Sametime
Version 9.0.0

Sametime 9.0.0
Software Development Kit

Gateway Integration Guide

Edition Notice

Note: Before using this information and the product it supports, read the information in "Notices."

This edition applies to version 9.0.0 of IBM Lotus Sametime (program number 5725-M36) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Chapter 1. Introduction

The IBM® Lotus® Sametime® Gateway is an extensible platform, built on the IBM WebSphere® Application Server. The Sametime Gateway allows real-time collaboration communities, such as Sametime and public IM services, to share presence awareness and interact with each other.

This document, along with the samples and javadoc in the Sametime Gateway Toolkit, provide you with the information you need to build Message Handler plug-ins and event consumers in order to extend IM compliance, policy enforcement and logging requirements when federation is used to connect one local community to one or many external communities.

This document is organized as follows:

- An overview of the Sametime Gateway
- The Sametime Gateway architecture
- Extending the Sametime Gateway
- Sample extension applications

Terms to know

The following terms are used throughout this integration guide.

Table of terms and definitions used in this guide.

Term	Definition
API	Application Programming Interface
Common Event Interface (CEI)	The common event infrastructure is a core component of the WebSphere Enterprise Service Bus. The common event infrastructure provides facilities for the run-time environment to persistently store and retrieve events from many different programming environments.
Eclipse	Eclipse is an open platform for rich client development. More information about Eclipse is available at: http://www.eclipse.org
EJB	Enterprise Java Bean
Extension	Allows new capabilities to be added to the server environment.
Extension Point	Declares how functionality of a plug-in can be added on to. An extension point can be likened to an electric outlet.
IBM WebSphere® Application Server	IBM WebSphere Application Server is a Java™ 2 Enterprise Edition (J2EE) and Web services-based application server, built on open standards, that helps you deploy and manage applications ranging from simple Web sites to powerful on demand solutions. WebSphere Application Server offers a rich application deployment environment with a complete set of application services, including capabilities for transaction management, security, clustering, performance, availability, connectivity, and scalability. It is J2EE compliant and provides a portable Web deployment platform for Java components, XML, and Web services that can interact with databases and provide dynamic Web content. SIP functionality is delivered beginning with WebSphere Application Server 6.1.

	The SIP components in WebSphere Application Server have a tight integration with the existing HTTP servlet and portlet work, with which you can write a highly converged HTTP and SIP application with seamless failover provided by the Proxy Server. In WebSphere Application Server, the Web container and SIP container are converged and are able to share session management, security and other attributes. In this model, an application that includes SIP servlets, HTTP servlets, and portlets can seamlessly interact, regardless of the protocol. More information can be found at: http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp
IDE	Integrated Development Environment, for example, the Rational Application Developer or Eclipse IDEs.
ISV	Independent Software Vendor
Plug-in	An Eclipse platform feature component. More information about Eclipse plug-ins is available at: http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-overview-2005-12.pdf
Presence Awareness	Ability for a user to be notified of the online status of another user.
RTC	Real-time Collaboration
SIP	Session Initiation Protocol. Initially this protocol was defined for the creation of media sessions between users, but now the SIP protocol has been expanded to include: presence, IM, n-way chat, Voice over IP (VOIP) audio/video and file transfer. SIP closely resembles HTTP and SMTP and instantiates, terminates and modifies sessions. This makes SIP highly extensible, scalable and able to fit into a variety of architecture mechanisms. SIP, and the standards surrounding SIP, provide the mechanisms to look up, negotiate, and manage connections to peers on any network over any other protocol.
Virtual Places (VP)	An IBM proprietary protocol that is used in the Sametime system of IBM. This protocol enables real-time collaboration functionality, such as presence, IM, n-way chat and e-meetings.
XMPP	Extensible Messaging and Presence protocol. Another protocol that was created by the IETF. The XMPP protocol is very strongly based on XML streams and enables presence, IM and n-way chat. The XMPP protocol has a wide open source community that defines additional features in the protocol. The XMPP is more accepted in the some parts of the US federal government and the finance industry. The XMPP implementation will conform to RFC specifications (see RFC 3920 and 3921)

Suggested Reading

For more information, see these Web sites:

Table of suggested reading with a URL for each document.

What	Where
WebSphere Application Server 7.0 Infocenter, specifically how the Extension Registry works in WAS 7.0	http://publib.boulder.ibm.com/infocenter/wasinfo/ v7r0/index.jsp?topic=/com.ibm.websphere.nd.doc/info/ae/ae/cweb_extensions.html
WebSphere Application Server 7 Infocenter	http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp
Sametime Information Center	Included with Sametime SDK and also available at `http://publib.boulder.ibm.com/infocenter/sametime/v8r5/index.jsp`
Common Event Infrastructure (CEI)	`http://publib.boulder.ibm.com/infocenter/dmndhelp/v6rxmx/index.jsp?topic=/com.ibm.websphere.wesb.doc/doc/ccei_admin_usingCEI.html`

The Development Set-up Environment

The samples created for this guide were developed using:

- Microsoft® Windows® XP
- IBM Rational® Application Developer® 6.0.1

You can use a development tool built on the Eclipse® SDK, such as one of the Rational software development tools like Rational Application Developer 6.0.1 and beyond.

Chapter 2. The Sametime Gateway

Instant messaging (IM) has become an essential productivity tool for individuals and businesses of all sizes. IM systems are organized into communities – populations of users that access a common system. The benefit of using IM within the scope of a single community is great: users can realize quick, straightforward communication facilitated by presence information.

As IM capabilities become a ubiquitous part of the internal toolset for business computer users across all organizations, those users now wish to realize the benefits of IM in their communications with persons outside their organization or community. This is the function of the Sametime Gateway -- to allow connection of Sametime communities to other communities. These can be public communities, such as AOL® Instant Messenger, or private communities hosted by business organizations. Using a modular architecture, the Sametime Gateway is able to interoperate against the most important IM standards and implementations.

In addition to providing connectivity between different IM communities, the Sametime Gateway also provides tools to manage conversations between these communities for:

- Policy enforcement
- Logging
- Privacy

IBM as well as third-party vendors can create additional plug-ins that extend the gateway's functionality, providing, for example:

- IM compliance
- Threat protection
- Enhanced policy enforcement, logging and privacy.

What is the Sametime Gateway?

The Sametime Gateway is a fully extensible platform, built on the WebSphere Application Server. The WebSphere Application Server provides a container environment for the Gateway components, as well as infrastructure services for administration, transport, authentication and system extensions.

The Sametime Gateway

- Receives messages from one or more communities.
- Checks the legitimacy of the messages passing through it.
- Translates the message protocol, if necessary.
- Forwards the messages to their destination.

The Sametime Gateway provides for interaction between a local community and one or many external communities, as well as:

- Protocol translation
- Secure dual authentication between servers
- Adding message handler plug-ins and processing content which has be logged

What the gateway provides

The Sametime Gateway provides:

- Federation
- Open Programming model
- Cross community traffic management

Federation

Federation is the ability for users from one community to interact with users from another external community. The users on both ends know that they are not members of the same community, but otherwise should notice no perceptible difference during the course of their interaction. These different communities can be using different protocol mechanisms, such as Virtual Places protocol (VP), Session Initiation Protocol (SIP) or Extensible Messaging and Presence protocol (XMPP).

For the Sametime gateway, a community can be defined as a set of users connected by a common user directory. There are three types of communities:

- Local
- External
- Clearinghouse

You can have an unlimited number of external communities, but you can have only one local community and one clearinghouse community.

The local community is the local Sametime community.

An external community is a set of users in domains connected by a common directory and belonging to a remote company or organization.

A clearinghouse community is a federated group of users linked by an enterprise's message router. When a message contains destination domains not found elsewhere in a routing configuration, the message may be routed to a clearinghouse community if one exists. A route to a clearinghouse enables Sametime Gateway users to connect to a much wider community.

One of the benefits of federation is the one to many nature of the protocol translation that is handled by the Sametime Gateway.

The following image is a representation of the federation model:

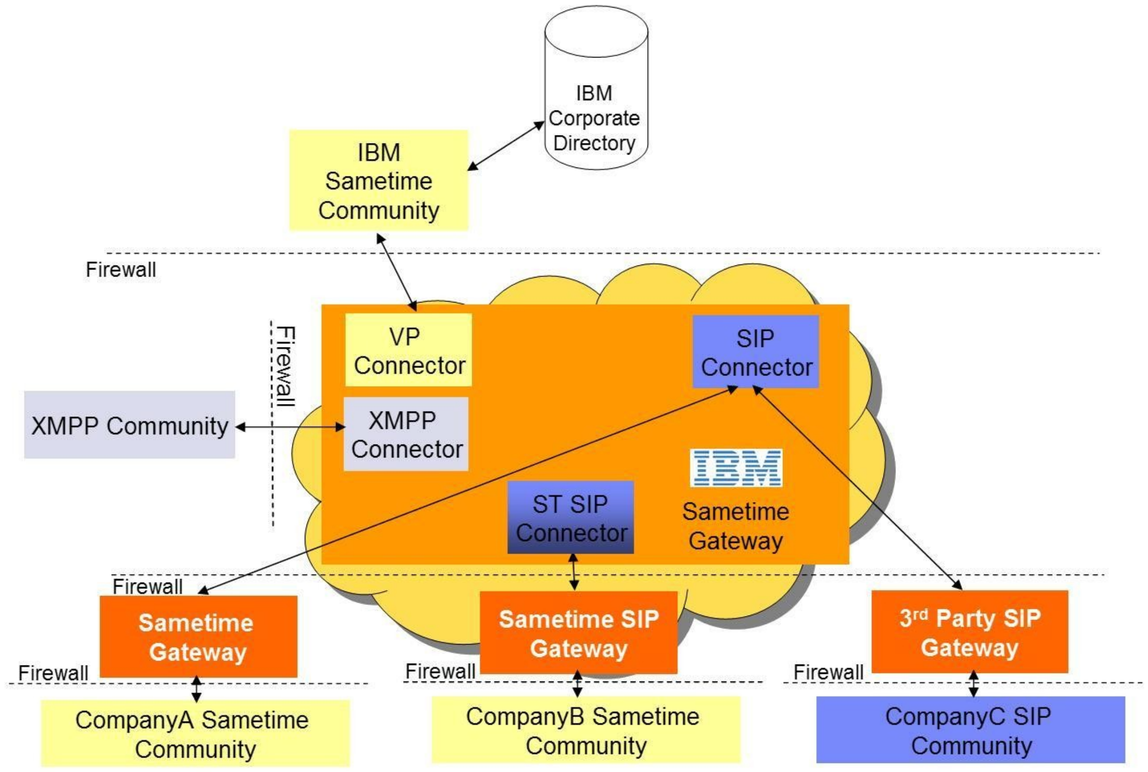


Figure 1: Sametime Federation model

Open Programming Model

The Sametime Gateway employs an open-standards plug-in framework that is built on top of WebSphere Application Server. One of the values offered by the WebSphere Application Server platform is the addition of the Eclipse extension registry framework that allows for solutions to be developed by ISVs.

This includes:

- Translation protocols. Future releases will support the addition of IM protocols as needed
- Message handler plug-ins can be added to provide additional services (all messages are passed to plug-ins for processing)

External community traffic management

The gateway also acts as the focal point for all extranet IM traffic. Message handler plug-ins will provide services such as policy enforcement, content filtering, privacy and logging requirements.

Chapter 3. Parts of the Sametime Gateway

The Sametime Gateway consists of five major components:

- WebSphere Application Server
- The Sametime Gateway core
- Translation protocols
- Message handler plug-ins
- Administration

The following image is a more detailed view of the Sametime Gateway.

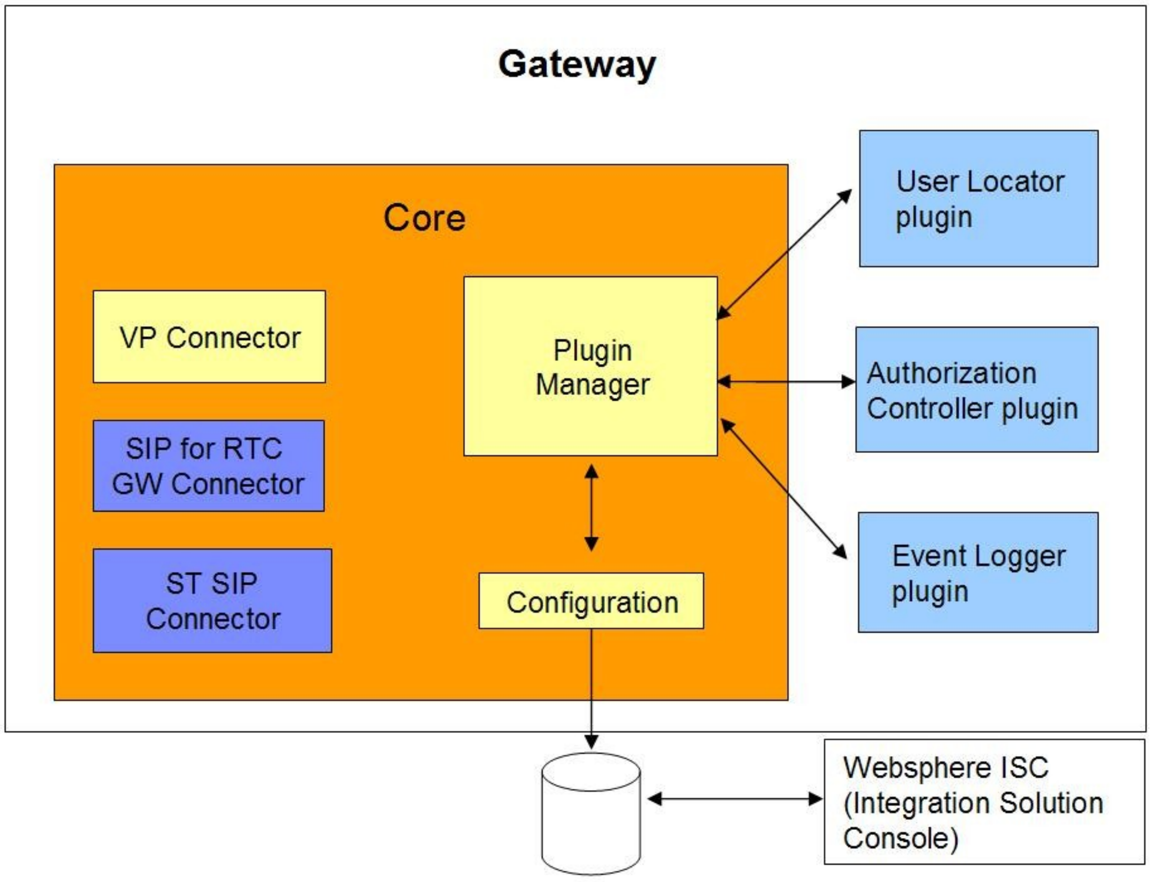


Figure 2: Sametime Gateway

WebSphere Application Server

The WebSphere Application Server provides a SIP infrastructure, Common Event Infrastructure (CEI) and extensions mechanism which are leveraged by the Sametime Gateway. The SIP Infrastructure provides scalability, clustering and failover and is leveraged by the IBM Sametime Gateway.

WebSphere Application Server supports clusters, which are groups of servers that are managed together and participate in workload management. The IBM Sametime Gateway uses clusters to provide scalability and failover. Message Handler plug-in developers should not assume that all messages will be processed by the same plug-in in the same Sametime Gateway server. When a cluster is used, identical Sametime Gateway servers including plug-ins are configured. If a Message Handler plug-in uses a data store it should be accessible from all servers in the clusters, in order to provide support for clustering. This allows plug-ins on different servers to have access to consistent data.

The Logger Message Handler plug-in uses the Common Event Infrastructure to publish Common Base Events (CBE) that describe all message handler communication between an internal and external community. ISVs can develop event consumers to process this information by receiving asynchronous event notifications or by performing queries which process historical event data from the Common Event Infrastructure persistent data store.

The WebSphere extensions mechanism is based on the Eclipse programming model for extensions. With WebSphere Application Server:

- Applications first define extension points; these are points in application operation where new features can be added.
- Developers create extensions that "plug in" to these extension points.
- At runtime, the application obtains the set of installed extensions and invokes them at the appropriate time.

For example, an extension mechanism can be employed to add additional web search engines to a search Web site. The search web site exposes a search extension point; the extensions provide access to different web search engines. This allows additional web search engines to be added without modifying the search web site. The Sametime Gateway for Message Handler plug-ins follows this model.

Creating extension requires two basic steps:

- Implementing the needed extension interface.
- Creating the plug-in descriptor file, plugin.xml, which declares the extension and allows its discovery at runtime.

You can find additional information about the WebSphere Application Server SIP Infrastructure, CEI Infrastructure and extension registry at
http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp

The Sametime Gateway core

The Sametime Gateway Core is the central Gateway component that coordinates the operation of the different supporting modules. The Core is realized as both a J2EE enterprise application, as well as Java class libraries running as WebSphere extensions.

The Sametime Core:

- Starts and manages the different translation protocols.
- Routes the gateway messages.
- Manages the communities.
- Communicates with the plug-ins.

Translation protocols

A protocol defines how software connections communicate. There are many different types of communication protocols, from TCP/IP which sets internet communication to Session Initiation Protocol.

In the Sametime Gateway, each translation protocol supports a specific protocol, and receives and transfers messages. It is a plug-in component that translates various real time protocols to a normalized internal format.

Sametime Gateway will include the following translation protocols:

Table of protocols and how they are used by Sametime Gateway.

The following translation protocol…	Communicates with
Virtual Places (VP)	• A local Sametime VP server (when present)
Session Initiation Protocol (SIP)	• Foreign domains using SIP Derivations of this translation protocol will handle the public IM networks
Sametime SIP GW – Subset of SIP	• Foreign Sametime domains that have deployed the older Sametime SIP Gateway
Other translation protocols can be added as needed, for example, Extensible Messaging and Presence Protocol (XMPP). An XMPP server like GoogleTalk.	

Message handler plug-ins

Message handler plug-ins can be added to provide additional message management features, for example access control or IM compliance. Message handler plug-ins are also able to view, modify, or even delete messages from being processed by other plug-ins or translation protocols. Message handler plug-ins can be developed internally or externally to provide the required functionality.

The following message handler plug-ins are included with the Sametime Gateway installation and will be covered in greater depth in Chapter 4, *Overview of the Sametime Gateway Architecture*:

- User locator plug-in
- Authorization controller plug-in
- Event logger plug-in

Using plug-ins to extend the Sametime Gateway

There are a variety of supported options for the deployment of Sametime Gateway extensions. Extensions used by the Sametime Gateway should be deployed as J2EE enterprise application, EAR files. This provides the best approach for initialization and lifecycle of the extension.

The following table lists the plug-ins and samples included with the integration guide.

Table of examples included with this guide.

Plug-in or sample	Purpose	Extensions
Hello World	This simple example demonstrates how to create, package and deploy a plug-in.	*PluginRegistration
Compliance Logger	Demonstrates chat transcript logging based on criteria.	*PluginImHandler *PluginSessionHandler
Presence Privacy	Demonstrates how to implement more sophisticated privacy from presence awareness.	*PluginPresenceHandler

* To save space the asterisk (*) denotes `com.ibm.collaboration.realtime.gateway.plugin`

Chapter 4. Overview of the Sametime Gateway Architecture

The previous chapter introduced Sametime Gateway features and gave you a flavor for what's necessary to extend the gateway client with your own functionality. This chapter presents an overview of the Sametime Gateway architecture including the sample message handler plug-ins and event consumers that are included with the Sametime Gateway. This guide’s source code samples leverage the major components you would frequently use in your own code.

The topics in this chapter are designed to enhance your "programmer's intuition" of how the pieces fit together. This understanding will serve you well as a foundation for your continued exploration of the possibilities the Sametime Gateway offers you and your customers.

Note At this point you should have a firm understanding of the WebSphere Application Server and Rational Application Developer. Familiarity with the Eclipse extension registry framework, added to WebSphere Application Server (WAS) 7.0 will be required to develop plug-ins. Event consumers will leverage the Common Event Infrastructure (CEI) and Common Base Event (CBE).

Architectural Goals

Whether it's a new operating system or application framework, every programmer begins learning how a system works by understanding the layers of components that ultimately deliver its functionality; it's easier to grasp the division of responsibilities of such a layered design. In this respect, Sametime Gateway is no different from any other product. The base layers provide common capabilities and the upper layers add more specialized behaviors. This chapter will introduce these layers.

The WebSphere Application Server platform serves as the foundation of the Sametime Gateway. A key benefit of this platform choice is the ability to extend the gateway by developing message handler plug-ins and event consumers. Message handler plug-ins are integrated with the gateway using the Eclipse extension framework. Event consumers process events published by the gateway to the Common Event Infrastructure.

Programmer’s Viewpoint

The benefit of using the Eclipse extension registry framework as an integration platform is the ability to welcome new functionality into the gateway using an industry standard approach. This allows message handler plug-ins to be developed independently and ISVs to add value. Each message handler plug-in is developed as a J2EE application which processes messages. They are installed into I WebSphere Application Server using the standard Administration process and then configured with the gateway administration. Once a plug-in is configured and registered with the gateway, it will be called to process messages of interest without requiring a gateway restart.

Plug-in solutions can be developed for the Sametime Gateway to manage internal to external community communications. Typically, these types of communications are more restrictive than would be allowed for internal IM community communications.

The Sametime Gateway Core and not the plug-ins themselves, converts each protocol packet into common gateway interfaces and events. This insulates the plug-ins from protocol changes and differences. Message handler plug-ins provide an integration point while the protocol packets are being processed to modify, interrogate and/or prevent messages from being delivered to translation protocols.

The logger message handler plug-in included with the Sametime Gateway uses the Common Event Infrastructure to log events. These events can be consumed later and processed by externally developed event consumers. This allows CPU and time intensive processing to be accomplished asynchronously without affecting gateway performance.

The Sametime Gateway is based on standards allowing engineers to focus on developing solutions instead of learning a new platform. They can leverage existing skills and knowledge based on WebSphere Application Server, Eclipse extension registry framework and CEI. If they don’t have this experience then IBM and external publications can provide the background and information.

Key Concepts

As you read in Chapter 2, federation is the technique used by the gateway for internal to external community communication. The gateway contains translation protocols for different protocols. Each translation protocol manages its own connection, translating the protocol packets into common gateway events which can then be processed consistently in the gateway.

The following image shows a multiple-community configuration deployment.

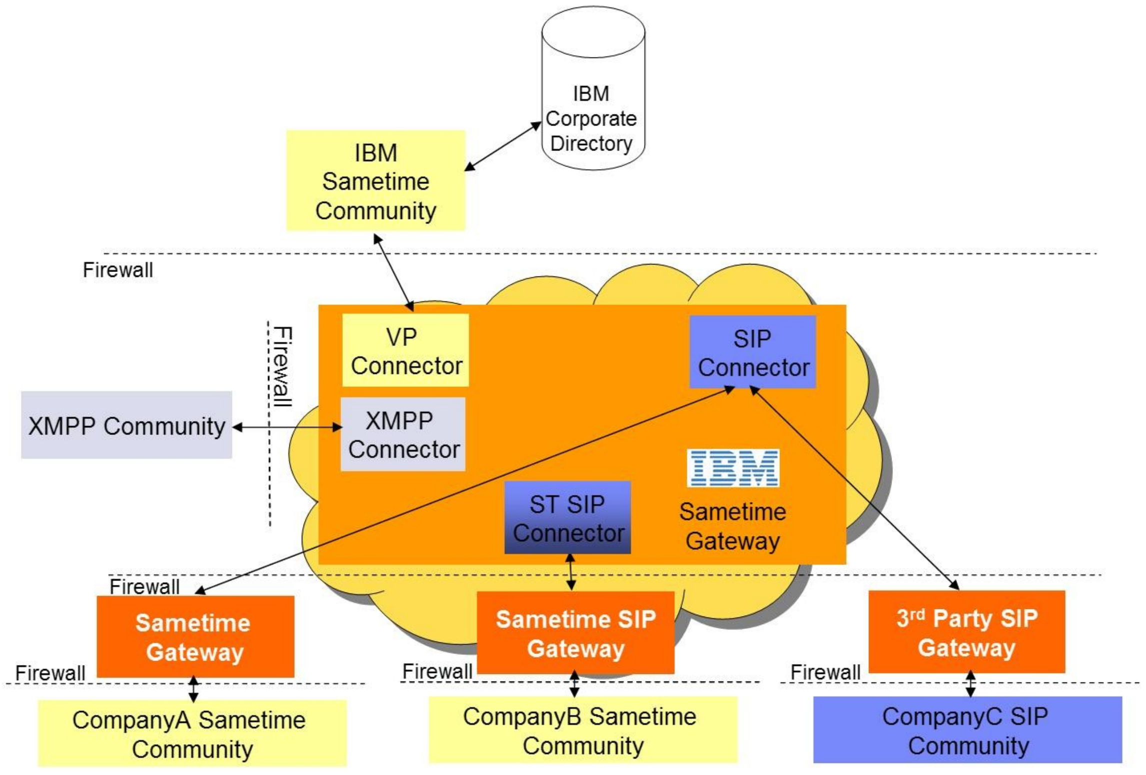


Figure 3: Sametime Gateway with several communities

In this example, the IBM Sametime Community is the internal community and the other communities are external. In most cases the community servers are behind a firewall and the Sametime Gateway and other gateways are outside the firewall, represented by the cloud in the picture, providing connectivity. The connectors in the blue boxes above represent components using the SIP (or SIP dialect) protocol. The light beige boxes represent connector or Community components using the VP protocol. The orange boxes represent gateways.

In this release, which only supports federation, the SIP, SIP dialect, and XMPP protocols can be used to connect an internal Sametime community and external community. This is displayed as the black lines from within the Sametime Gateway to the other gateways for external communities. No connecting lines are displayed between the connectors within the Sametime Gateway because the gateway core manages which translation protocols are used. This is defined by the gateway administrator when communities are defined. An internal Sametime community and external Sametime community cannot be directly connected within a single IBM Sametime Gateway. Instead, each Sametime community must use a gateway to administer the connection and its security.

The diagram above displays many scenarios. We will discuss two in detail. The first is on the left bottom and has an internal IBM Sametime Community which uses the Sametime Gateway to connect to another Sametime Gateway to access an external Sametime Community. The second is in the center bottom and has an internal Sametime community which uses the Sametime Gateway to connect to Sametime SIP Gateway to access an external Sametime Community. .

Translation protocols have been discussed here, to provide a complete overview, but they can not be developed externally or extended in this release.

Once an internal community has access to other external communities a new level of management, control and compliance must be provided. This is accomplished in the Sametime Gateway with message handler plug-ins. Some plug-ins are developed by IBM and are included in the gateway, while other plug-ins can be developed externally, then installed and configured separately as shown in the image below. The Plug-in Manager and Configuration are notified when message handler plug-ins are started and stopped.

The following image shows a representation of the Sametime Gateway model and where WebSphere Application Server fits in.

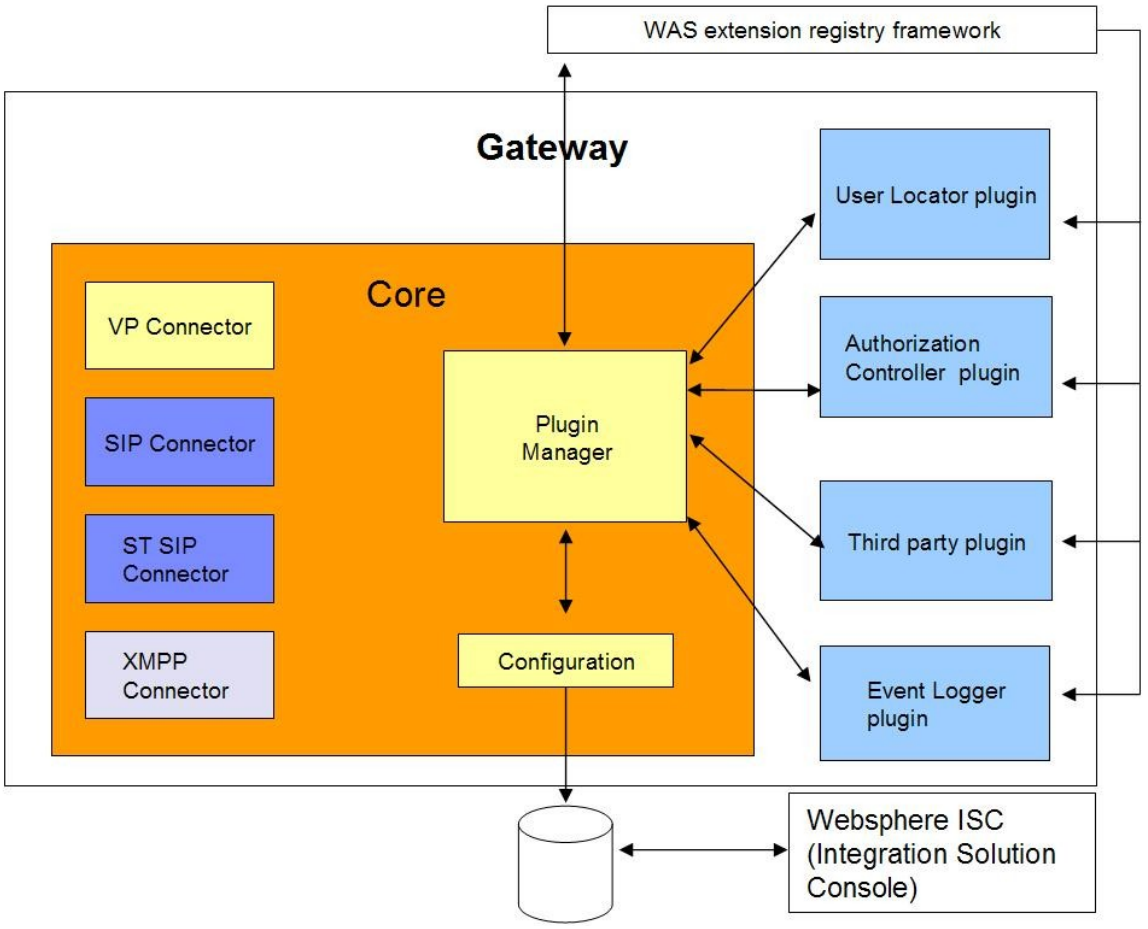


Figure 4: Sametime Gateway with WAS framework depicted

Third parties add value by developing message handler plug-ins that implement one of the interfaces associated with presence, IM or both. The Sametime Gateway and translation protocols convert the protocol packets into Common Gateway Events and call the plug-ins, which implement the specific interfaces.

Message handler plug-ins are called after the incoming protocol process and before the outgoing protocol process. Because of this, plug-ins have the opportunity to change what happens in the overall process. For example, if the status is changed from success to failure this would prevent the outgoing protocol process call. In addition, contents of the parameters could be changed, masking parts of a message.

In addition to developing plug-ins, third parties can add value by developing event consumers. An event consumer application uses the Java Message Service (JMS) interface to subscribe to the destination configured by CEI. The Logger plug-in included with the Sametime Gateway publishes events to CEI as configured by the administrator. The JMS message contains the log message, which can be retrieved as a Common Base Event including the Sametime Gateway specific data. Event consumers should be used for asynchronous processing when currency and modifying message content or flow is not required.

The rest of this chapter will provide details on the key component layers, extension points, and classes / interfaces that will be part of your plug-ins or event consumers.

Extension Registry Framework

The Eclipse extension registry framework, used by WebSphere Application Server, follows a plug-in system similar to a standard Eclipse plug-in. Applications are extensible by plug-ins that plug into the application's declared extension points.

The Sametime Gateway's extension point is:

```
com.ibm.collaboration.realtime.gateway.mhplugin
```

Message handler plug-ins created by IBM or external vendors must use this extension. Plug-ins can be installed, started, stopped and removed without forcing the Sametime Gateway to restart in order to incorporate these changes.

The plugin.xml file, which is part of the plug-in deployment files, defines how the plug-in works with the platform runtime.

You can find additional information about the WebSphere Application Server implementation at:

```
http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?
topic=/com.ibm.websphere.nd.doc/info/ae/ae/cweb_extensions.html
```

Find additional information about the Eclipse Plug-in Architecture at:

```
http://www.eclipse.org/articles/Article-Plug-in-
architecture/plugin_architecture.html
```

Summary of Components

This section briefly describes some of the key components discussed in the previous section.

Message Handler Interfaces and Events

Real Time Collaboration is implemented by multiple protocols. The Sametime Gateway includes translation protocols which translate from various real time protocols to a normalized internal format. The internal format consists of interfaces and event classes. Interfaces define a set of methods and parameters are passed to each method as a specific event class. The interfaces and events insulate plug-ins from protocol differences and changes.

This release will include interfaces for IM and presence. The following table lists the interfaces and events:

Table listing interfaces and events.

Feature	Interface	Event	EventStatus
Presence	PluginPresenceHandler	PresenceEvent	EventStatus
IM	PluginSessionHandler	SessionEvent	SessionEventStatus
IM	PluginImHandler	ImEvent	SessionEventStatus

Interfaces contain two types of methods:

- An action method which performs the action
- A corresponding response method which provides the status of the action.

For example, void subscribe(PresenceEvent subEvent) has a matching void onSubscribe(EventStatus subEvent). Action methods are passed an **Event** which contains all relevant information including an **EventStatus**. The EventStatus can be set to a failure code which allows a plug-in to filter a message and prevent it from being processed by other plug-ins and dispatched to the destination translation protocol.

The PluginRegistration defines the message handler plug-in life cycle and is extended by:

- PluginPresenceHandler
- PluginSessionHandler
- PluginImHandler

Both Event and EventStatus contain a SessionID and RequestID to define the association between related methods calls. SessionID is used to define an IM or presence session. This allows a message handler to associate multiple interface method calls to the same top level action. For example, the same SessionID will be used for the startSession, onStartSession, instantMessage(s), onInstantMessage(s), endSession and onEndSession for an IM conversation between two users. A chat logging message handler plug-in would use the SessionID to determine which IM messages are related to a conversation between two users. SessionIDs are unique.

The RequestID is also unique and is used to identify the request and response pair within a session. In the example above for SessionID, startSession and onStartSession would have one unique RequestID, instantMessage and onInstantMessage have another and, endSession and onEndSession would have a third.

Managing Message Handler Plug-ins

The Sametime Gateway Core manages the Sametime Gateway message handler plug-ins. Message handler plug-ins extend the message management features of the Sametime Gateway by providing solutions which process, filter, modify and log communications between a local and external community or communities. For this section the term plug-in manager is used to define the Sametime Gateway Core components which manage, track and call the message handler plug-ins. The plug-in manager will manage both plug-ins distributed and configured with the Sametime Gateway and plug-ins developed by third parties to provide additional value.

All plug-ins are initially registered as disabled. An administrator must enable and configure a plug-in before it will be used by the Sametime Gateway. The WebSphere Application Server extension framework notifies the plug-in manager when J2EE modules which contain plug-ins are started or stopped. Each J2EE module might contain more then one plug-in. Only plug-ins that provide an extension for the `com.ibm.lab.plugins.gw.gwplugin` extension point will be registered.

The PluginRegistration interface defines the plug-in life cycle. The life cycle includes an init() method to verify a plug-in is initialized and a terminate() method which notifies the plug-in to perform cleanup related to the gateway. When a plug-in is called by the gateway to initialize, it will be passed any custom properties defined by the gateway administrator for this plug-in. It will also be passed a callback method which is called by the plug-in when initialization is complete and the plug-in is ready to process messages. Plug-ins are notified to initialize when a plug-in is enabled or the Sametime Gateway is started.

The customPropertiesChanged() method is called when a gateway administrator changes properties for this plug-in.

Plug-ins are notified to perform any cleanup related to the gateway when a plug-in is disabled, removed, or the Sametime Gateway is shutdown. For example, a message handler plug-in may track IM chat sessions for chat logging purposes. If the translation protocol does not close all IM sessions when the Sametime Gateway is shut down, the plug-in can release any gateway related resources at this time.

The flow of messages to message handler plug-ins

The Sametime Gateway translation protocols translate various real time protocols to a normalized internal format. The gateway core controls the flow of messages through the gateway including the Plug-in Manager. Figure 3 illustrates an IBM Sametime Gateway with five message handler plug-ins. The blue arrows show messages entering the gateway and being processed starting with #1. The red arrows show messages leaving each plug-in, Plug-in Manager and gateway ending with #11. Three of the plug-ins are developed by IBM and distributed with the Sametime Gateway. Two plug-ins, labeled external, are developed by third parties.

The following image represents the flow of messages to the message handler plug-ins.

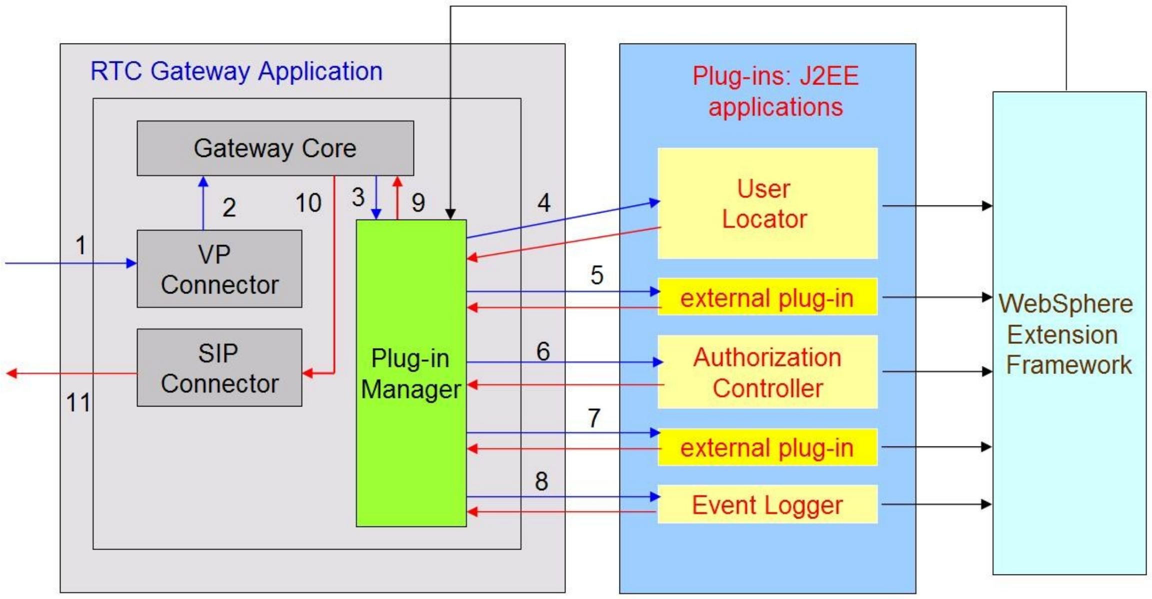


Figure 5: Message flow through message handler plug-in

It is important to note that not every message will be delivered to every plug-in. A Plug-in is only called by the manager for the interfaces it implements. he interface defines the type of messages a plug-in will process. For example, User locator and Authorization controller do not implement the PluginImHandler; they will never be called for Instant Messages since access checking is done at the start of an IM session only, and therefore will not be called per message exchange.

The Event Logger must have access to all messages and it will implement all interfaces. The Event Logger is also a “mandatory” plug-in. A mandatory plug-in is always invoked regardless of whether a previously run message handler completed the task successfully or not. A plug-in is mandatory when the “Run this message handler regardless of the status of previous message handlers” flag is checked in the Message Handlers property page of the Gateway ISC. More information on the specific interfaces will be described below.

Figure 5, above, provides a high level description of the flow but not the logic behind the plug-in manager and the Sametime Gateway core. This logic allows the message handler plug-ins to provide the functionality required by filtering, modifying and logging messages. This logic will be explained using pseudo, or bogus, code; specific sections will be discussed in more detail.

The pseudo code below is valid for any message handler interface and uses **subscribe** and **onSubscribe** methods from the PluginPresenceHandler for this example. It is annotated with the numbers used in Figure 5 and in Figure 6, below. Figures 5 and 6 depict the flow of messages between the message handler plug-ins and the gateway core and how the plug-ins can affect this processing.

```
Connector A creates a PresenceEvent and sets status to success and assigns it a unique request ID (#2)
Connector A calls subscribe operation on the Core passes PresenceEvent {
    Core calls the plug-in manager subscribe operation (#3) {
        while status == success - loop through conditional plug-ins (#4, #5, #6)  {
            call subscribe handler which sets status in PresenceEvent.
            if status != success break (A)
        }

        loop through unconditional plug-ins (#7, #8) {
            call subscribe handler and never modify status in PresenceEvent.
            <Don't check status>
        }
    }

    if status != success (B)  {
        Core calls onSubscribe handler on Connector A
    }

    If status == success (#10) {
        Core calls subscribe handler on Connector B and return
    }
}
```

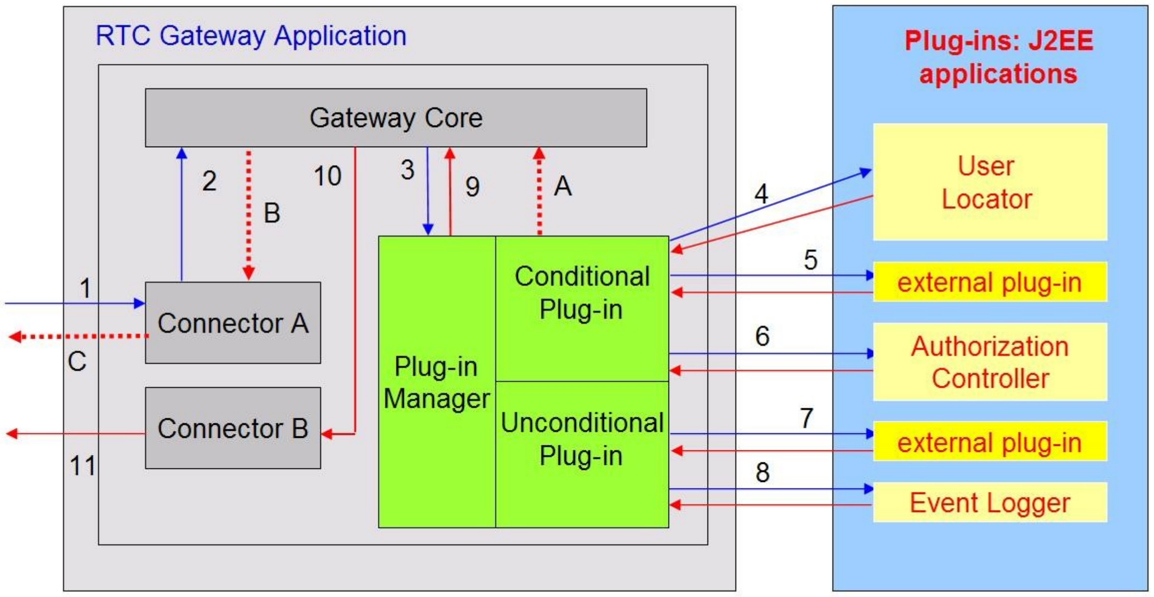


Figure 6: Handling the subscribe action

Since **asynchronous** processing could be performed by **Connector B** to process the **subscribe operation** a separate onSubscribe method is used to return the status to **Connector A**. The events passed to subscribe and onSubscribe will contain the same Request ID. This is described in the pseudo code below.

```
Connector B calls onSubscribe operation on the Core passing PresenceEventStatus {
    Core calls the plug-in mgr onSubscribe operation  (#3) {
        loop through conditional plug-ins (#4, #5, #6) {
            call onSubscribe handler with PresenceEventStatus ()
            <Don't check status>
        }

        loop through unconditional plug-ins (#7, #8)  {
            call onSubscribe handler with PresenceEventStatus ()
            <Don't check status>
        }
    }

    call onSubscribe handler with PresenceEventStatus on Connector A (#10)
}
```

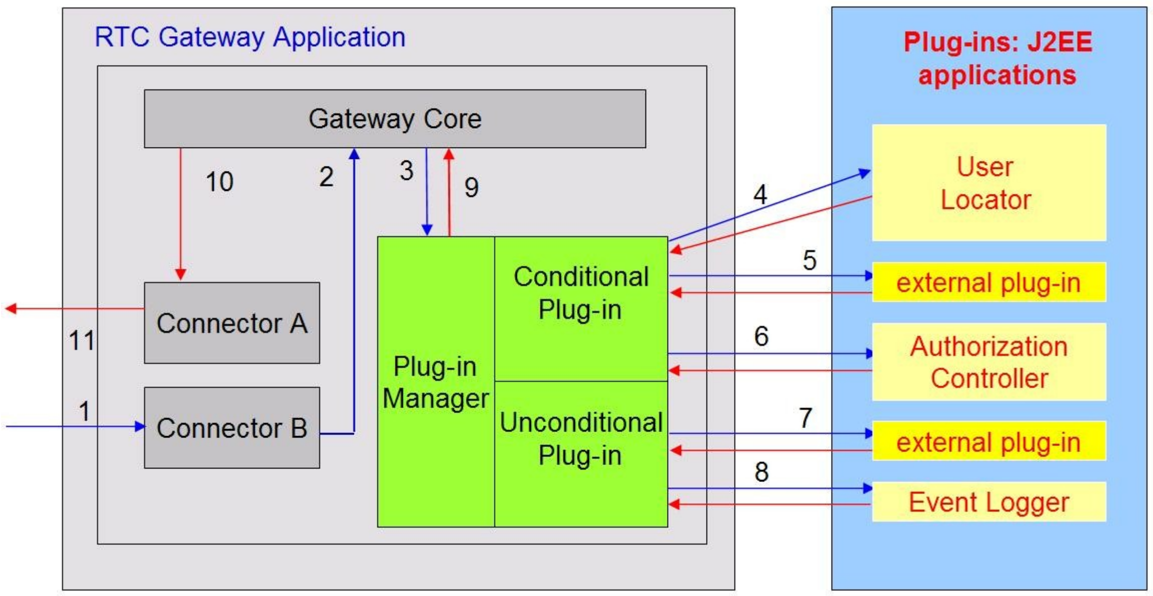


Figure 7: Handling the onSubscribe response

Message Handler Plug-ins included in this release

The Sametime Gateway includes three message handler plug-ins, which leverage the same foundation as externally developed plug-ins. The goal of the Sametime Gateway architecture is to provide greater extensibility and integration.

The following table lists the message handler plug-ins included with the Sametime Gateway and their function:

Table listing the message handler plug-ins included with Sametime Gateway

Plug-in	**Function**
User Locator	Determines routing to target user. This plug-in must be the first plug-in and is conditional.
Authorization controller	Allow or disallow a message based on configuration. This plug-in is conditional.
Event logger	Publish information about gateway events. Maintain statistics about gateway traffic.
Other message handler plug-ins can be added as needed either by IBM or by ISVs	

Message Handler properties

The gateway administrator is responsible for defining four properties for each Message Management Plug-in. Three of the properties used directly by the Plug-in Manager are:

- Enable or disable the plug-in,
- Order
- Whether a plug-in is always called

Only plug-ins that are enabled will participate in message processing. Transition between states causes the appropriate life cycle methods to be called. Each message is processed in the plug-in order defined by the administrator.

The User locator must be first in order to populate the required fields in the common gateway event. Third parties should **not** replace the User locator plug-in in this release.

By convention, the Event Logger plug-in should be last so it can log any modifications performed by all preceding plug-ins. Changing the order of the logger might cause modifications made by plug-ins ordered later in the list to be missed. The gateway administration does not enforce any order constraints but will document the two recommendations above.

All other plug-ins should be ordered to maximize performance and interactions between plug-ins. For example, a plug-in which can quickly perform access control should be ordered before a plug-in which performs processing because if access is denied then in many cases processing should be skipped.

In addition to order and enabled state, the administrator can also set a property which defines whether the plug-in is always called or only called if the previous plug-in succeeds. Plug-ins which are used for compliance or logging are normally always called because it is important to log all messages irrelevant of success or failure. Plug-ins which provide threat protection, SPIM (spam over instant messaging) or access control are usually conditional because they have no need to process messages which have already failed. The plug-in manager provides this control for conditional plug-ins by checking the return status from each method.

Besides affecting which conditional plug-ins participates, a failure status will be processed by the Gateway Core and appropriate translation protocol. For example referring to figure 3, if the Authorization controller returns a failure status then the second conditional external plug-in (#7) will not be called but and the Logger (#8) will be called. The failure status will be returned to the Gateway Core (#9) and then to the translation protocol (#10). he translation protocol will convert the failure into the appropriate protocol packet (#11).

Plug-in custom properties are the last property a gateway administrator can define. These properties are specific to each plug-in, passed to the appropriate plug-in during the life cycle init method but not evaluated by the Plug-in Manager. Properties are passed as name value pairs and should be documented by each Message Management Plug-in. Custom properties are optional provide a mechanism to pass arguments.

Methods vs. “on” Methods

This topic was quickly introduced in the section on Message Handler Interfaces and Events when RequestIDs were discussed. The interfaces are designed to allow for asynchronous processing by translation protocols and status information is returned in response methods instead of action method. This was also discussed in the pseudo code, Figure 4 and Figure 5 where Connector A sends an action to Connector B, like subscribe. Some translation protocols process the request synchronously while others will process asynchronously. Once Connector B finishes processing subscribe, it will then call the onSubscribe method on the gateway core. Irrelevant of success or failure, onSubscribe will always be delivered back to the plug-in manager and originator translation protocol. The RequestID used in an action and the corresponding response will always be consistent and unique.

Message Handler plug-ins modify and filter messages

Message handler plug-ins can log, modify and filter messages. They are passed the message content processed by translation protocols between an internal and external community. They can log and process information using any of the accessor methods on the Event or EventStatus for IM or presence. More information on logging will be discussed later in the section on Logging. Only the action methods, like subscribe, are passed an Event which provides write access. Unconditional plug-ins should never modify the EventStatus contained in the Event because previous plug-ins have processed this Event and assume the EventStatus will not change.

The Event class and corresponding derived classes provide write access to action methods. In this section we will discuss the two common operations but more complex options are possible. The first is modifying the content of a message. This is accomplished by a message handler plug-in implementing the instantMessage method on PluginImHandler. The setMessage method on ImEvent can be used to modify the content.

The other common operation is filtering/preventing an action, like subscribe, by a message handler plug-in returning a failure status. As described in the pseudo code, conditional plug-ins which return a failure status prevent other conditional plug-ins and the destination translation protocol from being invoked. Then the gateway core calls the source translation protocol's "on" method with a failure status. For example, an access control message handler plug-in could return a failure for the subscribe or notify method to prevent presence notifications/status between two users.

Consuming Events published by the Sametime Gateway

The Common Event Infrastructure (CEI) provides facilities for the run-time environment to persistently store and retrieve events from many different programming environments. Events are represented using the Common Base Event (CBE) model which is a standard XML-based format that defines the structure of an event.

- Event sources store information
- Event consumers retrieve events.

The Sametime Gateway contains an Event logger message handler plug-in which is an event source. An event consumer is only provided as a sample and will be discussed in latter sections.

An event source creates an event object whenever something happens that either needs to be recorded for later analysis or which might require additional work. The Event logger message handler plug-in publishes Sametime Gateway specific information as ExtendedDataElement. ExtendedDataElement elements record application and business specific information.

An event consumer is an object which receives asynchronous event notifications or queries and processes historical event data from the persistent data store. These events are Java objects which can be interrogated using the Common Base Event interface. Asynchronous event notifications are received using the Java Messaging Service (JMS) interface to subscribe to a queue or topic. The Event Access interface is used to query events from the persistent data store. A sample of each event consumer is included in the SDK.

More information about Common Event Infrastructure, Common Base Events and Event Consumers can be found in the:

- WebSphere Application Server Version 7.0 InfoCenter:

  ```
  http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/topic/com.ibm.websphere
  .base.doc/info/aes/ae/ctrb_cei.html
  ```

- WebSphere Enterprise Service Bus InfoCenter:

  ```
  http://publib.boulder.ibm.com/infocenter/dmndhelp/v6rxmx/topic/com.ibm
  .websphere.wesb.doc/doc/ccei_admin_usingCEI.html
  ```

Event Source: Logger message handler plug-in

The Sametime Gateway contains the Event logger message handler plug-in which is the event source responsible for publishing IM and presence information. This allows you to trace how the gateway processes messages. It uses the Common Event Infrastructure as the standard for publishing this information and each information point is published as a CommonBaseEvent.

Each CommonBaseEvent is composed of standard fields and ExtendedDataElements. The standard fields contain information about the event which allows event consumers to filter and skip irrelevant events. An element can contain ancillary information and is composed of:

- A name field
- A type field
- The value field

In the case of the Event logger, a CommonBaseEvent is recorded when

- A session starts and stops
- An instant message is sent
- A presence subscription is created or released
- A presence notification takes place

This corresponds to each time a method on an interface is called and the ExtendedDataElements reflects the parameters for those methods.

Event Consumers

When IM and presence is enabled between an internal and external community, logging can become an important requirement. In some cases, only basic logging of all actions and responses is required. Advanced logic can also be applied to logging by providing actions, notifications and selective recording when specific criteria or rules are met. These features are not included with the Sametime Sametime Gateway; third parties can provide this advanced logic by providing message handler plug-ins or event consumers.

For most logging solutions, there is no difference between message handler plug-ins and event consumers because both are provided with the same information. The benefit of using an event consumer is that processing is not inline in the gateway potentially affecting performance and events are persisted. Event consumers should be used for asynchronous processing when immediate processing, modifying message content or flow is not required. The SDK includes two event consumer samples.

The Sametime Gateway can be deployed in a cluster to provide failover or improve scale. Message handler plug-ins must be developed to support clusters as described in the "WebSphere Application Server" section of Chapter 3. Event consumers are not affected by Sametime Gateways deployed in a cluster.

Sametime Gateway ExtendedDataElements

The information logged for each method is defined by an event (for example, SessionEvent or SessionEventStatus). The data is logged using a flat representation and doesn't expose class hierarchy or containment. No differentiation is made for data that is:

- Stored on the superclass or derived class. For example, ImEvent extends Event.
- Contained in another class. For example, Event contains an EventStatus.

All relevant information for each method/event pair is published. Tables 1, 2 and 3 and the javadoc can be used to parse and understand the ExtendedDataElement.

In order to understand the ExtendedDataElement entries published by the Event logger you need to understand how the values contained relate to the methods on the interfaces. The elements named RtcGatewayEventType and RtcGatewayEventDirection are the keys used to identify the interface and method. These are numeric values, however, and the relationship is not clear from parsing the ExtendedDataElement alone.

The table below is included to give you a map to relate these values to the interface and method call that CommonBaseEvent reflects. The javadoc and this guide included with the SDK fully defines all the parameters and circumstances in which the method is called and can be consulted for detailed information on the other elements and how this call relates to what processing is going on in the Gateway.

Table listing the CommonBaseEvent interfaces, methods, and events

Event Type	Event Direction	Interface	Method	Event
7	0	PluginSessionHandler	endSession	SessionEvent
6	0	PluginSessionHandler	startSession	SessionEvent
5	0	PluginPresenceHandler	Fetch	PresenceEvent
4	0	PluginImHandler	instantMessage	ImEvent
3	0	PluginPresenceHandler	Notify	PresenceEvent
2	0	PluginPresenceHandler	Unsubcribe	PresenceEvent
1	0	PluginPresenceHandler	Subscribe	PresenceEvent
7	1	PluginSessionHandler	onEndSession	SessionEventStatus
6	1	PluginSessionHandler	onStartSession	SessionEventStatus
5	1	PluginPresenceHandler	onFetch	EventStatus
4	1	PluginImHandler	onInstantMessage	SessionEventStatus
3	1	PluginPresenceHandler	onNotify	EventStatus
2	1	PluginPresenceHandler	onUnsubcribe	EventStatus
1	1	PluginPresenceHandler	onSubscribe	EventStatus

The following two tables define constants used in the javadoc for `com.ibm.collaboration.realtime.gateway.event.Event` and `com.ibm.collaboration.realtime.gateway.plugin.EventLogType`. These tables are useful when you use the javadoc to parse the event log.

Table listing RtcGatewayEventType constants

RtcGatewayEventType	Constant
CANCEL_SUBSCRIBE_TYPE	8
END_SESSION_TYPE	7
START_SESSION_TYPE	6
FETCH_EVENT_TYPE	5
IM_EVENT_TYPE	4
NOTIFY_EVENT_TYPE	3
UNSUBSCRIBE_EVENT_TYPE	2
SUBSCRIBE_EVENT_TYPE	1

Table listing EventLogType constants

EventLogType	Constant
EVENT_DIRECTION_ACTION_METHOD	0
EVENT_DIRECTION_ON_METHOD	1

The naming convention used for Sametime Gateway ExtendedDataElements begins every log value with "RtcGateway" followed by a label describing the value which matches the getter method defined in the javadoc.

The following table logs similar values.

Table listing ExtendedDataElements log values

Log Value	JavaDoc	Class
RtcGatewaySenderURI	getOriginatorUser	com.ibm.collaboration.realtime.gateway.event.Event
RtcGatewayReceiverURI	getDestinationUser	com.ibm.collaboration.realtime.gateway.event.Event
RtcGatewaySenderCommunity	getOriginatorCommunity	com.ibm.collaboration.realtime.gateway.event.Event
RtcGatewayReceiverCommunity	getDestinationCommunity	com.ibm.collaboration.realtime.gateway.event.Event
RtcGatewayMessageContent	getMessage	com.ibm.collaboration.realtime.gateway.event.ImEvent
RtcGatewaySessionTerminationReason	getTerminateReason	com.ibm.collaboration.realtime.gateway.event.SessionEvent
RtcGatewayEventStatus	getStatus	com.ibm.collaboration.realtime.gateway.event.EventStatus
RtcGatewayEventReason	getReason	com.ibm.collaboration.realtime.gateway.event.EventStatus

Each ExtendedDataElements element for the Sametime Gateway is composed of a name field for the name of the element, a type field, for the type of the element, and the value field, which has the actual value. The type is defined in the `org.eclipse.hyades.logging.events.cbe` package as an ExtendedDataElement.TYPE. Currently INT and STRING are being used.

The table below contains some examples of the ExtendedDataElement.TYPEs.

Table listing ExtendedDataElement.TYPE values

Name	ExtendedDataElement.TYPE	Example value
RtcGatewayEventDirection	TYPE_INT	1
RtcGatewaySenderCommunity	TYPE_STRING_VALUE	IBM

Sample Log

Two log snippets were generated using the sample message driven bean that writes output published from the Event logger to the RTCGWServer trace log file, app_server_root\profiles\RTCGW_Profile\logs\RTCGWServer\trace.log. These snippets are broken into reviewable chunks for the purpose of this document in the sections, "IM Chat Session: PluginSessionHandler & PluginImHandler" and "Presence: PluginPresenceHandler", later in this chapter.

The first log snippet describes an IM Chat Session and the second snippet describes a user being added to a buddylist. The same RtcGatewaySessionID is used in each snippet because each log defines a unique session between two users.

The RtcGatewayRequestID is used to associate methods and "on" methods (i.e. startSession and onStartSession) and each pair will use the same requestID.

Before reviewing the logs, you should understand how one Common Base Event should be parsed and evaluated.

```
IBM Sametime Gateway RtcGatewayLoggerEvent Creation Time: 2006-10-
08T13:50:21.187+08:00
RtcGatewayEventDirection: 0
RtcGatewayEventType: 6
RtcGatewaySenderURI: user709@ibm.com
RtcGatewayReceiverURI: user409@lotus.com
RtcGatewaySenderCommunity: lwpps2
RtcGatewayReceiverCommunity: lwpst2
RtcGatewaySessionID: 9.181.65.251_1160286237500_32
RtcGatewayRequestID: 9.181.65.251_1160286237500_34
RtcGatewayEventStatus: 0
RtcGatewayEventReason:
RtcGatewaySessionDescription:
RtcGatewaySessionTerminationReason:
```

The following steps outline one way to parse and evaluate a CBE.

Table listing the steps involved in parsing and evaluating a CBE.

Step	Action
1	Determine the Interface, Method and Event using the RtcGatewayEventDirection and RtcGatewayEventType. **Note**: A value of 0 and 6 respectively signifies a **PluginSessionHandler method:** startSession **event:** SessionEvent
2	Use the javadoc and this guide to review when startSession is called and the data contained in the SessionEvent.
3	RtcGatewaySenderURI and RtcGatewayReceiverURI map to the methods getOriginatorUser and getDestinationUser defined on the Event base class.
4	RtcGatewaySenderCommunity and RtcGatewayReceiverCommunity map to the methods getOriginatorCommunity and getDestinationCommunity on the Event base class.
5	The log does not preserve hierarchy • RtcGatewaySessionID and RtcGatewayRequestID map to methods on the Event base class. • RtcGatewayEventStatus and RtcGatewayEventReason map to methods contained in EventStatus.

The sample Logger Event Consumer is responsible for the output above. This information is written using java.util.logging.Logger. In addition to the Sametime Gateway information, each line also contains date, time and class information
For example:

```
[10/8/06 13:50:21:859 CST] 0000003d MessageLogger 3
com.ibm.collaboration.realtime.sample.eventconsumer.messagelogger.MessageLoggerService
logEvent
```

The complete line in trace.log would appear as:

```
[10/8/06 13:50:21:859 CST] 0000003d MessageLogger 3
com.ibm.collaboration.realtime.sample.eventconsumer.messagelogger.MessageLoggerService
logEvent RtcGatewayEventDirection: 1
[10/8/06 13:50:21:859 CST] 0000003d MessageLogger 3
com.ibm.collaboration.realtime.sample.eventconsumer.messagelogger.MessageLoggerService
logEvent RtcGatewayEventType: 6
```

The sample Logger Event Consumer only writes information when diagnostic trace is enabled. Review the Sametime Gateway help in the Sametime Information Center for more information on "Setting a diagnostic trace". Logging for the samples must be enabled and set to All Message and Trace Levels for `com.ibm.collaboration.realtime.sample.`

IM Chat Session: PluginSessionHandler & PluginImHandler

The events published for an IM Chat Session have a specific order.

1. A session always begin with a startSession event specifying the sender and receiver URI and community.
2. This is followed by an onStartSession.

3. A number of instantMessage and onInstantMessage events will then be logged, one for each message.
4. The sender and receiver URI and community will flip as each user sends a message.
5. An endSession event is logged
6. An onEndSession is logged when a user closes a chat session.

Depending on the processing by other plug-ins, an "on" event might not be logged. When this condition occurs, the action event will contain a failure status.

The following code snippet describes : user709@ibm.com initiating an IM Chat Session with user409@lotus.com

```
START_SESSION_TYPE
PluginSessionHandler    method: startSession          event: SessionEvent

IBM Sametime Gateway RtcGatewayLoggerEvent Creation Time: 2006-10-
08T13:50:21.187+08:00
RtcGatewayEventDirection: 0
RtcGatewayEventType: 6
RtcGatewaySenderURI: user709@ibm.com
RtcGatewayReceiverURI: user409@lotus.com
RtcGatewaySenderCommunity: lwpps2
RtcGatewayReceiverCommunity: lwpst2
RtcGatewaySessionID: 9.181.65.251_1160286237500_32
RtcGatewayRequestID: 9.181.65.251_1160286237500_34
RtcGatewayEventStatus: 0
RtcGatewayEventReason:
RtcGatewaySessionDescription:
RtcGatewaySessionTerminationReason::

START_SESSION_TYPE:
PluginSessionHandler    method: onStartSession   event: SessionEventStatus

IBM Sametime Gateway RtcGatewayLoggerEvent Creation Time: 2006-10-
08T13:50:21.781+08:00
RtcGatewayEventDirection: 1
RtcGatewayEventType: 6
RtcGatewaySessionID: 9.181.65.251_1160286237500_32
RtcGatewayRequestID: 9.181.65.251_1160286237500_34
RtcGatewayEventStatus: 0

IM_EVENT_TYPE
PluginImHandler         method: instantMessage   event: ImEvent

IBM Sametime Gateway RtcGatewayLoggerEvent Creation Time: 2006-10-
08T13:50:23.281+08:00
RtcGatewayEventDirection: 0
RtcGatewayEventType: 4
RtcGatewaySenderURI: user709@ibm.com
RtcGatewayReceiverURI: user409@lotus.com
RtcGatewaySenderCommunity: lwpps2
RtcGatewayReceiverCommunity: lwpst2
RtcGatewaySessionID: 9.181.65.251_1160286237500_32
RtcGatewayRequestID: 9.181.65.251_1160286237500_35
```

```
RtcGatewayEventStatus: 0
RtcGatewayEventReason:
RtcGatewayMessageContent: How is the weather
```

IM_EVENT_TYPE
PluginImHandler **method:** onInstantMessage **event:** SessionEventStatus

```
IBM Sametime Gateway RtcGatewayLoggerEvent Creation Time: 2006-10-
08T13:50:23.734+08:00
RtcGatewayEventDirection: 1
RtcGatewayEventType: 4
RtcGatewaySessionID: 9.181.65.251_1160286237500_32
RtcGatewayRequestID: 9.181.65.251_1160286237500_35
RtcGatewayEventStatus: 0
RtcGatewayEventReason:
```

IM_EVENT_TYPE
PluginImHandler **method:** instantMessage **event:** ImEvent

```
IBM Sametime Gateway RtcGatewayLoggerEvent Creation Time: 2006-10-
08T13:55:25.406+08:00
RtcGatewayEventDirection: 0
RtcGatewayEventType: 4
RtcGatewaySenderURI: user709@ibm.com
RtcGatewayReceiverURI: user409@lotus.com
RtcGatewaySenderCommunity: lwpps2
RtcGatewayReceiverCommunity: lwpst2
RtcGatewaySessionID: 9.181.65.251_1160286237500_32
RtcGatewayRequestID: 9.181.65.251_1160286237500_36
RtcGatewayEventStatus: 0
RtcGatewayEventReason:
RtcGatewayMessageContent: The weather is fine.
```

IM_EVENT_TYPE
PluginImHandler **method:** onInstantMessage **event:** SessionEventStatus

```
IBM Sametime Gateway RtcGatewayLoggerEvent Creation Time: 2006-10-
08T13:55:26.125+08:00
RtcGatewayEventDirection: 1
RtcGatewayEventType: 4
RtcGatewaySessionID: 9.181.65.251_1160286237500_32
RtcGatewayRequestID: 9.181.65.251_1160286237500_36
RtcGatewayEventStatus: 0
RtcGatewayEventReason:
```

END_SESSION_TYPE
PluginSessionHandler **method:** endSession **event:** SessionEvent

```
IBM Sametime Gateway RtcGatewayLoggerEvent Creation Time: 2006-10-
08T13:57:15.312+08:00
RtcGatewayEventDirection: 0
RtcGatewayEventType: 7
RtcGatewaySenderURI: user409@lotus.com
RtcGatewayReceiverURI: user709@ibm.com
RtcGatewaySenderCommunity: lwpst2
RtcGatewayReceiverCommunity: lwpps2
RtcGatewaySessionID: 9.181.65.251_1160286237500_32
```

```
RtcGatewayRequestID: 9.181.65.251_1160286237500_37
RtcGatewayEventStatus: 0
RtcGatewayEventReason:
RtcGatewaySessionDescription: end of session
RtcGatewaySessionTerminationReason: session ended by other user
END_SESSION_TYPE
PluginSessionHandler    method: onEndSession     event: SessionEventStatus

IBM Sametime Gateway RtcGatewayLoggerEvent Creation Time: 2006-10-
08T13:57:16.312+08:00
RtcGatewayEventDirection: 1
RtcGatewayEventType: 7
RtcGatewaySenderURI: user409@lotus.com
RtcGatewaySessionID: 9.181.65.251_1160286237500_32
RtcGatewayRequestID: 9.181.65.251_1160286237500_37
RtcGatewayEventStatus: 0
```

Presence: PluginPresenceHandler

The events published for Presence have a specific order.

1. Presence always begins with a subscribe event specifying the sender and receiver URI and community.
2. This is followed by an onSubscribe.
3. A number of notify and onNotify events will then be logged as presence state changes.
4. The sender and receiver URI and community will flip as user status updates are sent from the receiver to the sender.
5. An unsubscribe event is logged
6. onUnsubscribe is logged when the user is removed from the buddy list.

As noted above for IM Chat Session, "on" event might not be logged.

The following code snippet describes **user709@ibm.com** adding **user409@lotus.com** to a buddylist **user409@lotus.com sends** presence information. **user409@lotus.**com is then removed from the buddylist.

```
SUBSCRIBE_EVENT_TYPE
PluginPresenceHandler   method: subscribe       event: PresenceEvent

IBM Sametime Gateway RtcGatewayLoggerEvent Creation Time: 2006-10-
08T13:55:19.453+08:00
RtcGatewayEventDirection: 0
RtcGatewayEventType: 1
RtcGatewaySenderURI: user709@ibm.com
RtcGatewayReceiverURI: user409@lotus.com
RtcGatewaySenderCommunity: lwpps2
RtcGatewayReceiverCommunity: lwpst2
RtcGatewaySessionID: 9.181.65.251_1160286237500_43
RtcGatewayRequestID: 9.181.65.251_1160286237500_45
RtcGatewayEventStatus: 0
RtcGatewayEventReason:
SUBSCRIBE_EVENT_TYPE
PluginPresenceHandler   method: onSubscribe     event: EventStatus
```

```
IBM Sametime Gateway RtcGatewayLoggerEvent Creation Time: 2006-10-
08T13:55:20.328+08:00
RtcGatewayEventDirection: 1
RtcGatewayEventType: 1
RtcGatewaySessionID: 9.181.65.251_1160286237500_43
RtcGatewayRequestID: 9.181.65.251_1160286237500_45
RtcGatewayEventStatus: 0
RtcGatewayEventReason: OK
```

NOTIFY_EVENT_TYPE
PluginPresenceHandler **method:** notify **event:** PresenceEvent

```
IBM Sametime Gateway RtcGatewayLoggerEvent Creation Time: 2006-10-
08T13:55:20.406+08:00
RtcGatewayEventDirection: 0
RtcGatewayEventType: 3
RtcGatewaySenderURI: user409@lotus.com
RtcGatewayReceiverURI: user709@ibm.com
RtcGatewaySenderCommunity: lwpst2
RtcGatewayReceiverCommunity: lwpps2
RtcGatewayUserStatus: <?xml version="1.0" encoding="UTF-8"?><presence
xmlns="urn:ietf:params:xml:ns:pidf"
xmlns:dm="urn:ietf:params:xml:ns:pidf:data-model"
xmlns:rpid="urn:ietf:params:xml:ns:pidf:rpid"
xmlns:lt="urn:ietf:params:xml:ns:location-type"
xmlns:ci="urn:ietf:params:xml:ns:pidf:cipid" entity="user709@ibm.com"><tuple
id=""><status><basic>open</basic></status></tuple><dm:person
id="p1"><rpid:activities><rpid:note><rpid:note>I am
Active</rpid:note></rpid:note></rpid:activities></dm:person></presence>
RtcGatewaySessionID: 9.181.65.251_1160286237500_43
RtcGatewayRequestID: 9.181.65.251_1160286237500_49
RtcGatewayEventStatus: 0
RtcGatewayEventReason:
```

NOTIFY_EVENT_TYPE
PluginPresenceHandler **method:** onNotify **event:** EventStatus

```
IBM Sametime Gateway RtcGatewayLoggerEvent Creation Time: 2006-10-
08T13:55:20.64+08:00
RtcGatewayEventDirection: 1
RtcGatewayEventType: 3
RtcGatewaySessionID: 9.181.65.251_1160286237500_43
RtcGatewayRequestID: 9.181.65.251_1160286237500_49
RtcGatewayEventStatus: 0
RtcGatewayEventReason:
```

NOTIFY_EVENT_TYPE
PluginPresenceHandler **method:** notify **event:** PresenceEvent

```
IBM Sametime Gateway RtcGatewayLoggerEvent Creation Time: 2006-10-
08T14:03:14.375+08:00
RtcGatewayEventDirection: 0
RtcGatewayEventType: 3
RtcGatewaySenderURI: user409@lotus.com
RtcGatewayReceiverURI: user709@ibm.com
RtcGatewaySenderCommunity: lwpst2
RtcGatewayReceiverCommunity: lwpps2
```

```
RtcGatewayUserStatus: <?xml version="1.0" encoding="UTF-8"?><presence
xmlns="urn:ietf:params:xml:ns:pidf"
xmlns:dm="urn:ietf:params:xml:ns:pidf:data-model"
xmlns:rpid="urn:ietf:params:xml:ns:pidf:rpid"
xmlns:lt="urn:ietf:params:xml:ns:location-type"
xmlns:ci="urn:ietf:params:xml:ns:pidf:cipid" entity="user709@ibm.com"><tuple
id=""><status><basic>open</basic></status></tuple><dm:person
id="p1"><rpid:activities><rpid:away/><rpid:note><rpid:note>I am away from my
computer
now</rpid:note></rpid:note></rpid:activities></dm:person></presence>
RtcGatewaySessionID: 9.181.65.251_1160286237500_43
RtcGatewayRequestID: 9.181.65.251_1160286237500_56
RtcGatewayEventStatus: 0
RtcGatewayEventReason:
```

NOTIFY_EVENT_TYPE
PluginPresenceHandler **method:** onNotify **event:** EventStatus

```
IBM Sametime Gateway RtcGatewayLoggerEvent Creation Time: 2006-10-
08T14:03:14.562+08:00
RtcGatewayEventDirection: 1
RtcGatewayEventType: 3
RtcGatewaySessionID: 9.181.65.251_1160286237500_43
RtcGatewayRequestID: 9.181.65.251_1160286237500_56
RtcGatewayEventStatus: 0
RtcGatewayEventReason:
```

UNSUBSCRIBE_EVENT_TYPE
PluginPresenceHandler **method:** unsubcribe **event:** PresenceEvent

```
IBM Sametime Gateway RtcGatewayLoggerEvent Creation Time: 2006-10-
08T13:52:34.625+08:00
RtcGatewayEventDirection: 0
RtcGatewayEventType: 2
RtcGatewaySenderURI: user709@ibm.com
RtcGatewayReceiverURI: user409@lotus.com
RtcGatewaySenderCommunity: lwpps2
RtcGatewayReceiverCommunity: lwpst2
RtcGatewaySessionID: 9.181.65.251_1160286237500_43
RtcGatewayRequestID: 9.181.65.251_1160286237500_57
RtcGatewayEventStatus: 0
RtcGatewayEventReason:
```

UNSUBSCRIBE_EVENT_TYPE
PluginPresenceHandler **method:** onUnsubcribe **event:** EventStatus

```
IBM Sametime Gateway RtcGatewayLoggerEvent Creation Time: 2006-10-
08T13:52:35.359+08:00
RtcGatewayEventDirection: 1
RtcGatewayEventType: 2
RtcGatewaySessionID: 9.181.65.251_1160286237500_43
RtcGatewayRequestID: 9.181.65.251_1160286237500_57
RtcGatewayEventStatus: 0
```

Chapter 5. Extending the Sametime Gateway

The Hello World plug-in, that is included with the Sametime Gateway SDK, demonstrates creating, packaging and deploying a simple plug-in. It can also be applied to other purposes, like debugging, in order to differentiate between gateway problems and user plug-in problems.

The Hello World plug-in:

- Implements the PluginRegistration
- Prints a greeting message when init() is received
- Prints a parting message when terminate() is received.
- Updates messages when a customPropertiesChanged() is received.

Development checklist

To create a Sametime Gateway plug-in, you will need to complete each of the tasks listed in the table below and described in the following sections.

Table listing the tasks required to create a Sametime Gateway plug-in.

Step	Description
1	Create the project.
2	Add the jar files.
3	Create the plug-in's package and class.
4	Write the plugin.xml file.
5	Package and deploy the plug-in.
6	Enable the plug-in.
7	Test the plug-in.

Create the project

A project needs to be created to store the plug-in. The Eclipse plug-in extension model is used, however, it is not deployed in the same way as an Eclipse Rich Client plug-in. Full instructions on how this has been implemented in WebSphere Application Server 7.0 can be found at the following URL:

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/com.ibm.websphere.nd.doc/info/ae/ae/cweb_extensions.html

Follow these steps to create a new project.

Table listing the steps needed to complete a new progject.

Step	Action
1	Launch **Rational Application Developer**.
2	Create a new Dynamic Web Project project; **File >New >Dynamic Web Project**
3	Type **gw.samples.helloworld** in the Name field.
4	Uncheck **Add module to an EAR project** in the advanced section.

5	Click **Finish**.

Note The samples distributed with the SDK are packaged as an EAR file. In this chapter, a WAR file is used to package the sample. Both techniques are valid.

Add the jar files

An extension to the Sametime Gateway depends on several classes that are included with the installation files. The next step in setting up the project is to make those classes available.

At runtime these classes will be available on the server as part of the running context. For the development and build stages, however, you need to depend on them externally. You will need to copy the rtc.gatewayAPI.jar file from the STGateway/st_gw_lib directory on the installation server, to a location on your development machine. Once that is completed, you need to add them to your project's dependencies.

Follow these steps to add the necessary JAR files.

Table listing the steps to add the JAR files.

Step	Action
1	Open up the properties dialog box for the project you just created, **gw.samples.helloworld**.
2	Navigate to the **Java Build Path** pane and select the **Libraries** tab.
3	Click **Add External Jars.**
4	Navigate to the RTCGateway /lib directory, where you copied the jar files, and select **rtc.gatewayAPI.jar**. Click **OK**.
5	Confirm all dialogs and return to the development environment.

Create the plug-in's package and class

The Hello World plug-in is initialized when the Sametime Gateway core is started, and prints, "Hello world". When the core is shut-down, the plug-in is terminated and prints "Goodbye world". These messages are set from the custom properties for the plug-in upon initialization, and when they are changed in the admin interface. For more information on overriding the init(), terminate() and customPropertiesChanged() methods see 7.1, The PluginRegistration.

Once the default messages are set, you will need to fill in the three functions that are declared in PluginRegistration, init(), customPropertiesChanged(), and terminate(). Since all plug-ins must implement PluginRegistration, you will see this pattern throughout all of your plug-ins.

You need to fill in the code for init(). There are three parts to this:

- Extract the needed values from the initialization parameters.
- Perform the initialization.
- Signal back to the core that you are done.

In a plug-in with a longer startup time, the init() method would typically start a separate thread to perform the initialization. This would be passed the **PluginsStartupCallback** object and when it was finished it would notify the core through this object.

Note Even if your plug-in has no special initialization to do (or performs it in its constructor rather than upon init()), you still need to fill in this function and call the callback routine with the id of your plugin as defined in the plugin.xml file. Otherwise the core will assume your plug-in is not ready yet and you will receive no further notification messages.

Once the init() parameters have been set you need to fill in the customPropertiesChanged() method. CustomPropertiesChanged() receives a single argument, the same as the first argument for init(). This contains the current state of the custom properties. If you just want to know what is changed, you need to persist them yourself and calculate the delta. In this case we just run through them exactly the same as we did in init() and extract their values.

Finally, you need to fill in the terminate() method by adding the goodbye message.

Follow these steps to create a package and class for the plug-in.

Table listing the steps to create a package and class for the plug-in.

Step	Action
1	Select **File** > **New** > **Package**.
2	Name the package **com.ibm.collaboration.realtime.sample.plugin.helloworld**
3	Right click the package and select **New** > **Class** to create a class within the helloworld package to house the actual plug-in code.
4	Name the class **HelloWorldPlugin**.
5	Specify that it is to implement the interface **com.ibm.rtc.gateway.plugin.PluginRegistration**. **Result**: This will create a class shell defining the init(), terminate() and customPropertiesChanged () methods.
6	Declare a static name for this plug-in. This value must be the concatenation of the id of the plugin and the id of the extension for the mhplugin extension point as defined in the plugin.xml file. Your code should look like the following: private final static String PLUGIN_NAME = "com.ibm.collaboration.realtime.sample.samples.helloworld"; Note: This isn't required, but it is good practice.

7	Next, create member variables which will house the greeting and parting strings. Set the default values to "Hello world" and "Goodbye world". Your code should like the following lines:
	`  private String    _hello = "Hello world";`
	`  private String    _goodbye = "Goodbye world";`

8	You need to extract the needed values from the initialization parameters . One of the parameters passed to init() is a **Collection** of **CustomProperty** objects. We need to iterate through these and look for properties named "helloworld.msg.hello", and "helloworld.msg.goodbye". If these are found, we extract their values and assign it to the class variables. If we don't' find it, the default values are retained.
	For example:
	`if (prop.getName().equals("helloworld.msg.hello"))`
	`     _hello = prop.getValue();`
	`   else if (prop.getName().equals("helloworld.msg.goodbye"))`
	`     _goodbye = prop.getValue();`
	
	Note The property namespace is unique to each plug-in and we don't' have to prefix it with the plug-in name, but it's good practice for clarity.

9	The only action involved with the initialization is to print out the greeting message:

	`System.out.println(_hello);`
10	Lastly we have to signal back to the core that we have finished our initialization. The second argument passed to init() is a callback function. We call the pluginStart() method on this with our plug-in’s name as its only argument.

11	Your code should look something like this:
	`public void init(Collection properties, PluginsStartupCallBack callback) {`
	`  // Interrogate initialization parameters`
	`  for (Iterator i = properties.iterator(); i.hasNext(); ) {`
	`    CustomProperty prop = (CustomProperty)i.next();`
	`    if (prop.getName().equals("helloworld.msg.hello"))`
	`      _hello = prop.getValue();`
	`    else if (prop.getName().equals("helloworld.msg.goodbye"))`
	`      _goodbye = prop.getValue();`
	`  }`
	`  System.out.println(_hello);`
	`  callback.pluginStarted(PLUGIN_NAME);`
	`}`

12	Fill in the customPropertiesChanged() method. Your code should look something like this:
	`public void customPropertiesChanged(Collection properties) {`
	`  // Interrogate initialization parameters`
	`  for (Iterator i = properties.iterator(); i.hasNext(); ) {`
	`    CustomProperty prop = (CustomProperty)i.next();`
	`    if (prop.getName().equals("helloworld.msg.hello"))`
	`      _hello = prop.getValue();`
	`    else if (prop.getName().equals("helloworld.msg.goodbye"))`
	`      _goodbye = prop.getValue();`
	`  }`
	`}`

13	Fill in the terminate() function. The code should look something like:
	`public void terminate() {`
	`  System.out.println(_goodbye);`
	`}`

14	Save and exit.

Write the plugin.xml file

In order to be identified as a plug-in used by the Sametime Gateway, the plug-in code is referenced as an extension point in the plugin.xml file. The plugin.xml file identifies the plug-in to the environment, and what extension points it extends.

The following table lists some of the parts of the plugin.xml file and their function.

Table listing some parts of the plugin.xml file, along with their functions.

Part	Function
<plugin>	Identifies the plug-in
id	the name of the primary package
name	A text name
version	Version number for this plug-in.
provider-name	Identifies who created the plug-in
<extension point>	Declares the point that is being extended.

id	The extension needs a unique ID. For example, helloworld. This is passed back in the pluginStarted() callback method in the init() function in your plugin.
name	A name for the extension point you are creating.
<action class>	Within the extension tag we then specify a <action> tag with a single "class" attribute. This points to the class we defined above and tells the environment that this should be instantiated and used to process events on the gateway.

You can provide more than one plug-in in a single project. In this case you just have duplicate <extension> entries for each plug-in.

Follow these steps to identify this class as a Sametime Gateway addition.

Table listing the steps to identify a class as a Sametime Gateway addition.

Step	Action
1	Create the **Web Content/WEB-INF/plugin.xml** file
2	Create the <plugin> tag and set the following parameters: • Id: **com.ibm.collaboration.realtime.sample.samples** • Name: **RTC Demonstration Plug-in** • Version: **1.0.0** • Provider: **your company's name**.
3	Create the <extension> tag and set the following parameters: • Point: **com.ibm.collaboration.realtime.gateway.mhplugin** • ID: **helloworld** • Name: **Hello World Plugin**
4	Create the <action> tag and set its class to **com.ibm.collaboration.realtime.sample.plugin.**`helloworld`**.HelloWorldPlugin.**

5	The plugin.xml file should look like the following:
	`<plugin`
	`   id="com.ibm.collaboration.realtime.sample.samples"`
	`   name="Sametime Gateway Sample Plug-ins"`
	`   version="1.0.0"`
	`   provider-name="IBM">`
	`   <extension`
	`      point="com.ibm.collaboration.realtime.gateway.mhplugin"`
	`      id="helloworld"`
	`      name="Hello World Plugin">`
	`      <action`
	`class="com.ibm.collaboration.realtime.sample.plugin.helloworld.HelloWorldP`
	`lugin">`
	`      </action>`
	`   </extension>`
	`</plugin>`

Package and deploy the plug-in

Follow these steps to package and deploy the Hello World plug-in.

Table listing the steps to package and deploy the "Hellow World" plug-in.

Step	Action
1	Select **File > Export** from the RAD main menu
2	Select the **WAR File** option and click **Next**.

3	Select **gw.samples.helloworld** for your Web Project and pick an appropriate directory on disk for the Destination.
4	Click **Finish**.
5	Launch your Web Browser and navigate to your WebSphere Application Server Integrated Solution Console. **Note**: The following steps also appear in the Sametime Gateway Information Center, Adding or removing a message handler.
6	In the left navigation bar click **Applications**.
7	Choose **Install New Application**.
8	Select **Local File System** and point it at the war file you created, and provide a context root. **Note**: The context root is not used but must be provided for this sample.
9	Continue through the next few screens. **Save** your new configuration.
10	Click **Applications > Enterprise Applications.**
11	Select **gw.samples.helloworld,** the J2EE application which contains the message handler**.**
12	Click **Start** to start the application. **Result**: This will register the plug-in with the Sametime Gateway.

Note These instructions are for a simple deployment. Consult the WebSphere Application Server documentation for all of the options and possibilities for a more complicated scenarios.

Enable the plug-in

The WebSphere Application Server Integrated Solutions Console Real-Time Collaboration Panel is used to configure the properties of the message handler. The configuration options include, setting type, defining order and determining whether a plug-in is always called. After the message handler is configured, you must enable it.

Follow these steps to enable the Hello World plug-in.

Table listing the steps to enable the "Hello World" plug-in.

Step	Action
1	Open the **WebSphere Application Server Integrated Solutions Console**.
2	Click **Sametime Gateway > Message Handlers** in the left hand navigator. **Result**: You should see the Message Handler list page.
3	Verify that the Hello World plug-in is displayed, as **Disabled**. **Note**: If it is not there, you will need to troubleshoot your message handler plug-in.
4	**Click Hello World to edit its properties, such as type:**
5	Select **Run the message handler regardless of whether previous handlers complete their processing of messages**. **Note**: This step is optional. If this is selected the message handler runs even if the preceding message handler failed to complete its handling of a message.
6	Click **OK**. **Result**: You are returned to the Message Handler list page.
7	On the Message Handler list page, select **Hello World**. Click **Enable.** **Optional**. Use the Move buttons to change the order in which the message handler processes

	messages. **Note** The User Locator message handler **must** be first and the Event logger message handle should be last.

Refer to the *Sametime Gateway Installation and Configuration Guide* for more information.

Test the plug-in

Test the Hello World plug-in by opening Open **SystemOut.log**, from the WebSphere Application Server log directory, for example:

```
WAS70\AppServer\profiles\RTCGW_Profile\logs\RTCGWServer
```

You will see the greeting message, which indicates that your plug-in was instantiated and called correctly.

Congratulations! You have written your first plug-in!

Chapter 6. Sample Plug-ins

This chapter presents the sample plug-ins that are included in the Sametime Gateway SDK.

The following table lists the plug-in samples as well as what they do and the extensions they use.

Table listing plug-in samples with descriptions of each.

Plug-in or sample	Purpose	Extensions
Hello World	This simple example demonstrates how to create, package and deploy a plug-in.	*PluginRegistration
Compliance logger	Demonstrates chat transcript logging based on criteria.	*PluginRegistration *PluginImHandler *PluginSessionHandler
Presence Privacy	Demonstrates how to implement more sophisticated privacy from presence awareness.	*PluginRegistration *PluginPresenceHandler

- To save space the asterisk (*) denotes `com.ibm.collaboration.realtime.gateway.plugin.`

Hello World plug- in

As you saw in Chapter 5, *Extending the Sametime Gateway*, the Hello World plug-in extends the PluginRegistration interface to generate messages when the gateway is started and shut down.

The compliance logger plug- in

Corporations are interested in tracking their dealings, both internally and externally. Many IM applications typically have administrative tools to handle tracking and message logging, and even the ability to monitor key words or patterns. However, once messages go in or out of the internal server, the ability to track and trace these messages is lost.

The Sametime Gateway provides a way to administer internal to external communications and vice versa.

The compliance logger plug-in implements the PluginRegistration (through inheritance), PluginImHandler and PluginSessionHandler. When a session starts is takes details and creates an object to track session messages. As messages come in, they are appended to the tracking object. If the message contains a word from a watch list, the tracking object has a flag set. When a session ends, if the tracking object has the watch flag set, the contents of the session are saved.

For the purpose of this sample, the business logic for making the decision of what to log and how to log it is encapsulated in another class. The plug-in just demonstrates how to extract the needed information from the gateway notification messages and pass it down into your business logic.

Create the project

Create a package and a ChatLogPlugin class within this package. Choose to implement the PluginImHandler and PluginSessionHandler interfaces.

Follow these steps to create a new project.

Table listing the steps to create a new project.

Step	Action
1	Launch **Rational Application Developer**.
2	Create a new Dynamic Web Project project; **File >New >Dynamic Web Project**
3	Type **gw.samples.chatlog.plugin** in the Name field.
4	Uncheck **Add module to an EAR project** in the advanced section.
5	Click **Finish**.

Create the plug-in's package and class

Follow these steps to create a package and class for the plug-in.

Table listing the steps to create a package and class for the plug-in.

Step	Action
1	Click **File > New > Package**.
2	Name the package **com.ibm.collaboration.realtime.sample.plugin.chatlog**
3	Right Click the package and choose **New > Class** to create a class within the chatlog package to house the actual plug-in code.
4	Name the class **ChatLogPlugin**.
5	Select the **PluginImHandler** and **PluginSessionHandler** interfaces to be implemented. **Result**: This will create a class shell defining the methods necessary for those interfaces.

Code structure

Service classes determine if logging is to occur and handle the logging itself. This keeps the code fairly simple. For the Compliance logger, this practice highlights interfacing with the gateway without getting into customer specific methods for the business logic; it is also a good development practice.
You could even use the same Eclipse extension point and plug-in system to make the code more modular. You would define an interface in this plug-in for such a service and an extension point that other plug-ins could create a service for. This plug-in would then search for other plug-ins that defined this interface, instantiate them and direct calls to them.

Although this is more than this example needs, the ides is applicable in cases where there are a range of different logging or compliance products. Or this approach might be useful for sharing code between the mechanism for interacting with messages within a translation protocol and interacting with message that are between translation protocols.

Plugin Code

You will first need to address the overall lifecycle management of the plug-in. In the init() method add code to instantiate the logging service. This is called by the gateway when the plug-in is enabled. You will need to add code to the terminate() method to terminate all existing logging sessions.

Normally sessions are cleanly started and ended when the gateway is shut down, however, it is possible that no session termination will be indicated. So in your code you need to ensure that you have logged any sessions that are meant to be terminated.

You will also need to complete the customPropertiesChanged() method. Since it is not used, you need to mark it as non-operations.

Your code will look like the following:

```
public class ChatLogPlugin  implements PluginSessionHandler, PluginImHandler
{
 private final static String PLUGIN_NAME = "ChatLogPlugin";
 private ChatLogService   _logger;

 public void init(Collection properties, PluginsStartupCallBack callback) {
  System.out.println("ChatLogPlugin: Starting Chat Log Service");
  _logger = new ChatLogService();
  callback.pluginStarted(PLUGIN_NAME);
 }

 public void terminate() {
  System.out.println("ChatLogPlugin: Terminating Chat Log Service");
  _logger.endAllSessions();
 }

 public void customPropertiesChanged(Collection properties) {
  // NOP
 }
 …
}
```

Tracking session lifecycles

The PluginSessionHandler contains the following methods:

- startSession()
- onStartSession()
- endSession()
- onEndSession()

As described in the Reference section, "PluginSessionHandler," the basic operations, startSession() and endSession(), are called when a request is made from the originating community to the destination community. The on operations, onStartSession() and onEndSession(), are called when there is a successful return from the operation against the community. If another plug-in has blocked the operation the on operation methods are not called. If there has been a failure in the operation against the destination community, the on methods are called, but with a status indicating the reason for the failure.

For the compliance logger plug-in plug-in, the code that starts the tracking needs to be placed in the onStartSession() method. This will start tracking a session if it is successfully initiated. If the session fails to start, there is no reason to track it. You will want to extract the information that monitors the tracking failure or success in the StartSession() method and persist it in the class indexed by the requested value. The onStartSession retrieves this information and passes it to the tracker.

A HashMap or other in-memory persistence object is suitable for a sample like the compliance logger plug-in. However, if you have a clustered environment you have to remember that the plug-in interface methods, like startSession() and onStartSession(), are asynchronous and might be delivered to different machines in the cluster. A better implementation would be to persist such information in a globally accessible store.

The endSession() method will contain the terminate tracking code. Even if some other plug-in terminates the action, the conversation is recorded to the log. Because the information is directly available, the code extracts the pertinent information and passes it to the tracker.

Methods that are not used should be marked as non-operational.

The code looks something like this:

```
private HashMap   _startSessionParameters = new HashMap();

public void startSession(SessionEvent ssEvent) {
 System.out.println("ChatLogPlugin: startSession");
 // retrieve parameters we care about here
 String requestID = ssEvent.getRequestID();
 String description = ssEvent.getDescription();
 String participant1 = ssEvent. getOriginatorUser ();
 String participant2 = ssEvent. getDestinationUser ();
 // save the parameters
 _startSessionParameters.put(requestID+".description", description);
 _startSessionParameters.put(requestID+".participant1", participant1);
 _startSessionParameters.put(requestID+".participant2", participant2);
}

public void onStartSession(SessionEventStatus status) {
 String sessionID = status.getSessionID();
 String requestID = status.getRequestID();
 // retrieve persisted parameters
 String description = (String)_startSessionParameters.get(requestID+".description");
 String participant1 = (String)_startSessionParameters.get(requestID+".participant1");
 String participant2 = (String)_startSessionParameters.get(requestID+".participant2");

// call the business logic
 System.out.println("ChatLogPlugin: Starting Session with "+participant1+" and "+participant2
);
 _logger.startSession(sessionID, description, participant1, participant2);

// clean up persisted parameters
 _startSessionParameters.remove(requestID+".description");
 _startSessionParameters.remove(requestID+".participant1");
 _startSessionParameters.remove(requestID+".participant2");
}

public void endSession(SessionEvent esEvent) {
 // extract the parameters we care about
 String sessionID = esEvent.getSessionID();
 String terminationReason = esEvent.getTerminateReason();

// call the business logic
 System.out.println("ChatLogPlugin: Ending Session with "+esEvent. getOriginatorUser ()+" and
"+esEvent. getDestinationUser ());
 _logger.endSession(sessionID, terminationReason);
}

public void onEndSession(SessionEventStatus arg0) {
 // NOP
 System.out.println("ChatLogPlugin: onEndSession");
}
```

You will also need to track messages. The PluginImHandler interface uses the instantMessage() and onInstantMessage() APIs. Similar to tracking a session if it is successfully started, you need to add functionality to the onInstantMessage() method in order to track successfully delivered messages.

If the message is terminated by another plug-in or connecter failure it does not need to be tracked. Finally, the code needs to:

- Extract information from the instantMessage() event,
- Persist it if indexed by the requestId,
- Retrieve it via onInstantMessage()
- Pass it into the business logic.

For details on these values see the Reference section, *"PluginImHandler."*

Your code should look like the following:

```
private HashMap   _instantMessageParameters = new HashMap();

public void instantMessage(ImEvent imEvent) {
 System.out.println("ChatLogPlugin: instantMessage");
 String sessionID = imEvent.getSessionID();
 String requestID = imEvent.getRequestID();
 // retrieve parameters we care about here
 String from = imEvent. getOriginatorUser ();
 String to = imEvent. getDestinationUser ();
 String message = imEvent.getMessage();
 // save the parameters
 _instantMessageParameters.put(requestID+".from", from);
 _instantMessageParameters.put(requestID+".to", to);
 _instantMessageParameters.put(requestID+".message", message);
}

public void onInstantMessage(SessionEventStatus status) {
 String sessionID = status.getSessionID();
 String requestID = status.getRequestID();
 // retrieve persisted parameters
 String from = (String)_instantMessageParameters.get(requestID+".from");
 String to = (String)_instantMessageParameters.get(requestID+".to");
 String message = (String)_instantMessageParameters.get(requestID+".message");
 System.out.println("ChatLogPlugin: Message from "+from+" to "+to);

 // call the business logic
 _logger.trackMessage(sessionID, from, to, message);

 // clean up persisted parameters
 _instantMessageParameters.remove(requestID+".from");
 _instantMessageParameters.remove(requestID+".to");
 _instantMessageParameters.remove(requestID+".message");
}
```

Tracker Code

The SessionTracker object is a central object that is used to group semantic session information across various operation calls. Participants and their conversations are noted and tracked by the ChatLogService. This in turn, is indexed by the SessionID. The TrackingCriteriaService is consulted to determine if the conversation should be monitored, based on criteria such as key words or participant URLs. If something from the conversation is flagged for logging, a call is made to the PersistService to store the information, otherwise the object is freed at the end of the session.

In this simple implementation, all information is cached in memory for the duration of the session. In a production system you might cache this in a repository somewhere, or do partial logging once a certain size is reached. Or the logic might be reversed: all conversations are logged until their conclusion. If, at that time, it is decided not to keep the log, the information can then be deleted. The best choice will depend on the memory and load constraints on the server, and the persistence method.

The Presence Privacy plug- in

The Presence Privacy plug-in example implement a more sophisticated presence awareness privacy model. using the PluginPresenceHandler.

Business logic is applied and actions are taken based on subscription requests and notifications. When a presence subscription is requested, business logic is applied to see if this subscription should be allowed or not.

Upon subscription and notify business logic is in order to determine which action to take. we check our business logic to see if such a subscription is allowed. If not, we change the status of that request to failure and it stops there. If allowed, we also check later, when a notification of a request change goes out, and if not allowed there, we change the status message to “unknown”.

For the purpose of the demonstration, the business logic for making the decision of what to log and how to log it is encapsulated in another class. The plug-in just demonstrates how to extract the needed information from the gateway notification messages and pass it down into your business logic.

Create the project

Create a package and a PresenceBlockerPlugin class within this package. Choose to implement the PluginPresenceHandler interface.

Follow these steps to create a new project.

Table listing the steps to create a new project.

Step	Action
1	Launch **Rational Application Developer**.
2	Create a new Dynamic Web Project project; **File >New >Dynamic Web Project**
3	Type **gw.samples. presblock.plugin** in the Name field.
4	Uncheck **Add module to an EAR project** in the advanced section.
5	Click **Finish**.

Create the plug-in’s package and class

Follow these steps to create a package and class for the plug-in.

Table listing the steps to create a package and class for the plug-in.

Step	Action
1	Click **File > New > Package**.
2	Name the package **com.ibm.collaboration.realtime.sample.plugin.presblock**
3	Right Click the package and choose **New > Class** to create a class within the chatlog package to house the actual plugin code.
4	Name the class **PresenceBlockerPlugin**.
5	Select the **PluginPresenceHandler** interface to be implemented. **Result**: This will create a class shell defining the methods necessary for those interfaces.

Code structure

A service class determines if blocking is to occur. This keeps the code fairly simple. For the presence blocker, this practice highlights interfacing with the gateway without getting into customer specific methods for the business logic; it is also a good development practice.

Plugin Code

Similar to Hello World and the Compliance logger plug-ins, you need to manage the plug-ins overall lifecycle. Add code to the init() method that instantiates the blocking service when the plug-in is enabled. Because customPropertiesChanged() and terminate() aren’t used, they need to be marked as non-operations.

The code will look like the following:

```
public class PresenceBlockerPlugin  implements PluginPresenceHandler
{
 private final static String PLUGIN_NAME = "PresenceBlockerPlugin";

 private BlockingCriteriaService   _blockingCriteria;

 public void init(Collection properties, PluginsStartupCallBack callback) {
  System.out.println("PresenceBlockerPlugin: Starting Presence Blocking Service");
  _blockingCriteria = new BlockingCriteriaService();
  callback.pluginStarted(PLUGIN_NAME);
 }

 public void terminate() {
  // NOP
 }

 public void customPropertiesChanged(Collection properties) {
  // NOP
 }
 …
}
```

Tracking presence

PresenceInterface contains 4 methods:

- subscribe()
- onSubscribe()
- unsubscribe()
- onUnsubscribe()

Similar to the PluginPresenceHandler when a request is made from the originating community to the destination community subscribe() and unsubscribe() are called. The on operations, onSubscribe() and onUnsubscribe() are called when the operation returns successfully; if another plug-in has blocked the operation the on operation methods are not called. If there has been a failure in the operation on the destination community, the on methods are called, but with a status indicating the reason for the failure.

In order to block a presence subscription from starting you will need to put the code in the subscribe() method. Values are extracted and passed to the business logic. If the message is blocked, the event status is changed to an UNAUTHORIZED ACTION failure.

For details on these values see the Reference section "PluginPresenceHandler."

The remaining methods are not used so they are marked non-operational.

The code looks something like this:

```
public void subscribe(PresenceEvent subEvent) {
 System.out.println("PresenceBlockerPlugin: subscribe");
 // we etract from the event the parameters we need to use
 EventStatus status = subEvent.getEventStatus();
 String from = subEvent. getOriginatorUser ();
 String to = subEvent. getDestinationUser ();
 if (status.getStatus() != GwErrorCodes.SUCCESS)
  return;
 // We call the business logic
 if (_blockingCriteria.isBlocker(from, to))
 {
  // If the business logic requires it, we change the status
  System.out.println("PresenceBlockerPlugin: subscribe from "+from+" to "+to+" blocked.");
  EventStatus newStatus =
     EventStatusFactory.getInstance().createEventStatus(
          status.getSessionID(), status.getRequestID(),
          GatewayReturnCodes.UNAUTHORIZED_ACTION,
          status.getReason());
  subEvent.setEventStatus(newStatus);
 }
 else
  System.out.println("PresenceBlockerPlugin: subscribe from "+from+" to "+to+" allowed.");
}

public void unsubscribe(PresenceEvent unsubEvent) {
 // NOP
 System.out.println("PresenceBlockerPlugin: unsubscribe");
}

public void onSubscribe(EventStatus status) {
 // NOP
 System.out.println("PresenceBlockerPlugin: onSubscribe");
}

public void onUnsubscribe(EventStatus arg0) {
 // NOP
 System.out.println("PresenceBlockerPlugin: onUnsubscribe");
}
```

Finally you will need to track the notifications, too. The PluginPresenceHandler interface also contains the following four APIs:

- notify()
- fetch()
- onNotify()
- onFetch ()

Similar to the other plug-in samples, functionality for this plug-in should be added to the notify() method. Values need to be extracted, and if necessary, passed to the business logic. If you want to mask the status replace the current status with "unavailable".

The code looks something like this:

```
public void notify(PresenceEvent notifyEvent) {
 // we etract from the event the parameters we need to use
 String from = notifyEvent.getOrigUser();
 String to = notifyEvent.getDestUser();
 // We call the business logic
 if (_blockingCriteria.isBlocker(from, to))
 {
  System.out.println("PresenceBlockerPlugin: notify from "+from+" to "+to+"
blocked.");

  // If the business logic requires it, block the notify
  EventStatus status = notifyEvent.getEventStatus();
  notifyEvent.setEventStatus(
     EventStatusFactory.getInstance().createEventStatus(
        status.getSessionID(), status.getRequestID(),
        GatewayReturnCodes.UNAUTHORIZED_ACTION,
        status.getReason()));
 }
 else
  System.out.println("PresenceBlockerPlugin: notify from "+from+" to "+to+"
allowed.");
}

public void onNotify(EventStatus arg0) {
 // NOP
 System.out.println("PresenceBlockerPlugin: onNotify");
}

public void fetch(PresenceEvent event, boolean isNewSession) {
 // NOP
}

public void onFetch(EventStatus status, boolean isNewSession) {
 // NOP
}
```

Importing the samples into RAD

Follow these steps to import the message handler plug-in samples included in the IBM Lotus Real-Time Collaboration SDK into RAD.

Table listing the import the message handler plug-in samples.

Step	Action
1	Launch **RAD**. Accept the default workbench location if none has been selected.
2	Click **File > Import > EAR file**.
3	Click **Next**.
4	Click Browse and select the rtcgw_samplesEAR.ear from where the Sametime Gateway SDK was installed.
5	Click **Finish.**
6	To view the project, click the **Open Perspective** icon at the top right corner of RAD.
7	Click **J2EE**. Result: There will probably be an error message because a Sametime Gateway jar file needs to be added to the Java build path.
8	**Expand** EJB Projects and then **Right click** the rtcgw_samples name, in the left navigator. Click **Properties**.
9	Click **Java Build Path, then select the Librareis tab**. If rtc.gatewayAPI.jar already appears in the list select it and click **Remove**.
	Click **Add External JARs**.
11	Browse for the **rtc.gatewayAPI.jar** which you can download from the Sametime Gateway to your local machine. This is located in <WAS_RTCGateway>\<rtcgw>\rtc_gw_lib
12	Browse for the com.ibm.events.client.jar which you can download from the Sametime Gateway to your local machine. This is located in <WAS_RTCGateway>\<AppServer>\plugins. **Note**: This is required by event consumer samples.
13	Click **OK**.

Exporting the samples as an EAR file

Once you have modified your plug-ins you need to package them as an EAR file in order to deploy them.

Table listing the steps to package plug-ins as an EAR file.

Step	Action
1	Launch **RAD**. Accept the default workbench location if none has been selected
2	**Click File > Export > EAR file**
3	Select EAR Project: **rtcgw_samplesEAR**
4	Select **Destination**: Click **Browse** next to Destination and select a destination folder. **Note**: This can be any folder you choose.
5	Click **Finish**

In order to install the EAR file you just created you need to refer to the WAS / gateway Integrated console solutions documentation. This information can also be found in the Sametime Gateway Information Center. Chapter 7 contains a section, “Installing and running the samples,” which also addresses installing and running the samples described in this guide.

Chapter 7. Sample Event Consumers

This chapter presents the sample event consumers that are included in the Sametime Gateway SDK. The following table lists the event consumer samples as well as what they do and the event consumer type.

Table listing the event consumer samples with descriptions of each.

Sample	Purpose	Type
Logger	Demonstrates how to parse and log the Sametime Gateway data contained in Common Base Events. This example also includes logic for counting and logging the number of open sessions per community.	Asynchronous event notifications
QueryOpenSessions	Demonstrates how to perform a query using the Event Access Interface and parse the results returned as Common Base Events.	Query events using the Event Access Interface

Event Consumer Logger Sample

In Chapter 5, a sample was discussed using the message handler plug-in interfaces. This sample tracked IM sessions and saved those sessions which contained a specific word. Event consumers do not use these interfaces. Instead, these interfaces are used by the event source, which is the Event logger included in the Sametime Gateway.

The Event logger event source uses the Common Event Infrastructure to publish Common Base Events and must be enabled by setting the Message handler properties. More information is provided in the Sametime Gateway Information Center to configure the Event logger. This sample is an event consumer implemented as a message driven bean.

Many applications use the Common Event Infrastructure to publish Common Base Events. The logger sample used in this guide is only interested in Sametime Gateway events and a NotificationHelper is used to filter out events. A notification helper enables event consumers to interact with the messaging infrastructure by using the messaging programming model rather than a Common Event Infrastructure interface. The notification helper provides an event selector for filtering. The sample specifies only Sametime Gateway events will be delivered to the Message Driven Bean's onMessage method.

Two methods in the Message Drive Bean will be discussed in this section:

- Initialization is performed in the ejbCreate method
- Message processing is performed in the onMessage method.

All initialization and message processing is performed in the MessageLoggerService. The Message Drive Bean is named, MessageLoggerBean and this class does not contain any business logic.

Initialization

Initialization for a message driven bean is performed in the ejbCreate method. ejbCreate will call init on the MessageLoggerService to create a notification helper using the notification helper factory. The event selector is passed an XPath expression that is used to filter event notifications. Any of the standard

Common Base Events fields could be used. For this sample, extensionName will be used to define only Sametime Gateway events

```
public class MessageLoggerService {
   private NotificationHelper _notificationHelper;

   public void init() {
      try{
         InitialContext _initCtx = new InitialContext();

         // create the notification helper used to define
         // the messages processed and retrieved.
         NotificationHelperFactory nhFactory =
            (NotificationHelperFactory)
            _initCtx.lookup("com/ibm/events/NotificationHelperFactory");

         _notificationHelper = nhFactory.getNotificationHelper();

         try {

            // Only receive Sametime Gateway Messages
            _notificationHelper.setEventSelector(
               "CommonBaseEvent[@extensionName='RtcGatewayLoggerEvent']");

         } catch (EventsException e1) {
            e1.printStackTrace();
         }
      } catch(NamingException e){
         e.printStackTrace();
      }
   }
```

Message Processing and logging

Once the event selector is set, the message driven bean will receive messages which match the XPath expression. The bean's onMessage method is called to receive these notifications and is passed a javax.jms.Message object. The code to log the messages is contained within the MessageLoggerService instead of the MessageLoggerBean.

The notification helper converts the javax.jms.Message into a CommonBaseEvent which contains standard fields and the Sametime Gateway ExtendedDataElement. For each event, this sample logs the creation time, source component name and all ExtendedDataElement.

The MessageLoggerService, processMessage performs the logging and also contains a call to `_trackerService.processEvent` which is discussed below. logEvent is synchronized because the application server could create multiple Message Drive Bean instances resulting in intermixed output from multiple messages.

```
public void processMessage(javax.jms.Message msg){
      EventNotification[] notifications = null;
      try{
         notifications = _notificationHelper.getEventNotifications(msg);
         if(notifications != null){
            for(int i = 0; i < notifications.length; i++){
               int notifType = notifications[i].getNotificationType();
               if(notifType ==
                     NotificationHelper.CREATE_EVENT_NOTIFICATION_TYPE){
```

```
                CommonBaseEvent cbe = notifications[i].getEvent();
                if(cbe != null){
                   // Log every event
                   logEvent(cbe);

                   // Track open sessions per community.
                   _trackerService.processEvent(cbe);
                }
             }
          }
       }
    } catch(EventsException e){
       e.printStackTrace();
    }
 }

 private synchronized void logEvent(CommonBaseEvent cbe){

    // Begin each Event with header information: Component Name
    // and Creation Time.
 String logTime = cbe.getCreationTime();
 ComponentIdentification compId = cbe.getSourceComponentId();
 String componentName = compId.getComponent();
 System.out.println(componentName + " " +
       LoggerConstants.CBE_EXTENSION_NAME + " Creation Time: "
       + logTime);

 // Dump each Extended Data Element Name and Value.
 EList eListData = cbe.getExtendedDataElements();
    ListIterator listIter = eListData.listIterator();

    while(listIter.hasNext()){
       ExtendedDataElement exData =
          (ExtendedDataElement)listIter.next();
       EList eListDataValues = exData.getValues();
       ListIterator listIterVals = eListDataValues.listIterator();

       while(listIterVals.hasNext()){
          // assuming one value per ExtendedDataElement
          String value = (String)listIterVals.next();
          System.out.println(exData.getName() + ": " + value);
    }
 }
}
```

Adding logic to process the Events

The MessageLoggerService, processMessage provided basic logging. This section will show how easy it is to add business logic to event consumers. The CommunityTrackerService class contains business logic which tracks the number of open sessions for each external community. This sample has no user interface and only logs information to SystemOut.log, the basic elements needed to process the ExtendedDataElement, although improved user interface and administration could be added.

The first step is to verify that the required events are published by the Gateway. The PluginSessionHandler, startSession and endSession methods provide the required information to increment and decrement the session counter per external community. The important ExtendedDataElement elements are:

- RtcGatewayEventDirection – specifies an action method.

- RtcGatewayEventType – specifies Start and End Session
- RtcGatewaySenderCommunity – specifies a community
- RtcGatewayReceiverCommunity - specifies a community

The onStartSession and onEndSession could also be used. The difference between these methods and limitations was discussed earlier in this guide.

The following table shows the relevant events:

Table listing session events and methods.

Event Type	**Event Direction**	**Interface**	**Method**	**Event**
7	0	PluginSessionHandler	endSession	SessionEvent
6	0	PluginSessionHandler	startSession	SessionEvent
7	1	PluginSessionHandler	onEndSession	SessionEventStatus
6	1	PluginSessionHandler	onStartSession	SessionEventStatus

The following code sample shows how a CommonBaseEvent can be processed to track the number of open IM sessions.

```
public void processEvent(CommonBaseEvent cbe){
   boolean bCont = true;
   boolean bStartSession = false;
   boolean bEndSession = false;
   String senderCommValue = null;
   String receiverCommValue = null;

   if(cbe != null){
      EList eListData = cbe.getExtendedDataElements();
      ListIterator listIter = eListData.listIterator();

      while(listIter.hasNext()&& bCont){
         ExtendedDataElement exData =
            (ExtendedDataElement)listIter.next();

         // Only Process Events required for Start/End
         // session counter
         if (exData.getName().equals("RtcGatewayEventDirection")) {
            bCont = (getValueInt(exData)==
               GWEventLogConstants.EVENT_DIRECTION_ACTION_METHOD ?
                  true : false);
         } else if (exData.getName().equals("RtcGatewayEventType")) {
            // Only process Start and End Sessions Types, all other
            // types will abort processing
            int valueInt = getValueInt(exData);
            if (valueInt == Event.START_SESSION_TYPE) {
               bStartSession = true;
               bCont = true;
            } else if (valueInt == Event.END_SESSION_TYPE) {
               bEndSession = true;
               bCont = true;
            } else {
               bCont = false;
            }
         } else if
```

```
            (exData.getName().equals("RtcGatewaySenderCommunity")){
               senderCommValue = getValue(exData);
         } else if
            (exData.getName().equals("RtcGatewayReceiverCommunity")){
               receiverCommValue = getValue(exData);
         }

         // We're done processing this Event when all required
         // fields have been processed.
         if (requiredFields(bStartSession, bEndSession,
               senderCommValue, receiverCommValue)) {
            bCont = false;
         }
      }

      if (requiredFields(bStartSession, bEndSession,
            senderCommValue, receiverCommValue)) {

         CommunityTracker commTrack = updateCounter(bStartSession,
            senderCommValue, receiverCommValue);

         // Only write count to log file every N changes
         // to the counter.
         if (commTrack != null && doLog(commTrack)) {
            System.out.println(
               "IBM Sametime Gateway Sample: IM open session Count: "
               + commTrack.getOpenSessionCount() +
               " for Community: " +
               commTrack.getCommunity1() + " and " +
               commTrack.getCommunity2());
         }
      }
   }
}
```

UpdateCounter, shown above, is tracking open sessions per external community. An event does not specify whether a community is internal or external and the community assigned to the RtcGatewaySenderCommunity or RtcGatewayReceiverCommunity can change depending on who initiated the session.

This is one difference between an event consumer and a message handler plug-in. Message handler plug-ins can use the Community Interface returned from getOriginatorCommunity. To solve this limitation, the method updateCounter creates a key for the counter by combining the SenderCommunity and ReceiverCommunity using a consistent rule. This meets the requirements of the Event Consumer Logger sample because only one internal community exists and concatenating the internal and external community (using alphabetical order) uniquely identifies the internal/external community pair.

```
int commCompare = senderCommValue.compareTo(receiverCommValue);
String key = (commCompare < 0) ? senderCommValue+receiverCommValue :
                     receiverCommValue+senderCommValue;
```

Event Consumer QueryOpenSessions Sample

Event consumers can

- Process notifications
- Perform queries and process historical event data from the persistent data store.

Queries can be the result of a user action or business logic. The QueryOpenSessions sample is initiated by a user action. Queries can also occur as part of an event retrieval initialization or for logging and reporting applications.

The QueryOpenSessions sample is related to the Event Consumer Logger sample which was tracking the count of open IM sessions per community. The Event Consumer Logger writes a status message to SystemOut.log every N changes to a session counter. This message contains the count of open sessions between an internal and external community.

For QueryOpenSessions, a simple JSP was developed for gathering input and displaying results. The internal and external community name from a status message is the required input. The output is a list of the five most recent IM sessions between the internal and external community.

The JSP will call the QueryDelegate class. The delegate will instantiate the QueryService which contains the business logic. The service has two methods:

- One performs the initialization
- The other performs the query

Query Initialization

Initialization includes creating a com.ibm.events.access.EventAccess object to perform the query.

The following code creates an instance of the event access session bean used to query the event service.

```
public void initEventAccess(){

Object eventAccessHomeObj = null;
   try {
      InitialContext initCtx = new InitialContext();
      eventAccessHomeObj =
         initCtx.lookup("ejb/com/ibm/events/access/EventAccess");
   } catch (NamingException e) {
      e.printStackTrace();
   }

   EventAccessHome eventAccessHome = (EventAccessHome)
       PortableRemoteObject.narrow(eventAccessHomeObj,
                                  EventAccessHome.class);

try {
      _eventAccess = (EventAccess) eventAccessHome.create();
   } catch (RemoteException e1) {
         e1.printStackTrace();
   } catch (CreateException e1) {
      e1.printStackTrace();
   } catch (EventsException e) {
      e.printStackTrace();
   }
}
```

Perform the query

The Event Consumer QueryOpenSessions sample uses the queryEventsByEventGroup method from the com.ibm.events.access.EventAccess object. This method accepts the following arguments:

- EventGroup

- EventSelector
- AscendingOrder
- MaxEvents

All the arguments are easy to provide except the eventSelector which can be a complex XPath query.

The JSP prompts the user for the two communities which uniquely define an internal / external community pair. The query defines both alternatives for RtcGatewaySenderCommunity and RtcGatewaySenderCommunity since the user might not know which community is internal or external. This will display the use of “and” and “or” in the XPath query. The parameters are marked in the query as community1 and community2 and replaced with the users input. The output of this sample will display the five most recent IM sessions and this is accomplished by adding a check for RtcGatewayEventType equal '6' which will only list startSession and filter all other Events.

An example of the query created by the sample is included below:

```
CommonBaseEvent[
((extendedDataElements[@name='RtcGatewaySenderCommunity' and @type='string' and
@values='community1'] and extendedDataElements[@name='RtcGatewayReceiverCommunity'
and @type='string' and @values='community2'])
or
(extendedDataElements[@name='RtcGatewaySenderCommunity' and @type='string' and
@values='community2'] and extendedDataElements[@name='RtcGatewayReceiverCommunity'
and @type='string' and @values='community1']))
and
extendedDataElements[@name='RtcGatewayEventType' and @type='int' and @values='6']
]
```

The method listCommunityChats is called by the queryDelegate and performs the query. Once the query is executed, all results are returned as an array of CommonBaseEvent in ascending order as specified in the query. The same code used in the event consumer samples above can be used to process the results.

```
   public ArrayList listCommunityChats(String origComm, String destComm,
                                      int count) {
      ArrayList results = null;

   try {
         // Create the query for different community alternatives
      String origCommQuerySend =
"extendedDataElements[@name='RtcGatewaySenderCommunity' and @type='string' and
@values='" + origComm + "']";

      String destCommQueryRec =
"extendedDataElements[@name='RtcGatewayReceiverCommunity' and @type='string' and
@values='" + destComm + "']";

      String destCommQuerySend =
"extendedDataElements[@name='RtcGatewaySenderCommunity' and @type='string' and
@values='" + destComm + "']";

      String origCommQueryRec =
"extendedDataElements[@name='RtcGatewayReceiverCommunity' and @type='string' and
@values='" + origComm + "']";

         // Create the query for StartSession
      String startSessionQuery = "extendedDataElements[@name='RtcGatewayEventType'
and @type='int' and @values='6']";

         // Create both orders, orig first or last.
      String origANDdest = "(" + origCommQuerySend + " and " + destCommQueryRec +
")";
      String destANDorig = "(" + destCommQuerySend + " and " + origCommQueryRec +
")";


      String query = "CommonBaseEvent[(" + origANDdest + " or " + destANDorig + ")
and " + startSessionQuery + "]";

         CommonBaseEvent[] events = eventAccess.queryEventsByEventGroup(
                              "All events", query, true, count);

         results = new ArrayList();
         for (int i = 0; i < events.length && i < count; i++) {
            StringBuffer oneResult = buildOneResult(events[i]);
            results.add(oneResult);
         }

      } catch (RemoteException e1) {
         e1.printStackTrace();
      } catch (EventsException e) {
         e.printStackTrace();
      }

      return results;
   }
```

Sample source directory structure

The Real-Time Collaboration SDK contains five samples. Three are message handler plug-ins and two are event consumers. All sources for the samples are distributed with the SDK in the EAR file, rtcgw_samplesEAR.ear. This EAR file is ready to be imported into the IBM Rational Software Development Platform.

Use the following steps to import the EAR file:

Table listing the steps to import the EAR file.

Step	Description
1	Launch **RAD**. Accept the default workbench location if none has been selected
2	Click **File** > **Import** > **EAR file**. Click **Next**.
3	Browse to the Sametime Gateway SDK and locate the file: **...\samples\rtcgw_samplesEAR.ear**
4	Accept the defaults
5	Click **Finish**

Note These steps duplicate information presented in the section, "Importing the samples into RAD."

The message handler plug-ins are contained in the project EJB Projects/rtcgw_samples in the packages:

- `com.ibm.collaboration.realtime.sample.plugin.helloworld`

- `com.ibm.collaboration.realtime.sample.plugin.chatlog`
- `com.ibm.collaboration.realtime.sample.plugin.presblock.`

The event consumers are contained in the project EJB Projects/rtcgw_samples in the packages:

- `com.ibm.collaboration.realtime.sample.eventconsumer.messagelogger`
- `com.ibm.collaboration.realtime.sample.eventconsumer.query.`

The message handle plug-ins requires rtc.gatewayAPI.jar and the event consumers require com.ibm.events.client.jar. Both jars can be found on the Sametime Gateway server in the following location:

- `<WebSphere Application Server>\AppServer\plugins\com.ibm.events.client.jar`
- `<WebSphere Application Server>\<RTC Gateway>\\rtc_gw_lib\rtc.gatewayAPI.jar`

These two jars must be added to the EJB Projects/rtcgw_samples Java Build Path, Libraries tab as External JARs… for the rtcgw_samples project. See the section, "*Add the jar files*" for more information.

Follow these steps to deploy the samples as an EAR file.

Table listing the steps to deploy the samples as an EAR file.

Step	Description
1	Click **File** > **Export** > **EAR** file. Click **Next**.
2	Select rtc.gatewaySamplesEAR as the EAR Project
3	Provide a destination directory
4	Select **Finish**

Installing and running the samples

The samples must be exported as an EAR file and installed before they can be used. The instructions are listed above. Since all five samples are packaged in a single EAR file the configuration for the event consumer must always be performed.

The message handler plug-in samples include diagnostic trace logging. The event consumer logger sample will log Common Base Events published by the Event logger message handler plug-in. This information is only written to trace.log, for example, app_server_root\profiles\RTCGW_Profile\logs\RTCGWServer\trace.log, when diagnostic trace is enabled. Review the Sametime Gateway help in the Sametime Information Center for more information on "Setting a diagnostic trace". Logging for the samples, com.ibm.collaboration.realtime.sample, must be enabled and set to All Message and Trace Levels.

The custom properties for the Event logger must be enabled first to specify which events to publish and are then consumed by the event consumer logger sample.

Creating an activation specification for the Message Driven Bean

Before you install the EAR file, you must create an activation specification for the message driven bean in WebSphere Application Server. This information is also available in the Sametime Gateway Information Center.

Follow these steps to create an activation specification for the message driven bean in WebSphere Application Server.

Table listing the steps to create an activation specification.

Step	Description
1	From the Integrated Solutions Console, click **Service Integration** > **Buses**.
2	Select **CommonEventInfrastructure_Bus**, and then click **Destinations**
3	Select the destination with the following name: **CommonEventInfrastructureTopicDestination**.
4	Click **Publication Points**.
5	Using a text editor, copy and paste the long name for use later. For example: **dibby.RTCGWServer.CommonEventInfrastructureTopicDestination@dibby.RTCGWServer-CommonEventInfrastructure_Bus**
6	From the Integrated Solutions Console, click Re**sources** **> JMS > Activation Specifications**.
7	In scope, select the server level. For example: **Node=dibby, Server=RTCGWServer**
8	Click **New**.
9	With **Default messaging provider** selected, click **OK**.
10	Type any name in the **Name** field. For example: **CEI_Topic_ActivationSpec**
11	For the **JNDI Name**, type: **jms/cei/TopicActivationSpec**
12	For the **Destination** type, select **Topic**.
13	For the **Destination JNDI Name**, type the following: **jms/cei/notification/AllEventsTopic**
14	For the Bus name, select **CommonEventInfrastructure_Bus**.
15	For the Subscription durability, select **Non-durable**.
16	For the Subscription name field, paste the long name that you copied in step 5. For example: dibby.RTCGWServer.CommonEventInfrastructureTopicDestination@dibby.RTCGWServer-CommonEventInfrastructure_Bus
17	For the **Client identifier** field, paste the portion that comes before the @ symbol. For example: **dibby.RTCGWServer.CommonEventInfrastructureTopicDestination**
18	For the Durable subscription home field, paste the portion that comes after the @ symbol. For example: **dibby.RTCGWServer-CommonEventInfrastructure_Bus**
19	Click **OK**, and then **Save**.

Installing the Sample EAR file

Follow these steps to install the EAR file.

Table listing the steps to install the EAR file.

Step	Description
1	From Integrated Solutions Console, click **Applications** > **Install New Application**.
2	Browse to the location of the sample EAR file
3	Accept the defaults provided by WebSphere Application Server and click **Next**.
4	Click **Next** again to go to the **Bind listeners for message driven beans** panel.
5	Select Sametime Gateway Samples as the EJB module and **Activation Specification Bindings**.
6	In the **Target Resource JNDI Name** field, type: **jms/cei/TopicActivationSpec**
7	For the **Destination JNDI Name**, type: **jms/cei/notification/AllEventsTopic**
8	For the **AuthenticationSpec authentication** alias, type an authentication alias if WebSphere Application Server security is enabled. Use your primary administrative user name that you created when you enabled administrative security.
9	Click **Next** and then **Finish**.
10	Review for errors and **Save** the master configuration. **Note**: The application must be started before using the samples.

Configuring the sample message handler plug-in samples

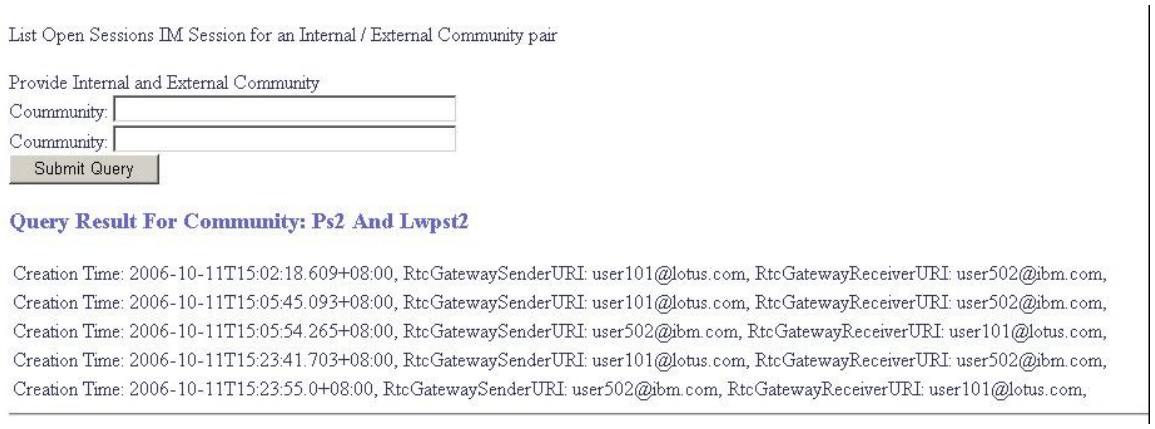
After the sample EAR file is installed the message handler plug-in samples must be enabled. Refer to the Sametime Gateway Information Center, "*Adding or removing a message handler,*" for instructions.

Accessing and configuring the Sample Query JSP

The sample query JSP can be accessed at the following URL:

```
http://yourRTCGateway@domain:9080/SampleQueryJSP/
```

Provide the two community names and click Submit Query button. The order of the communities is not important but the case must be exact. This will display the five oldest IM Chat sessions in the Common Event Infrastructure persistent store, including the sender and receiver URIs.


List Open Sessions IM Session for an Internal / External Community pair

Provide Internal and External Community

Community:

Community:

Query Result For Community: Ps2 And Lwpst2

```
Creation Time: 2006-10-11T15:02:18.609+08:00, RtcGatewaySenderURI: user101@lotus.com, RtcGatewayReceiverURI: user502@ibm.com,
Creation Time: 2006-10-11T15:05:45.093+08:00, RtcGatewaySenderURI: user101@lotus.com, RtcGatewayReceiverURI: user502@ibm.com,
Creation Time: 2006-10-11T15:05:54.265+08:00, RtcGatewaySenderURI: user502@ibm.com, RtcGatewayReceiverURI: user101@lotus.com,
Creation Time: 2006-10-11T15:23:41.703+08:00, RtcGatewaySenderURI: user101@lotus.com, RtcGatewayReceiverURI: user502@ibm.com,
Creation Time: 2006-10-11T15:23:55.0+08:00, RtcGatewaySenderURI: user502@ibm.com, RtcGatewayReceiverURI: user101@lotus.com,
```

Figure 8: Sample Query JSP

The query might fail because of security role mapping.

Follow these steps to define the security role for the eventConsumer.

Step	Description
1	From the Integrated Solutions Console, click Service Integration --> Common Event Infrastructure --> Event service
2	Click Map security roles to users or groups and set appropriate security role for eventConsumer.
3	Review for errors and Save the master configuration

Command- line scripts

In addition to programmatic interfaces, command-line scripts are available to access some functions of the Common Event Infrastructure. This includes useful scripts, like eventquery an eventpurge. More information can be found at WebSphere Enterprise Service Bus InfoCenter .

Chapter 8. Reference

This chapter outlines the different interfaces used throughout this guide.

- PluginRegistration
- PluginPresenceHandler
- PluginImHandler
- PluginSessionHandler

More information about message handler interfaces and events can be found in the javadoc in the Sametime Gateway Toolkit.

The PluginRegistration interface

The PluginRegistration interface provides awareness to Sametime Gateway plug-ins of their enablement status and their custom properties.

Methods of the PluginRegistration code

The following table lists the methods of the PluginRegistration and their function.

Table listing the methods of the PluginRegistration and their function.

Part	Function
init	This is called when the plug-in is being enabled or the Sametime Gateway is started.. Processing Options This notification is used to indicate that the plug-in should start to initialize itself if it requires lifecycle management. The callback must be called when this has finished. If the initialization is time consumptive then it should start a separate thread to do so. Why you would hook it All plug-ins must fulfill this method. At a minimum they need to call the callback to indicate they are ready.
terminate	This is called when the plug-in is being disabled or the Sametime Gateway is shutdown. Processing Options This has no parameters or return value. There are no processing options. Why you would hook it To be informed of plug-in disablement. You don't want to block this execution thread for very long. If you have longer termination procedures, you should spawn them off into other threads.
customPropertiesChanged	This is called when the custom properties for a plug-in are changed. Processing Options Properties may be read but not changed. Why you would hook it If your plug-in's behavior is determined by custom property settings, you will want to track these if they are changed on the system without disabling the plug-in.

The PluginPresenceHandler

This interface may be implemented by message handler plug-ins to receive notifications on presence functions passing through the gateway. All of the primary methods are passed the PresenceEvent interface. The on methods are passed the EventStatus interface.

Methods of the PluginPresenceHandler code

The following table lists the methods of the PluginPresenceHandler and their function.

Table listing the methods of the PluginPresenceHandler and their function.

Part	Function
subscribe	This is called when the originating user informs the destination community that they wish to receive notifications of the status of the destination user. Processing Options The PresenceEvent object can be interrogated for the values above. If you desire processing to be halted, you can set the status in the event to an error value. (See com.ibm.collaboration.realtime.GatewayReturnCodes). Why you would hook it To track relationships between users of different communities. To provide access control over who was allowed to be aware of presence between different communities
onSubscribe	This is called when the destination community has received the request to receive notifications and it has been accepted or rejected Processing Options The EventStatus object can be interrogated for the values above. Why you would hook it To log statistics of failure rates.
unsubscribe	This is called when the originating user no longer wishes to received notifications of the status of the destination user in the destination community. Processing Options The PresenceEvent object can be interrogated for the values above. If you desire processing to be halted, you can set the status in the event to an error value. (See com.ibm.collaboration.realtime.GatewayReturnCodes). Why you would hook it If you initiated particular tracking when subscribed, you could terminate that tracking at this point.
onUnsubscribe	This is called when the destination community has received the request to cancel notifications and it has been accepted or rejected. Processing Options The EventStatus object can be interrogated for the values above. Why you would hook it To log statistics of failure rates.
notify	This is called to notify the originating user of a status change of the destination user. Processing Options The PresenceEvent object can be interrogated for the values above. If you desire processing to be halted, you can set the status in the event to an error value. (See com.ibm.collaboration.realtime.GatewayReturnCodes). If you desire to filter the status

	you can reset the value passed back. Why you would hook it To mask or otherwise selectively filter who received which notification messages.
onNotify	This is called when the "originating" server has received and processed the notification change. Processing Options The EventStatus object can be interrogated for the values above. Why you would hook it To log statistics of failure rates.
fetch	This is called by the originating user to quickly retrieve the status of the destination user. In many cases this is performed after a time-out occurs to perform the operations needed to "resubscribe." Processing Options The PresenceEvent object can be interrogated for the values above. If you desire processing to be halted, you can set the status in the event to an error value. (See com.ibm.collaboration.realtime.GatewayReturnCodes). Why you would hook it To mask or otherwise selectively filter who received status update messages.
onFetch	This is called when "destination" server has received and responded to the request for status update. Processing Options The EventStatus object can be interrogated for the values above. Why you would hook it To log statistics of failure rates

PluginImHandler

This interface may be implemented by message handler plug-ins to receive notifications on instant messages passing through the gateway. All of the primary methods are passed the ImEvent interface. The on methods are passed an SessionEventStatus interface.

Methods of the PluginImHandler code

The following table lists the methods of the PluginImHandler and their function.

Table listing the methods of the PluginImHandler and their function.

Part	Function
instantMessage	This is called when an instant message is about to be sent from the originating community to the destination community between the given users in the referenced session. Processing Options The ImEvent object can be interrogated for the values above. If you desire processing to be halted, you can set the status in the event to an error value. (See com.ibm.collaboration.realtime.GatewayReturnCodes). If you desire to change or emend the message, you may change the value of it in the ImEvent object. Why you would hook it • To compile statistics on how many messages are being broadcast between communities. (e.g. usage rates)

	• To examine the text of messages sent between two communities, or by certain classes of users. (e.g. finance department can't talk about stock prices) • To stop communication of certain types of text between two communities. (e.g. intranet URLs cannot be exchanged to another community)
onInstantMessage	This is called when an instant message has been sent from the originating community to the destination community between the given users in the referenced session. Processing Options The EventStatus object can be interrogated for the values above. Why you would hook it To compile failure statistics on communication between communities.

PluginSessionHandler

The PluginSessionHandler interface may be implemented by message handler plug-ins to receive notifications on the start and end of IM conversation sessions. All of the primary methods are passed the SessionEvent interface. The on methods are passed an SessionEventStatus interface.

Parts of the PluginSessionHandler code

The following table lists the parts of the PluginSessionHandler and their function.

Table listing the parts of the PluginSessionHandler and their function.

Part	Function
startSession	This is called when someone wants to initiate a chat session with another in an external community. Processing Options The SessionEvent object can be interrogated for the values above. If you desire processing to be halted, you can set the status in the event to an error value. (See com.ibm.collaboration.realtime.GatewayReturnCodes). Why you would hook it You can use this to track session requests. Or To add some sort of control onto who could start sessions between specific communities, you could use this to check those criteria, and then return an error if access is not allowed.
onStartSession	This is called when a start session request has been passed to the destination community, and it has been accepted or rejected. Processing Options The EventStatus object can be interrogated for the values above. Why you would hook it • You could use this to track the start of sessions. • If you want to track elements of a session, you might use this to receive notification of the initiation so that you can then track the session ID. • To compile failure statistics, you could scan for failures and log them.
endSession	This is called when the user in the origin community wants to end a previously started instant message session. Processing Options

	The SessionEvent object can be interrogated for the values above. If you desire processing to be halted, you can set the status in the event to an error value. (See com.ibm.collaboration.realtime.GatewayReturnCodes). <u>Why you would hook it</u> You could use this to track end session requests. If you are tracking elements of a session, this lets you know you can retire active scanning for the session ID in question. Accumulated information can be logged to disk or discarded.
onEndSession	This is called when the end session request has been processed by the destination community. <u>Processing Options</u> The EventStatus object can be interrogated for the values above. <u>Why you would hook it</u>

Chapter 9. Conclusion

Instant Messaging (IM) has become another critical business tool. However, the widespread use of different translation protocols across multiple communities has impeded the growth of cross-organization IM exchanges. The Sametime Gateway rovides an enterprise ready solution that supports multiple protocols between internal and external communities as well as the ability to manage IM traffic between these communities.

In addition, by leveraging a modular architecture, built on top of WebSphere Application Server, the Sametime Gateway is extensible; IBM and third-party vendors, as well as your own internal development staff, can create solutions to manage IM traffic.

Notices

Trademarks