RFC-25 Messaging API

Confidential, Draft
veg-rfc_25_messaging-1_00D

21 Pages

Abstract

Services on a Service Gateway must be able to communicate with external entities, such as back-end servers, management systems and other peer services. This RFC presents a simple messaging API, providing a standardized way for sending and receiving datagrams.

0 Document Information

0.1 Table of Contents

0.2 Status

This document specifies a messaging service for the Open Services Gateway Initiative, and requests discussion and suggestions for improvements. Distribution of this document is unlimited within OSGi.

0.3 Acknowledgement

The design and formulation of the material presented in this document has been made in collaboration with Mark Clayton and Bernie Weisshaar, Motorola. Several other members of the OGSi vehicle expert group have also contributed with suggestions and remarks, especially Mattias Rönnblom, Ericsson, and Darryl Mocek, Sun.

0.4 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in [1].

`Source code is shown in this typeface`.

0.5 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
Initial 1.00A	May 31 2001	Anders Rimén, Gatespace ar@gatespace.com, Johan Agat, Gatespace agat@gatespace.com

1.00B	July 23 2001	Updated with comments from the VEG list and from the New York VEG meeting July 10-11 2001.
1.00C	Aug 20 2001	Updated to efficiently support applications layers placed on top of it, by providing insight into the service and a well-defined local behaviour.
1.00D	Sept 04 2001	Updated with comments from Motorola and from the Göteborg VEG meeting August 29-30 2001.

1 Introduction

Services on a Service Gateway must be able to communicate with external entities, such as back-end servers, management systems and other peer services. This document presents a simple messaging API, defining functionality for a client to asynchronously send and receive messages.

The semantics of the send operation is very much like that of any standard datagram service, and provides no guarantee as to in which order messages are delivered or even if they are delivered at all. Streaming and other forms of communication, like synchronous messaging and ordered delivery of messages, are not covered by this RFC. The messaging API can be used to implement such communication services or for negotiating stream parameters, but such details are beyond the scope of this document.

2 Motivation and Rationale

How services communicate will be highly dependent on the software and hardware environment of the Service Gateway, which may even be subject to change over time. The available or preferred communication channels may depend on such factors as:

- Physical location of sender and recipient (mobile applications)
- Urgency of a message (cost vs. speed tradeoff)
- Faults (multiple redundant communication channels)

Service developers may not know at design time the communication capabilities of the target run time environment. A standardized messaging API can abstract the communication and relieve its clients of routing and many other messaging details. The result is:

- Smaller and simpler services
- All services can benefit from advanced messaging features

Many services will need to send messages containing sensitive data, which must be authenticated and/or protected from eavesdropping. Although the messaging API has been designed with these considerations in mind, security is not visible in the API. Instead, secure communication may be controlled by configuration. We refer the reader to Section 3.2.4 for a deeper discussion of this issue.

3 Technical Discussion

The messaging API is designed to provide a simple, transport independent datagram service, allowing more advanced messaging features, e.g., reliable messaging, to be layered above. This section discusses the requirements placed on the messaging API, motivation for the design decisions made and some of the implications that they have.

3.1 Overview of the API Specification

This section gives a brief overview of the messaging API. The full details of the API is presented in Section 4.

The messaging API consists of four interfaces. The fundamental methods for asynchronously sending and receiving messages are provided by the three interfaces `Message`, `MessageService` and `MessageListener`. The fourth interface, `MessageStatusListener`, provides insight into the status of sent messages. The class `FederatedAddressPermission` is used to represent access rights to use the messaging service.

Messages are sent using the `send` method of the `MessageService`, which must first be fetched from the framework service registry. To receive messages, an object implementing the `MessageListener` interface must be registered as a service. A messaging service will deliver all incoming messages to the `MessageListeners` that listen on a port matching the destination of the message. The messages sent and delivered through the messaging API contain a sequence of bytes as the data payload and the address of the remote peer involved. For messages that are sent, the address specifies the destination and for messages received, the address specifies the origin.

A `MessageStatusListener` may also be registered to receive events related to sent messages. These events indicate changes in status or the occurrence of an error.

3.2 Design Decisions and their Implications

3.2.1 Simplicity and Minimalism

The overall design philosophy behind the messaging service is *simplicity*. The API design has been aimed at producing a very simple and minimal API for asynchronous peer-to-peer message communication. It should be suitable for use in an environment where communication is potentially expensive and the communication overhead imposed by the messaging service must be kept at a minimum.

This design philosophy has influenced all other design decisions to a rather large extent.

3.2.2 Generality

Messages in the messaging API have been designed to carry byte arrays, as opposed to `java.lang.Objects` or other more specific data structures. This allows messages to be received and interpreted by applications running on platforms that are not Java enabled, which might be useful for back end systems.

3.2.3 Addressing

Addressing is done using the `FederatedAddress`, described in [3]. The whole purpose of the `FederatedAddress` is to enable communication between OSGi services, which is exactly what this RFC addresses.

3.2.4 Secure Communication and other Advanced Delivery Features

The API does not contain any methods for controlling how a message is transported or delivered, i.e., whether it is encrypted, digitally signed, compressed, or with what level of urgency (priority) it is treated. The messaging API has been designed with security and other advanced delivery features in mind but they have not been explicitly integrated in the messaging API in order to keep it simple.

A messaging service can nonetheless support secure communication and other advanced delivery and transport features but the client application cannot use the messaging API to control when such messages are sent or any of the parameters involved. Instead, these must be controlled using a separate API or through configuration of the messaging service.

Leaving control of security and other advanced delivery features to the configuration of the messaging service has several advantages:

- It gives the Gateway Operator or other entity that manages the configuration of the Service Gateway explicit control over which communication is encrypted, digitally signed, compressed, etc. That entity thus has the possibility to setup the configuration so that all messages sent to and from ports used by sensitive services, e.g., billing services, are digitally signed and encrypted, that all communication routed using a certain low bandwidth transport is compressed, that all communication sent to the back ends alarm handling system is routed with a certain priority, etc.

- It relieves the burden of bundle programmers since they do not need to decide whether to send data using certain delivery features. At the same time, it does not deprive bundles of the possibility to send data on different ports that are configured with different transport and delivery features. For instance, a bundle can use two different ports for its communication, one for high priority messages and one for normal priority. Depending on the functionality and configuration of the messaging service, communications on these ports may be treated equally or with different priority.

3.2.5 Security and Control of Usage

The `FederatedAddressPermission` class defined in the Communication Namespace API, [3], enables messaging services to perform fine-grained control over how a given bundle uses it. The security policy that assigns `FederatedAddressPermissions` to bundles provides an explicit control over the ports used for receiving messages and also the destination address and reply-port used when messages are sent. The security issues addressed with `FederatedAddressPermissions` are:

- Eavesdropping. A bundle will not receive any messages on a port for which it does not have the `FederatedAddressPermission` to listen on.

- Spoofing. A bundle cannot send messages specifying a reply port which it does not have the `FederatedAddressPermission` to send from.

- Unauthorized use of resources. A bundle cannot send messages to a destination host or port unless it has the `FederatedAddressPermission` to send to that destination host and port.

3.2.6 Whiteboard Approach

The messaging API has been designed to use the whiteboard approach. To receive messages, a bundle registers a `MessageListener` in the framework registry. The alternative design would be to have bundles first check out the `MessageService` and then register a `MessageListener` directly with it. One advantage with registering `MessageListeners` as services is that it allows bundles to receive messages without going through the trouble of checking out and tracking the messenger service. The whiteboard approach also allows simple implementation of bundles that never send any messages but only passively listen for messages, e.g., on a port for broadcast of events.

An issue with whiteboard-style `MessageListeners` is that when a bundle registers a `MessageListener` on a particular port, it will not get any feedback from the framework indicating whether it has sufficient `FederatedAddressPermissions` to use that port. If the registration of `MessageListeners` instead were made through a method in the messaging service, the bundle would be notified of insufficient permissions with a `SecurityException`, completely analogous to how the `send` method behaves. Under the whiteboard model, a bundle must actively fetch this information itself, e.g., by calling its associated `Bundle` instance. However, there is generally very little that a bundle can do to compensate for lack of permissions but to log the incident or raise some kind of alarm. If the reason is incorrect configuration of the Service Gateways security policy or the bundle itself, the manager of the gateway is likely do desire a notification of the incident. Such a notification could however be made by the messaging system itself and does not have to involve the bundle directly.

3.3 Message as an Interface

The messaging API specifies the `Message` as an interface instead of a class. Moreover, the `Message` interface only has methods for getting the address and data contained in the Message, not for setting them. As the API does not contain any default implementation of the `Message` interface, bundles using the messaging service must have their own implementations or use implementations provided elsewhere.

While it might seem cumbersome that bundles must implement their own `Messages`, this design follows the principles of minimalism and generality and has great advantages. Most importantly, the API does not standardize any unnecessary functionality and makes implementations of different kind of messages possible. All the information needed by the implementation of the messaging service to send a message is available through the `Message` interface, namely the payload data to send and where to send it. A `Message` is solely intended to communicate this information between the messaging service and the bundle using it. Thus the messaging service never needs to set the data or address of an outgoing message and neither does a client bundle on an incoming one.

3.4 Service Properties

This section describes the service properties used by the messaging related services.

MessageListener:

- `org.osgi.service.messaging.address.host`
  Specifies the host part of the address this listener shall be bound to.

- `org.osgi.service.messaging.address.port`
  Specifies the port part of the address this listener shall be bound to.

MessageService:

- `org.osgi.service.messaging.max_size`
  Indicates an upper bound on the message size handled by the `MessageService`.

4 API Specification

4.1 org.osgi.service.comm.messaging
Interface MessageService

public interface **MessageService**
The `MessageService` provides a simple datagram service allowing bundles to send messages to foreign gateways, or other entities in the `FederatedAddress` name-space. The `MessageService` does not guarantee that sent messages are delivered in order, that a given message is delivered only once, or that it is indeed delivered at all. However, all messages that do reach their destination will do so intact, meaning that the payload data and source address carried by the message have not been corrupted during transport.

A bundle can receive replies by supplying a `ServiceReference` to a `MessageListener`, in which case the address associated with the listener will be used as reply address.

A bundle can track a message by supplying a `ServiceReference` to a `MessageStatusListener`. The specified listener will be called with a status event and the message identity that was returned by the call to `send()` that sent the message for which the event occurred. Users of the `MessageService` API must handle the possible synchronization issues themselves. The `MessageStatusListener` might be called before the message identity returned by the call to `send()` has been properly taken care of, thus reporting an event for a message with an identity yet unknown to the program.

All `MessageServices` shall have a PID assigned, allowing applications to distinguish between messaging services.

A messaging service that is restarted and has messages queued for delivery, must generate a `MessageStatusListener.CANCELED` event for all messages it looses due to the restart.

If a `MessageService` is released by an application, e.g., due to application restart, the messaging service is free to cancel the application messages or continue to deliver them. `MessageStatusListener.CANCELED` events must be generated for all messages canceled.

A `MessageService` may have the service property `org.osgi.service.messaging.max_size` set to indicate that it has a limitation on the message size it can handle. The value of this service property is an `Integer` that should be treated as an upper bound on the size of payload data `Messages` may contain. Sending of `Messages` from/to peers with long addresses might fail even if the size of the message payload is below the indicated maximal size. If this service property is not set, the `MessageService` can handle messages of any size.

Services implementing ordered or more reliable communication features may be layered above the `MessageService`.

A note on resending: When a message is sent with the `MessageService`, the messaging system will do its best effort to deliver the message to its destination. If the delivery fails locally, i.e., before the message has left the local system, the sending bundle is guaranteed a notification via the `MessageStatusListener` interface. If the delivery fails somewhere along the route, the messaging system will do its best effort to notify the sending bundle but there is no guarantee that such remotely generated status events reach a status listener. Thus, in rare cases, it may be correct for a bundle to resend messages even if no message status event signaling delivery error has been received. The bundle programmer should be aware that there is a risk of flooding the network if messages are resent too frequently. Techniques like exponentially increasing timeouts can be used to reduce the risk of network flooding.

Field Summary

`static java.lang.String`	**MAX_SIZE** Constant to help referencing the max-size service property.

Method Summary

`boolean`	`**cancel**(long id)` Cancel the message with the specified identity.
`long`	`**send**(Message msg, org.osgi.framework.ServiceReference listenerRef, org.osgi.framework.ServiceReference statusRef)` Sends the specified message to its destination address.

Field Detail

4.1.1 MAX_SIZE

```
public static final java.lang.String MAX_SIZE
```

Constant to help referencing the max-size service property. It has the value "org.osgi.service.messaging.max_size".

Method Detail

4.1.2 send

```
public long send(Message msg,
                 org.osgi.framework.ServiceReference listenerRef,
                 org.osgi.framework.ServiceReference statusRef)
     throws java.io.IOException
```

Sends the specified message to its destination address. The address and data returned by the message must not be altered before the call to `send()` returns. If the `send()` call returns successfully, the `MessageService` has accepted the message and knows how to transport it one step along the way towards its destination. This does not imply that the message has actually left the sending host. The `send()` method does not block until the message has left the host or reached its destination. It might be queued internally in the `MessageService`, waiting for the appropriate conditions to transport it.

A `ServiceReference` to a `MessageListener` can be specified, in which case the address associated with the listener is used as reply address in the message. If this `ServiceReference` is not specified, the reply address in the message will have its port set to `null` and its host and domain set to addresses supported by the current messaging implementation. If multiple host names are supported by the messaging implementation, one will be chosen.

A `MessageListener` receiving the message can see who sent the message using `Message.getAddress()`. The port name of the address will be `null` if the sender did not specify a `ServiceReference` associated with a `MessageListener` when the message was sent, in which case the receiver will be unable to use the address to send a reply. Otherwise the receiver may create a reply and send it to the address in the received message.

A `ServiceReference` to a `MessageStatusListener` can be specified in order to track message events. The PID of the `MessageStatusListener` will be associated with the message sent, thus allowing status listeners to be reattached to messages. If this `ServiceReference` is not specified, no events will be generated for the message.

Parameters:
`msg` - The message to be sent. The destination address should be returned by the `getAddress()` method and the payload data should be returned by the `getData()` method. The `MessageService` API does not impose any limitations or restrictions on the size of the payload data, a `MessageStatusListener.MESSAGE_TOO_BIG` event will be generated if the message being sent is too large.
`listenerRef` - Service reference to a `MessageListener` that shall receive possible replies, or null if no reply is wanted. All `MessageListeners` listening on the same port as the one specified by `listenerRef` will receive the replies.
`statusRef` - Service reference to a `MessageStatusListener` that shall receive message events, or `null` if no events are wanted. For each message sent, the messaging implementation must either generate an error event, `MessageStatusListener.TYPE_ERROR`, or an event indicating the message has left the local system, `MessageStatusListener.SENT`, i.e., a `MessageStatusListener` registered at the time the event occurs is guaranteed at least one event per message sent. There is no requirement that a messaging implementation shall save events if no message status listener is registered, in this case the event will be lost. There is also no guarantee that remotely generated status events reach a status listener.

Returns:
A message identity associated with the sent message. This identity is used when a `MessageService` reports status events to `MessageStatusListeners`. It can also be used to cancel messages that have not yet left the local system. The identity is unique within the scope of a `MessageService`.
Throws:
`java.io.IOException` - if there is some failure accepting the message for delivery, e.g., the local queue is full.
`java.lang.IllegalArgumentException` - if the Message address is malformed or null, or if the specified `ServiceReferences` are not associated with the correct type of service objects, i.e., implementing `MessageListener` respectively `MessageStatusListener`.
`java.lang.SecurityException` - if a security manager that supports permissions exists and the caller does not have sufficient `FederatedAddressPermissions`.

The caller must have a `FederatedAddressPermission` with action `sendTo` and an address that implies the address in the message. If the port of the destination address is a temporary port, no permission control is performed. Thus, a bundles rights to reply to messages sent from temporary ports is controlled by the ServicePermission to fetch the `MessageService` from the framework.

If the reply port is given with a `ServiceReference` to a `MessageListener` registered with an explicitly specified port, the caller must have a `FederatedAddressPermission` with action `sendFrom` and an address that implies the address/port associated with the `MessageListener`.

If the reply port is given with a `ServiceReference` to a `MessageListener` associated with a temporary port, no special `FederatedAddressPermission` to send from that port is needed. The use of temporary ports as reply ports is controlled by `ServicePermission`. If a bundle has the `ServicePermission` to register a `MessageListener`, it also has permission to send messages from temporary ports.

4.1.3 cancel

`public boolean` **`cancel`**`(long id)`

Cancel the message with the specified identity. The identity is received as a result from a call to `MessageService`.send(). A message can only be canceled if it is locally queued by the messaging service, i.e., it has not yet left the local system. A message can furthermore only be canceled if it was previously sent from the current `MessageService`, i.e., different bundles using `MessageService` cannot cancel each others messages. A message that is successfully canceled is removed from the messaging queue and will not be delivered. A `MessageStatusListener.CANCELED` event is generated for each message successfully canceled.

Parameters:

`id` - The identity of the message to cancel.

Returns:

`true` if the message was canceled from the local queue; `false` if the message does not exist or has already left the local system.

4.2 org.osgi.service.comm.messaging Interface `MessageListener`

public interface **`MessageListener`**

The listener interface for receiving messages. A `MessageListener` is registered using the whiteboard model, i.e., the listener is registered as a service using `BundleContext.registerService()`, whereupon a messaging implementation gets the `MessageListener` service.

Each `MessageListener` is bound to a FederatedAddress and will receive incoming messages sent to that address. The domain part of the address is determined by the messaging implementation getting the service. The host and port parts can be specified by setting the service properties `org.osgi.service.messaging.address.host` and `org.osgi.service.messaging.address.port`.

To specify a port on the current gateway, the host part of the address can be set to `localhost` or not set at all. If the gateway is multi-homed, i.e., listens on multiple host names, the listener will catch messages to all host names for which it has permission.

The registered `MessageListener` object must have `FederatedAddressPermission` with action `listen` to use the specified address. If the registered `MessageListener` object does not have sufficient permissions, or if the address specified is syntactically incorrect, the messaging implementation will discard the listener and never call its `MessageListener`.receive() method.

If the service property `org.osgi.service.messaging.address.port` is not set, a temporary port is created, which is unique for a messaging implementation on the current service gateway, and is exclusively used for the registered `MessageListener`. Temporary port names all begin with `$tmp`, in order to avoid collisions with other listeners. If the service property `service.pid` (persistent identifier or PID) is set, a persistent temporary port name is assigned. If there does not exist a port name mapped to the PID, one is created which is assigned and saved by the messaging implementation. This implies that message listeners can survive life-cycle management of gateway and applications.

`ServicePermission` controls the use of temporary ports. If a bundle has permission to register a `MessageListener` as a service in the framework, it also has permission to receive messages on temporary ports.

`MessageListeners` are called with a Message and a `ServiceReference` when a message is received that is destined to the address bound to the listener. Messages are delivered synchronously to a `MessageListener`, i.e., the `MessageListener`.receive() method will not be called again until a previous call has returned.

If several `MessageListener`s are bound to the same address, they will all receive separate copies of the received message, allowing them to modify its contents without affecting other listeners.

Messaging implementations shall implement `Messages` passed to `MessageListeners` in such a way that they do not contain any references dependent of the messaging implementation's life-cycle, i.e., an application must be able to call `Message` methods independent of the life-cycle of messaging implementation.

Applications shall avoid holding references to messages received for long periods of time. If a message can not be handled directly, i.e., it is saved to be handled at some later time, the contents of the message should be copied in order to avoid stale references to Messages, preventing class loaders associated with messaging implementation from being freed.

The service reference received identifies which `MessageService` to use for sending replies, which is useful if multiple messaging services are registered on the platform.

Field Summary

`static java.lang.String`	**HOST** Constant to help referencing the host service property.
`static java.lang.String`	**PORT** Constant to help referencing the port service property.

Method Summary

`void`	`receive(Message msg, org.osgi.framework.ServiceReference replyRef)` Invoked when a message is received.

Field Detail

4.2.1 HOST

```
public static final java.lang.String HOST
```

Constant to help referencing the host service property. It has the value "org.osgi.service.messaging.address.host".

4.2.2 PORT

```
public static final java.lang.String PORT
```

Constant to help referencing the port service property. It has the value "org.osgi.service.messaging.address.port".

Method Detail

4.2.3 receive

```
public void receive(Message msg,
                    org.osgi.framework.ServiceReference replyRef)
```

Invoked when a message is received. If several `MessageListeners` are bound to the same address, they will all receive separate copies of the received message, allowing them to modify its contents without affecting other listeners.

The receive method is called synchronously, i.e., a new call is not made while a previous call is still active.

Parameters:
`msg` - The `Message` that is received.
`replyRef` - Service reference to a `MessageService` which can be used for sending replies.

4.3 org.osgi.service.comm.messaging
Interface `MessageStatusListener`

public interface **MessageStatusListener**

The listener interface for receiving status events for messages sent. A `MessageStatusListener` is registered using the whiteboard model, i.e., the listener is registered as a service using `BundleContext.registerService()`, whereupon a messaging implementation gets the `MessageStatusListener` service.

Each `MessageStatusListener` shall be assigned a PID. This allows messaging implementations to persistently associate the listener with specific messages, thus implying that status listeners can be re-attached to messages.

A `MessageStatusListener` is assigned to a `Message` when it is sent, using `MessageService.send()`. The listener is called with a message identity and an event code. The message identity is the same value that was returned when the message was sent, and can be used to associate the event with a specific message. Status events are delivered synchronously to a `MessageStatusListener`, i.e., the `MessageStatusListener.statusChanged()` method will not be called again until a previous call has returned.

Field Summary

`static int`	**`CANCELED`** Error code indicating that the message has been canceled.
`static int`	**`COMMUNICATIONS_UNAVAILABLE`** Error code indicating that the communications are not possible.
`static int`	**`DOMAIN_UNREACHABLE`** Error code indicating that the destination domain given in the federated address can not be reached.
`static int`	**`EXPIRED`** Error code indicating that the message has expired in transit.
`static int`	**`HOST_UNREACHABLE`** Error code indicating that the destination host given in the federated address can not be reached.
`static int`	**`INSUFFICIENT_RESOURCES`** Error code indicating that the message could not reach the destination because of insufficient resources someplace in the path.
`static int`	**`MESSAGE_TOO_BIG`** Error code indicating that message being sent is too large for the local system to send or too large for some hop along the way.
`static int`	**`PORT_UNREACHABLE`** Error code indicating that the destination port given in the federated address can not be reached.
`static int`	**`REFUSED`** Error code indicating that the message could not be received by the destination or an intermediate host.

`static int`	**`SENT`** Status code indicating that the message has left the local system.
`static int`	**`TYPE_ERROR`** A final status that indicates that some type of error occurred and that the message was not delivered.
`static int`	**`TYPE_STATUS`** A non-final status indicating message transitions.

Method Summary

`void`	**`statusChanged`**`(long messageId, int code)` This method is called when an event has occurred while sending a message associated with the specified message identifier.

Field Detail

4.3.1 TYPE_ERROR

`public static final int` **`TYPE_ERROR`**

A final status that indicates that some type of error occurred and that the message was not delivered.

4.3.2 TYPE_STATUS

`public static final int` **`TYPE_STATUS`**

A non-final status indicating message transitions.

4.3.3 DOMAIN_UNREACHABLE

`public static final int` **`DOMAIN_UNREACHABLE`**

Error code indicating that the destination domain given in the federated address can not be reached. This can be due to routing problems or that the domain is unreachable.

4.3.4 HOST_UNREACHABLE

`public static final int` **`HOST_UNREACHABLE`**

Error code indicating that the destination host given in the federated address can not be reached. This can be due to routing problems or that the host is not running or accepting traffic.

4.3.5 PORT_UNREACHABLE

`public static final int` **`PORT_UNREACHABLE`**

Error code indicating that the destination port given in the federated address can not be reached. This may be because there is no message listener registered at the given port. This error is not generated for multicast or broadcast addresses.

4.3.6 EXPIRED

`public static final int` **`EXPIRED`**

Error code indicating that the message has expired in transit. This means that the message was evicted from some store-and-forward queue because it had been lying around there for too long or that the TTL of the message was exceeded.

4.3.7 REFUSED

`public static final int` **`REFUSED`**

Error code indicating that the message could not be received by the destination or an intermediate host. This can be due to protocol mismatches or security issues.

4.3.8 COMMUNICATIONS_UNAVAILABLE

`public static final int` **`COMMUNICATIONS_UNAVAILABLE`**

Error code indicating that the communications are not possible.

4.3.9 MESSAGE_TOO_BIG

`public static final int` **`MESSAGE_TOO_BIG`**

Error code indicating that message being sent is too large for the local system to send or too large for some hop along the way. This situation is unlikely to change over time.

4.3.10 INSUFFICIENT_RESOURCES

`public static final int` **`INSUFFICIENT_RESOURCES`**

Error code indicating that the message could not reach the destination because of insufficient resources someplace in the path. This may be a temporary condition.

4.3.11 CANCELED

`public static final int` **`CANCELED`**

Error code indicating that the message has been canceled.

4.3.12 SENT

`public static final int` **`SENT`**

Status code indicating that the message has left the local system. This generated when a message is delivered to its next hop by a transport.

Method Detail

4.3.13 statusChanged

`public void` **`statusChanged`**`(long messageId, int code)`

This method is called when an event has occurred while delivering a message associated with the specified message identifier. The message identifier is the same value that was returned by `MessageService.send()` when the message in question was sent.

Parameters:

`messageId` – The identity associated with the message.

`code` - Event code specifying message status. The event type can be determined from the event code value `(code >> 8 == type)`.

4.4 org.osgi.service.comm.messaging Interface Message

public interface **Message**
A message holds data and a `FederatedAddress`. A message can either be sent to a foreign host using the `MessageService.send()` or received through a registered `MessageListener`.
The address contained in the message specifies the foreign host the message was received from or shall be sent to.

Method Summary

`org.osgi.service.comm.FederatedAddress`	**`getAddress`**`()` Returns the address associated with message.
`byte[]`	**`getData`**`()` Returns the payload data carries by this message.

Method Detail

4.4.1 getAddress

`public org.osgi.service.comm.FederatedAddress` **`getAddress`**`()`

Returns the address associated with message. If the message was received through a `MessageListener` the address represents the foreign host who sent the message. If the message is passed to `MessageService.send()` the address represents the foreign host the message shall be sent to.
When a `Message` is passed to `MessageService.send()`, the `FederatedAddress` returned by this method must not be altered until the call has terminated.
Returns:
The address this message was sent from or should be sent to.

4.4.2 getData

`public byte[]` **`getData`**`()`

Returns the payload data carried by this message. For `Message` implementations that store the message data internally as a byte array, there is no need to return a copy of that array as long as it is not modified during calls to `MessageService.send()`.
Returns:
The message data.

All Page Within This Box

5 Security Considerations

5.1 Controlling Access to `MessageListeners`

With the whiteboard approach to registering `MessageListeners` some precautions must be made to ensure the intended functionality. To prevent `MessageListeners` from being called by arbitrary bundles, the security policy must be set up such that `ServicePermission` to check out `MessageListener` services from the framework registry is only given to the appropriate bundles, i.e., those that implement message delivery systems.

6 Document Support

6.1 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. OSGi, Secure Messaging, RFP-17, January 2001

[3]. OSGi, Namespace, RFC-22, July 2001

6.2 Author’s Address

Name	Johan Agat
Company	Gatespace
Address	
Voice	
e-mail	agat@gatespace.com

Name	Anders Rimén
Company	Gatespace
Address	
Voice	
e-mail	ar@gatespace.com

6.3 Acronyms and Abbreviations

6.4 End of Document