED>
<!ATTLIST env-entry-value id ID #IMPLIED>
<!ATTLIST env-entry-type id ID #IMPLIED>
<!ATTLIST ejb-ref id ID #IMPLIED>
<!ATTLIST ejb-ref-name id ID #IMPLIED>
<!ATTLIST ejb-ref-type id ID #IMPLIED>
<!ATTLIST home id ID #IMPLIED>
<!ATTLIST remote id ID #IMPLIED>
<!ATTLIST ejb-link id ID #IMPLIED>
<!ATTLIST ejb-local-ref id ID #IMPLIED>
<!ATTLIST local-home id ID #IMPLIED>
<!ATTLIST local id ID #IMPLIED>
<!ATTLIST run-as id ID #IMPLIED>
<!ATTLIST pattern id ID #IMPLIED>
<!ATTLIST and id ID #IMPLIED>
<!ATTLIST or id ID #IMPLIED>
<!ATTLIST not id ID #IMPLIED>
<!ATTLIST equal id ID #IMPLIED>
<!ATTLIST contains id ID #IMPLIED>
<!ATTLIST exists id ID #IMPLIED>
<!ATTLIST subdomain-of id ID #IMPLIED>
<!ATTLIST var id ID #IMPLIED>
<!ATTLIST value id ID #IMPLIED>
```

15.6 Examples

15.6.1 A Basic Example

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE sip-app
  PUBLIC "-//Java Community Process//DTD SIP Application 1.0//EN"
  "http://www.jcp.org/dtd/sip-app_1_0.dtd">

<sip-app>
  <display-name>A Simple SIP Application</display-name>

  <context-param>
    <param-name>vxml-server</param-name>
    <param-value>sip:server17@vxml.example.com</param-value>
  </context-param>

  <servlet>
```

```
    <servlet-name>WeatherService</servlet-name>
    <servlet-class>com.example.Weather</servlet-class>
    <init-param>
      <param-name>weather-server</param-name>
      <param-value>http://example.com/weather</param-value>
    </init-param>
  </servlet>

  <servlet-mapping>
    <servlet-name>WeatherService</servlet-name>
    <pattern>
      <and>
        <equal>
          <var>request.method</var>
          <value>INVITE</value>
        </equal>
        <equal>
          <var>request.to.uri.user</var>
          <value>weather</value>
        </equal>
      </and>
    </pattern>
  </servlet-mapping>

  <session-config>
    <session-timeout>15</session-timeout>
  </session-config>
</sip-app>
```

A References

[3pcc] J. Rosenberg, J. Peterson, H. Schulzrinne and G. Camarillo, “Best Current Practices for Third Party Call Control in the Session Initiation Protocol”, Internet Engineering Task Force, June 2002. Work in progress.

[CPL] J. Lennox and H. Schulzrinne, “CPL: A Language for User Control of Internet Telephony Services”, Internet Engineering Task Force, January 2002. Work in progress.

[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.

[refer] R. Sparks, “The SIP Refer Method”, Internet Engineering Task Force, November 25, 2002. Work in progress.

[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.

[RFC 2806] A. Vaha-Sipila, “URLs for Telephone Calls”, RFC 2806, April 2000.

[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.

[RFC 3325] C. Jennings, J. Peterson, and M. Watson, “Private Extensions to the Session Initiation Protocol (SIP) for Asserted Identity within Trusted Networks”, RFC 3325, November 2002.

[RFC 3327] D. Willis and B. Hoeneisen, “Session Initiation Protocol (SIP) Extension Header Field for Registering Non-Adjacent Contacts“, RFC 3327, December 2002.

[RFC 3428] B. Campbell, J. Rosenberg, H. Schulzrinne, C. Huitema, and D. Gurle, “Session Initiation Protocol (SIP) Extension for Instant Messaging“, RFC 3428, December 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., “SIP Extensions for Presence”, Internet Engineering Task Force, March 2002. Work in progress.

[timer] S. Donovan and J. Rosenberg, “Session Initiation Protocol Extension for Session Timer”, Internet Engineering Task Force, July 2002. Work in progress.

B 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. The output of an application developer is a set of servlet classes and supporting libraries and files (such as images, compressed archive files, etc.) for the servlet application. The application developer is typically an application domain expert. The developer is required to 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 output of the application assembler is a servlet application archive or a servlet application in an open directory structure.

Application path A list of application instances to be invoked for incoming messages belonging to a particular dialog. This list is constructed as a side effect of the handling of the initial request and consists of the application acting as a UAC if the initial request was created within the container, the application acting as UAS if an application responded to the initial request, as well as 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 session Represents application instances. Application sessions acts as a store for application data and provides 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). In order 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 has established. Since it is a concatenation of a UAC and UAS, no explicit definitions are needed for its behavior.

Call A call is an informal term that refers to some communication between peers generally set up for the purposes of a multimedia conversation.

Call leg Another name for a dialog [RFC 2543]; not used in the revised SIP specification [RFC 3261].

Call Stateful A proxy is call stateful if it retains state for a dialog from the initiating INVITE 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 has been fully processed according to this specification and that should not be further modified, see section 6.2.

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 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, 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 is an example of this [RFC 3265].

Downstream A direction of message forwarding within a transaction that 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 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 it 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 it 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 of the Route header field. These procedures separate the destination of the request (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 to 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 is meant to invoke 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 as in a sequential search, 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 *authentica-*

tion data. The content and format of the principal name and the authentication data depend on the authentication protocol.

Protocol session 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 that does not terminate a SIP transaction. 1xx responses are provisional, 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 which 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 sometimes 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 in order 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 has generated a final response. 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, for example SIP *and* HTTP.

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 is associated by 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 which 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 will move the contents of a servlet application into the appropriate locations for that particular container.

SIP applications are to the SIP Servlet API what web applications are to the HTTP Servlet API.

SIP transaction A SIP transaction occurs between a client and a server and 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 is written so that it 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. Also known as a transaction stateful proxy. The behavior of a stateful proxy is further defined in Chapter 16. 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.

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 caused proxies to destroy the contents of the Request-URI when a Route header field was 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, not by matching it against rules, but based on an existing application path created while processing an earlier *initial* request establishing the dialog.

System Administrator The person responsible for the configuration and administration of the servlet container. The administrator is also responsible for overseeing the well-being of the deployed applications at run time.

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 and which servlets must not attempt to modify directly via calls to `setHeader` or `addHeader`. This includes Call-ID, From, To, CSeq, Via, Record-Route, Route, as well as Contact when used to specify a session signaling address, for example, in INVITEs and 200 response to INVITEs. System headers are discussed in section 6.4.2.

TLS Transport Layer Security. A layer four means of protecting TCP connections 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**. The user part is a user name or a telephone number. The host part 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 hyperlink (for example in a Web page) a SIP URI indicates that the specified user or service can be contacted using SIP.

Upstream A direction of message forwarding within a transaction that 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 UAC lasts only for the duration of that transaction. In other words, if a piece of software initiates a request, it acts as a UAC for the duration of that transaction. If it receives a request later, it assumes the role of a 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 which 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 will move the contents of a web application into the appropriate locations for that particular container.

Web application archive A single file that contains all of the components of a web application. This archive file is created by using standard JAR tools which allow any or all of the web components to be signed. Web application archive files are identified by the `.war` extension. A new extension is used instead of `.jar` because that extension is reserved for files which contain a set of class files and that can be placed in the classpath or double clicked using a GUI to launch an application. As the contents of a web application archive are not suitable for such use, a new extension was in order.