JSR-180 Maintenance Release Change Log

Changes are relative to release 1.0.1

Date modified: 01 December 2006

Please send feedback to: `jsr-180-comments@jcp.org`

PROPOSED changes

	Change	Description
1	Allowing multiple credentials in SipClientConnection.setCredentials()	Add a setCredentials(String[] usernames, String[] passwords, String[] realms) method to SipClientConnection so that multiple credential triplets can be given. This is necessary in the forking case when the received 401/407 error response might contain multiple authentication headers. The existing setCredentials method is kept for backward compatibility. Clarify the limitation in it's usage: in the forking case only one authentication triplet can be set in the Unauthorized state.
2	Document the `RuntimeExceptions` that can be expected to be thrown throughout the API.	`RuntimeExceptions` that the application might encounter (like `NullPointerException` if a method argument is null) should be specified in the Throws list of a method's documentation. These are missing from both the specification and the TCK.
3	BYE, NOTIFY, PRACK, UPDATE not in `initRequest()`	Clarify that requests BYE, NOTIFY, PRACK, UPDATE should not be able to be created from `initRequest()`. Since they belong clearly to a dialog, they are created from `SipDialog.getNewClientConnection()`
4	REFER documentation missing from `initRequest()` and `SipDialog`	Add to spec that REFER creates also a dialog.
5	`getHeaders/addHeader/removeHeader`	The specification does not say clearly how the concatenated header values are handled with header manipulation methods. Especially if they are supported and how. Documentation of these methods will be extended as shown in Appendix 1.
6	`SipRefreshHelper.update()` and `stop()`	Update() should throw `SipException`.INVALID_STATE if it is called before the refresh is active (eg. 200 OK is received). Similarly, `SipException` .INVALID_STATE is thrown from stop() if it is called before calling send(). In this case the binding to be cancelled is not yet established so no SIP message must be sent out. Clarify the handling of Expires parameter of contact header and the expires parameter of the update() method in the REGISTER case: If MIDlet sets the “Expires” parameter to the “Contact” header, the value of the “Expires” parameter that is given to the SipRefreshHelper.update() function is ignored. So “Expires” parameter of “Contact” header overrides the “expires” argument of the update() operation always except if value of the “expires” argument of the update() operation is zero. Refreshing is always stopped in that situation. If “Contact” header does not contain “Expires” parameter, the functions parameter is used as the “Expires” value normally. This behaviour follows the rules set in RFC-3261, section 10.3.
7	Clarifications to SipAddress	Clarify that in SipAddress.setURI(), the possible URI parameters don't overwrite existing URI parameters, they are simply ignored if present. Parameters can be manipulated by the get/set/removeParameter() calls. Also clarify that if the password field is present in the URI then SipAddress.getUser() should return the value of 'user:password' (instead of only 'user'). The generic format of the SIP URI is 'sip:user:password@host:port;uri-parameters', and there is no separate method for getting the password.

8	`SipClientConnection.receive()` queues responses.	The specification text is ambiguous in what order the responses should be initialized when receive() is called. The implementation of receive() method should queue the incoming responses in a FIFO queue.

9	Restricted headers access.	It might be good to select and agree on a list of headers that are not accessible or have read-only access by the user from Java API. Typically these headers are related to SIP routing and transaction handling so they are not needed to be read or modified by the user.
		
		Read-only headers. An implementation might throw SipException with INVALID_OPERATION error code if the user tries to modify the following headers:
		
		`Call-ID, Cseq, Proxy-Authenticate, WWW-Authenticate`
		
		Inaccessible headers. An implementation might return null if the user tries to read the following headers. Trying to modify these headers will result in SipException.INVALID_OPERATION:
		
		`Authentication-Info, Authorization, Call-ID, Cseq, Max-Forwards, Min-Expires, Proxy-Authorization, Record-Route, Security-Server, Security-Verify, Service-Route, Via`

10	No value in `SipHeader(String name, String value)` constructor	Giving an empty String or `null` as the value in `SipHeader(String name, String value)` should be allowed. This is convenient when e.g. Proxy-Authentication header is constructed. The user may then fill up parameters one by one. No need to give an initial value in the constructor.
11	`setCredentials()` example 2 code	It is wrong to use the realm value directly from the authentication header with `getParameter("realm")` the difference is: `SipHeader.getParameter("realm")` returns ""realm"" while the value given to `setCredentials()` should not have extra quotation marks. The sample code should be corrected. Clarify in SipHeader.getParameter() that for some header parameters the returned value contains quatation marks while for others it doesn't. Applications must be prepared to handle this.
12	State after error in `SipClientConnection`	Specify that the state of `SipClientConnection` should be Completed if sending ACK fails and there is forking case. It allows trying to send ACK again, wait for more 2xx responses or close the connection. If the application (after investigating the error code) finds out that the error is irrevocable, it can close the connection.
13	Exceptions in Terminated state	What should be the Exception for different methods if the `SipConnection` is in the Terminated state. P22 now says:"Terminated. The transaction and this connection is closed. The I/O methods above will throw IOException " Specify that the methods must throw SipException. INVALID_STATE if that is specified to the method and IOException for those methods (e.g. Input/OutputStream) that does not have SipException specified.
14	`SipServerConnection.send()` errors	It is not possible to give feedback if the asynchronous `SipConnection.send()` fails, later in time. Add a setErrorListener(SipErrorListener sel) method to SipConnection and define a new SipErrorListener interface that the application must implement if it wants to be notified about errors. SipErrorListener.notifyError(String message) contains implementation dependent, non-localized information about the error. Implementations are free to include platform specific error codes, exception traces etc. in the error message. Justification: if a SipServerConnection or SipClientConnectin is using the listener pattern then it won't be notified of transmission errors, as the existing listener callbacks will not be automatically called. For example the midlet will not be informed using the listener pattern if the GPRS connection fails.

15	Header access after message init	It is not feasible to mandate implementations to initialize immediately the headers that are listed in each init-request method. Should loosen the spec so that it says: "Following headers will be set on behalf of the user by the implementation before sending the request. The implementation must set these headers only if they are not overwritten by the user before sending the request" This applies to initRequest() initAck() initCancel() initResponse()
16	CANCEL and ACK dialogs	Specify how `SipServerConnection.getDialog()` behaves when called to the ACK and CANCEL requests? CANCEL: return "null" since the Dialog is in early state and the CANCEL does not relate to the dialog ACK: return the dialog normally if it is available
17	SipConnectionNotifier.acceptAndOpen() throws SecurityException	Add to the specification of SipConnectionNotifier.acceptAndOpen() method that it must check the caller's permissions and throws SecurityException if the required permissions are missing. The exception need not be present in the method's throws list (being a RuntimeException) but it should be listed in the Throws section of the javadoc.
18	Typo in page 1 (javax.microedition.io)	On page 1 (chapter 1, overview), last line just before the diagram. There should be javax.microedition.io.Connection and javax.microedition.io.Connector rather than javax.microedition.Connection and javax.microedition.Connector as printed.
19	`initResponse()` no spec about Contact	`SipServerConnection.initResponse()` does not say how the Contact header should be handled (who sets it) for the requests creating a dialog. See: RFC3261, p162 "Table 2: Summary of header fields, A--O" And also specs for REFER and SUBSCRIBE. Specify that for shared connections Contact is set by the system.
20	`SipServerConnection` state diagram text	Error in JSR180 specs. In the spec of `SipServerConnection`, there is a line just after the Note and state transition diagram : "Following methods are accessible in each state." This line should be changed to something like : "Following methods are restricted to a certain state. The table shows the list of restricted methods allowed in each state. "

21	Clarifications to the refreshing mechanism	The specification is in many places unclear about the exact behaviour of the SipRefreshHelper and SipRefreshListener classes. In addition to items #6, #32, #39, #41, #42 the following clarfications are proposed: 1. If a SipRefreshHelper.stop() operation fails (e.g due to error in the native SIP stack) then refreshing is stopped and a refresh event is sent to the listeners with status code 0. 2. If a SipRefreshHelper.update() operation fails (e.g due to error in the native SIP stack) then refreshing is stopped and a refresh event is sent to the listeners with the status code of the response received. If the status code is not available then an implementation dependent error code (and response phrase) is reported in the refreshEvent. The implementations are free to choose a suitable SIP error code (like 408 – Timeout) or an implementation specific code (with the exception of 0, 1xx and 2xx) and reason phrase. The behaviour is the same in case the OutputStream.close() operation fails within an update() operation. 3. The specification does not define the strategy for timely refreshing of registrations and subscriptions, implementations may choose the algorithm of when to send the refresh request. 4. The implementation MUST report the subsequent refresh responses to the refreshEvent() callback in case of failure response codes (3xx – 6xx), and it MAY report them in case of successful refreshing (2xx). It MUST NOT report provisional (1xx) responses. 5. The application can receive notification of response to the original request twice: once by calling receive() of the SipClientConnection which was used to send the request, and once in the SipRefreshListener.refreshEvent() callback. (Note that according to the previous statement this might not be true in case of successful response: the implementation may not report it in the refreshEvent() callback.) 6. In case of failure of either the original request or a subsequent refresh request the refreshing is automatically stopped and the failure response is reported in the refreshEvent() callback. 7. Passing 'null' as SipRefreshListener in SipClientConnection.enableRefresh() does not clear a previously set listener and does not stop a refresh. Refreshing should be stopped by calling stop(). Calling enableRefresh() for the second time with a non-null value does not overwrite the previously set listener. In this case the previously set listener remains valid, and the method throws SipException.INVALID_STATE 8. If a response arrives to a refresh request that was updated by the application since the request was sent then the application must not report the response in the refreshEvent() callback. 9. After an error response the refresh task is stopped and the corresponding ID is invalidated, so calling update() with the same ID will throw an exception. The application has to start a new refresh task if it wishes so and the implementation will assign a new refresh ID. 10. If refresh responses are not received due to network problems, SipRefreshHelper reports failure to the user in the refreshEvent() callback. The implementations are free to choose a suitable SIP error code (like 408 – Timeout) or an implementation specific code (with the exception of 0, 1xx and 2xx) and reason phrase. 11. JSR 180 refers to RFC3261, 10.2.2 on how to cancel bindings in the REGISTER case. Add as clarification that SUBSCRIBE and PUBLISH require different ways to cancel bindings in RFC 3265, 3.1.4.3 (SUBSCRIBE) and RFC 3903, 4.5 (PUBLISH) which the SipRefreshHelper should comply to. Similarly for update(), refer RFC 3903 Section 4.4 for modifying event state in the PUBLISH case. 12. Giving null or empty string as type or 0 as length in SipRefreshHelper.update() means that subsequent refresh request will have no content. It does not mean that the request will contain the content of the original request unmodified.

22	SipHeader.setParameter("name", "");	The spec should define the behaviour of setting parameter with empty string. Currently it only talks about null. The RI interpretes this at the moment like: SipHeader.setParameter("name", ""); -> name Specify that the behaviour is the same: the result is non-value parameter i.e. token-parameter. Similarly as the result of following call: SipHeader.setParameter("name", null);
23	Reading Content after sending message?	When the request or response has been sent it should not be required from implementations to store the message content. I.e. calling `openContentInputStream()` is not feasible for SipClientConnection or SipServerConnection right after sending message.
24	getDialog() after error response	The spec does not clearly say what the `SipConnection.getDialog()` should return, when 3xx, 4xx, terminating error response has been received. Specify that it should return 'null' when/after error response is received. An already (after 1xx or 2xx) fetched dialog, will turn to 'Terminated' state.
25	Dialog ID in spec at SipDialog.getDialogID()	"Dialog ID (Call-ID + remote tag + local tag)…" Remove exact form of dialogid (callid-tags) from spec, give reference to rfc instead. (RFC3261, p69)
26	Requirements for authentication	The implementations MUST support at least the Digest authentication, as defined in RFC 3261. Implementations that interface with a GSM or IMS/MMD identity module MUST also support Digest-AKA authentication. Implementations MUST handle both 401 and 407 responses if the required authentication mechanism is supported. Implementations MUST support the invocation of `javax.microedition.sip.SipClientConnection.setCredentials` both in Initialized and Unauthorized states. *This is requirement CID.180.3 of MSA (JSR-248). By making MSA requirements mandatory in JSR-180 they can be removed from future releases of MSA.*
27	`SipConnection.getReasonPhrase()` and `getStatusCode()` in *Unauthorized* state.	These methods should be available in *Unauthorized* state.
28	`SipClientConnection.initCancel()` on final response.	Make the following correction at P28 of jsr: Exceptions: “INVALID_STATE if the request can not be set, because of wrong state (in SipClientConnection) or the system has already got the 200 OK response (even if not read with receive() method).” Should be: “INVALID_STATE if the request can not be set, because of wrong state (in SipClientConnection) or the system has already got **a final response** (even if not read with receive() method).”

29	`SipAddress` URI parameters validation	Clarify that an implementation is not mandated to do any semantic checks in SipAddress parser in addition to ABNF syntax validation, which basically accepts any kind of parameter combinations. As information the following checks could be mentioned: (RFC3261 Page 149-151). - Even though an arbitrary number of URI parameters may be included in a URI, any given parameter-name MUST NOT appear more than once. - This extensible mechanism includes the transport, maddr, ttl, user, method and lr parameters. - For a SIPS URI, the transport parameter MUST indicate a reliable transport. - The maddr parameter indicates the server address to be contacted for this user, overriding any address derived from the host field. - The ttl parameter determines the time-to-live value of the UDP multicast packet and MUST only be used if maddr is a multicast address and the transport protocol is UDP. For example, to specify a call to alice@atlanta.com using multicast to 239.255.255.1 with a ttl of 15, the following URI would be used: sip:alice@atlanta.com;maddr=239.255.255.1;ttl=15 - The user URI parameter exists to distinguish telephone numbers from user names that happen to look like telephone numbers. If the user string contains a telephone number formatted as a telephone-subscriber, the user parameter value "phone" SHOULD be present. - The method of the SIP request constructed from the URI can be specified with the method parameter. - Since the uri-parameter mechanism is extensible, SIP elements MUST silently ignore any uri-parameters that they do not understand.
30	Which RFCs and methods are supported.	Follow the requirement in the latest MSA (JSR248) specification. Add the text in Appendix 2 to the package description chapter. *This is requirement CID.180.1 of MSA (JSR-248). By making MSA requirements mandatory in JSR-180 they can be removed from future releases of MSA.*
31	Mark DIALOG_UNAVAILABLE `SipException` error codes as deprecated.	This code is not used anywhere.
32	SipRefreshHelper. Which methods are supported to be refreshed.	Clarify that refreshing REGISTER and SUBSCRIBE requests must be possible using the refreshHelper mechanism, however it is not mandated that refreshing PUBLISH requests is supported by the refreshHelper mechanism. Refreshing PUBLISH requests must be possible by sending the appropriate SIP request from the application. *This is needed to satisfy requirement CID.180.5 of MSA (JSR-248). The MSA requirement is ambiguous in whether it is mandatory to support the refreshing of the given methods by the SipRefreshHelper mechanism. This clarification removes this ambiguity.* *By making MSA requirements mandatory in JSR-180 they can be removed from future releases of MSA.*
33	Responding 100 Trying (non-INVITE transaction)	Add to the SipClientConnection.receive() text and possibly on SipClientConnection state machine diagram that it 100 Trying responses for NIN transactions must not be passed up to the application level. Reasoning: it is not explicitly required/expected by JSR and it is anyway only useful information for the transaction layer (way below JSR180 API), which takes care of request resends.
34	Accept-Contact header in shared connections	It should be specified in more detail, when the Accept-Contact header is required to be set in outgoing requests. The following text should be added: The application must set the Accept-Contact header when using shared connection to all outgoing request except CANCEL and ACK.

35	Connector.open(String name) URI and quoted strings.	In JSR180 the specified URI for shared connections contains quotes as the URI-parameters: `sip:*;type="application/<app_subtype>"[;<other_params>]` In contrary CLDC1.1 says following about the Connector URI "The parameter string that describes the target should conform to the URL format as described in RFC 2396." In that RFC the quote " is considered unsafe and should be escaped. Clarify that - this usage is allowed. Reasoning: this shared URI is not used in the outgoing requests. It is just used locally. The parameters given in the URI are actually compared to the Accept-Contact header parameters, which allows quoted strings. - the implemenation must also allow the usage of the escaped form That is the application can use either quote characters or it's escaped version (%22).
36	Server connection modes	Clarify that it is not mandatory to support both shared and dedicated server connections.
37	Support for SipConnection.getRequestUri()	Clarify that getting and setting RequestUri is not mandated to be supported, that is getRequestURI can return null in any state and setRequestURI may throw SipException.INVALID_OPERATION or SipException_INVALID_STATE.
38	Behaviour of SipDialog objects	Clarify that all SipDialog objects belonging to the same dialog behave the same way: they have same dialog ID and same requests can be created from them. Note that they are not necessarily the same java object. There might be certain scenarios when this requirement is not feasible to be supported. In this case SipDialog.getNewClientConnection() operation must throw SipException.TRANSACTION_UNAVAILABLE, if creation of SipClientConnection object is not possible for any reason.
39	Header values of a SipClientConnection after a SipRefreshHelper.update()	Clarify that an update from a refreshHelper does not change the header values of the original request. Reasoning: SipClientConnection deals with a transaction constructed by the user, whereas SipRefreshHelper is normally implemented as an independet helper module. It will interact with the original request, but it does not update the headers of the original response/request. SipRefrehHelper sends the REGISTER update in an independent transaction. It will only report the MIDlet with listener callback refreshEvent(), where it just tells that 200 ok has been received (update succesfull). If the user is still keeping the original SipClientConnection those header fields are not touched.Generally the getHeader() returns always a header value from currently edited request or the latest response received.
40	Dialog creating methods	Implementations MUST support the creation of dialogs based on at least INVITE, SUBSCRIBE/NOTIFY, and REFER/NOTIFY requests. *This is requirement CID.180.4 of MSA (JSR-248). By making MSA requirements mandatory in JSR-180 they can be removed from future releases of MSA.*
41	Change 200 to 2xx in SipRefreshHelper/Listener	The documented behaviour for SipRefreshHelper / SipRefreshListener does not handle the successful 202 response described in RFC3265 (the RFC for the SUBSCRIBE message). Change references to 200 to 2xx.
42	The statusCode in SipRefreshListener.refreshEvent corresponds to the most recent response received	The current desciption is incorrect as it gives the original request as the source of status data.
43	Automatic response sending before notifying the MIDlet.	Add a new error code to SipException (ALREADY_RESPONDED) to signal that the system has sent a response to the SIP request before notifying the midlet. This is to fit with a platform supporting message queuing, as in RFC3428 (Session Initiation Protocol (SIP) Extension for Instant Messaging). Systems that implement the message relay functionality (see RFC3428) inside the terminal may have definitive knowledge that the terminal has already responded with 202 in which case this exception will be thrown in SipServerConnection.initResponse().

44	Setting Content-Length automatically	SipClientConnection.openContentOutputStream requires that the content length must be known in advance. This disables one of the advantages of output streams; an application that wants to write arbitrary strings via DataOutputStream.writeUTF will have to construct its own byte array, write to it with a ByteArrayOutputStream, and check the size of the array before it can call openContentOutputStream. Proposal: Allow the midlet to open the stream without setting the Content-Length; automatically set it before sending.
45	Redraw the state diagram of SipClientConnection	The state machine for SipClientConnection might be easier to understand if the sequence for sending the third message of an INVITE 3-way handshake was separated from the way to send all other messages. That state machine can also be terminated from any state, not just the ones that are explicitly show with a link to the closed state (so all links to the Terminated state can be implicit). Add this redrawn diagram as explanatory material to the specification of SipClientConnection, but the original one also should be kept as that contains the states used throughout the spec. The new diagram is shown in Appendix 3.
46	Transport protocol support	Add a section about the required transport support to the specification of SipConnection. A proposed wording: All compliant implementations MUST support sending and receiving SIP messages, at least on UDP and TCP transport protocols, as defined in RFC3261 (section 18). The default transport protocol MUST be UDP. Whenever requested, the preferred transport protocol MUST be indicated with the `;transport={transport}` parameter within the URI indicated in the `Connector.open()` method, either for client or server connections. The choice of transport protocols on which an UAS MUST listen are specified by RFC3261 (18.2.1): "For any port and interface that a server listens on for UDP, it MUST listen on that same port and interface for TCP. This is because a message may need to be sent using TCP, rather than UDP, if it is too large. As a result, the converse is not true. A server need not listen for UDP on a particular address and port just because it is listening on that same address and port for TCP." This means that for a SipConnectionNotifier listening on UDP transport protocol is not mandatory if all requests have been sent on top of TCP, that is if all the Contact headers sent within a registration or within dialogs indicate transport=tcp. *This is requirement CID.180.2 of MSA (JSR-248) with some additional explanatory text. By making MSA requirements mandatory in JSR-180 they can be removed from future releases of MSA.*

ACCEPTED changes

None

DEFERRED changes

None

Appendix 1. Clarification to the header manipulation methods

The text in *italic* and the examples are additional to the JSR180 specification. The examples highlight how the header manipulation is realized in different cases.

Method: `setHeader(String name, String value)`

Sets header value in SIP message. If the header does not exist it will be added to the message, otherwise the existing header is overwritten. If multiple header field values exist the topmost is overwritten. The implementations MAY restrict the access to some headers according to RFC 3261 [1].

The implementations are free to store the multiple header field-values either as separate rows or as comma-separated list (if the header type follows the grammar defined in [RFC3261] section 7.3).

The value argument of the method may contain a list of comma-separated header values. If there exist headers of the same type then only the first (topmost) one will be overwritten, indifferently of the number of header values in the comma-separated list. See exampe 3. below.

The method works atomically, that is if the value argument of setHeader() is a list of comma-separated values then the method should either set all of them or none of them. That is if an exception is thrown from the method then no headers are changed, even if the error occurred when setting the second, third etc header value.

Example1: Replacing single header field row. The message already contains following headers:

```
Route: <sip:alice@atlanta.com>
Route: <sip:carol@chicago.com>

setHeader("Route", " <sip:bob@biloxi.com>");
```

the result will be

```
Route: <sip:bob@biloxi.com>
Route: <sip:carol@chicago.com>
```

Example2: Setting multiple header field rows as a comma-separated list. The message already contains one header:

```
Route: <sip:carol@chicago.com>

setHeader("Route", "<sip:alice@atlanta.com>, <sip:bob@biloxi.com>");
```

the result will be either

```
Route: <sip:alice@atlanta.com>, <sip:bob@biloxi.com>
```

or

```
Route: <sip:alice@atlanta.com>
Route: <sip:bob@biloxi.com>
```

Example3: Setting multiple header field rows as a comma-separated list. The message already contains two headers:

```
Route: <sip:carol@chicago.com>
Route: <sip:joe@joe.com>

setHeader("Route", "<sip:alice@atlanta.com>, <sip:bob@biloxi.com>");
```

the result will be either

```
Route: <sip:alice@atlanta.com>, <sip:bob@biloxi.com>, <sip:joe@joe.com>
```

or

```
Route: <sip:alice@atlanta.com>
Route: <sip:bob@biloxi.com>
Route: <sip:joe@joe.com>
```

The result would be the same if the original message contained the headers in a concatenated list form:

```
Route: <sip:carol@chicago.com>, <sip:joe@joe.com>
```

Method: `addHeader(String name, String value)`

Adds a header to the SIP message. If multiple header field values exist the header value is added topmost of this type of headers. The implementations MAY restrict the access to some headers according to [RFC 3261].

The header value string may contain a single value or multiple values as a comma-separated list (that is, if it follows the grammar defined in [RFC3261] section 7.3). The implementations are free to store the multiple header field rows either as comma separated list or in separate rows.

The method works atomically, that is if the value argument of addHeader() is a list of comma-separated values then the method should either add all of them or none of them. That is if an exception is thrown from the method then no headers are changed, even if the error occurred when adding the second, third etc header value

Example1: Adding single header field row. The message already contains header `Route: <sip:carol@chicago.com>`.

```
addHeader("Route", " <sip:alice@atlanta.com>");
```

the result will be

```
Route: <sip:alice@atlanta.com>
Route: <sip:carol@chicago.com>

or

Route: <sip:alice@atlanta.com>, <sip:carol@chicago.com>
```

Example2: Adding multiple header field rows as a comma-separated list. The message already contains header `Route: <sip:carol@chicago.com>`.

```
addHeader("Route", "<sip:alice@atlanta.com>, <sip:bob@biloxi.com>");
```

the result will be either

```
Route: <sip:alice@atlanta.com>, <sip:bob@biloxi.com>
Route: <sip:carol@chicago.com>
```

or

```
Route: <sip:alice@atlanta.com>
Route: <sip:bob@biloxi.com>
Route: <sip:carol@chicago.com>
```

or

```
Route: <sip:alice@atlanta.com>, <sip:bob@biloxi.com>,
<sip:carol@chicago.com>
```

Method: `String getHeader(String name)`

Returns:

Topmost header field value, or null if the current message does not have such a header or the header is for other reason not available (e.g. message not initialized).

Example1: Get topmost Route header from a message that contains three Route headers in a comma-separated header field value.

```
Route: <sip:alice@atlanta.com>,<sip:carol@chicago.com>,
<sip:bob@biloxi.com>

getHeader("Route");
```

the result is:

```
<sip:alice@atlanta.com>
```

Method: `String[] getHeaders(String name)`

Gets the header field value(s) of specified header type. *The method returns the header field-values separated in an array regardless of how they are stored in the message.*

Example1: Get Route headers from a message that contains two Route headers in separate header field rows.

```
Route: <sip:alice@atlanta.com>
Route: <sip:carol@chicago.com>

getHeaders("Route");
```

the result is a String array:

```
{"<sip:alice@atlanta.com>", "<sip:carol@chicago.com>"}
```

Example2: Get Route headers from a message that contains three Route headers in a comma-separated header field value.

```
Route: <sip:alice@atlanta.com>,<sip:carol@chicago.com>,
<sip:bob@biloxi.com>

getHeaders("Route");
```

the result is a String array:

```
{"<sip:alice@atlanta.com>", "<sip:carol@chicago.com>",
"<sip:bob@biloxi.com>"}
```

Method: `removeHeader(String name)`

Removes header from the message. If multiple header field values exist the topmost is removed. The implementations MAY restrict the access to some headers according to [RFC 3261]. If the named header is not found this method does nothing.

The removeHeader() method only removes one header value even if the header values are stored in a comma-separated list.

Example1: Removing header from a message that contains two Route headers in separate header field rows.

```
Route: <sip:alice@atlanta.com>
Route: <sip:carol@chicago.com>

removeHeader("Route");
```

the result is:

```
Route: <sip:carol@chicago.com>
```

Example2: Removing header from a message that contains three Route headers in a comma-separated header field value.

```
Route: <sip:alice@atlanta.com>, <sip:carol@chicago.com>,
<sip:bob@biloxi.com>

removeHeader("Route");
```

the result is:

```
Route: <sip:carol@chicago.com>, <sip:bob@biloxi.com>
```

Appendix 2 SIP methods support

All compliant implementations MUST support at least the methods defined in RFC 2976, RFC 3261, RFC 3262, RFC 3265, RFC 3311, RFC 3428, RFC 3515, and RFC 3903. In particular, implementations MUST support:
– Sending INFO, REGISTER, OPTIONS, INVITE, CANCEL, BYE, ACK, PRACK, SUBSCRIBE, NOTIFY, UPDATE, MESSAGE, REFER, and PUBLISH requests on the `SipClientConnection` interface
– Receiving INFO, OPTIONS, INVITE, CANCEL, BYE, ACK, PRACK, SUBSCRIBE, NOTIFY, UPDATE, MESSAGE, and REFER requests on the `SipServerConnection` interface

- Implementations MUST also freely allow sending and receiving of any other non-dialogcreating requests, whether in-dialog or out-of-dialog, as described in RFC3261

The API contains dedicated methods for initiating some of these requests. Each request type is only required to be supported using the appropriate API method (as defined in the JSR 180 specification).
These RFCs have requirements for both the SIP protocol stack underneath the JSR 180 API as well as for the application level corresponding to the applications using the API. The SIP protocol stack MUST implement those requirements of the RFCs that are relevant for the stack, but following the application level requirements is the responsibility of the applications using the JSR 180 API. The responsibilities are approximately divided as follows:
Following are the requirements of the SIP protocol stack:
– Support the defined method type
– Support possible new header types defined in the RFC
– Support the responses defined in the RFC
– Manage basic transactions for new requests and responses
– Create and manage the dialog, if the request creates a dialog
Following are the requirements of the application level:
– Send appropriate requests and responses in the correct order
– Fill required user headers
– Fill required content with the required content type
– Maintain and provide event states and content for the PUBLISH method, as defined in RFC 3903
– Other application-specific requirements defined by the RFC that are not directly related to the SIP protocol

Appendix 3 SipClientConnection state diagram