Sametime

Version 9.0.0

Sametime 9.0.0

Software Development Kit

Directory and Database Access Toolkit Developer's 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

About This Guide **5**
Intended Audience 5
Requirements 5
How to Use this Guide 5
Toolkit Contents 6
Related Documents 6
Additional Information 6

Chapter 1. Introduction **7**
Persistent Data in the Community Server 7
Directory Access 7
Database Access 9
Configuration Data Access 9
Thread Safety of Extensions 10
Unsafe Standard Library Functions 10
The Sametime ThreadSafe C++ Library 10

Chapter 2. Chat Logging SPI **12**
Architecture 12
Instant messaging 12
N-way chat 12
Place 13
Activities 13
Chat Logging Management 14
Modes 14
Chat Logging in a Distributed Environment 14
Getting Started 17
Chat Logging SPI Contents 17
Building the Chat Logging DLL or Service Program or Shared Library 18
Chat Logging SPI Sample 23
Chatlogging.ini Flags for the Chat Logging SPI Sample 23
Chat Logging SPI Sample Parameters 24
Chat Logging SPI Sample Code 26
Chat Logging SPI Reference 106
Chat Logging SPI Parameters 106
Chat Logging SPI Functions 107

Chapter 3. Token Authentication SPI **125**
Getting Started 125
Token Authentication SPI Contents 125
Building the Token Authentication DLL or Service Program 126
General Considerations 126
Building and Installing a Token Authentication DLL for Windows 126
Building and Installing a Token Authentication Service Program for IBM i 127
Token Authentication SPI Reference 129

Token Authentication SPI Parameters....129
Token Authentication SPI Functions....130

Chapter 4. File Transfer SPI....134
Architecture....134
Instant Messaging....134
File Transfer Management....134
Modes....134
Multi-Thread Running....135
Synchronous Implementation....135
Configuration and Policy....135
Error Handling....136
Effect of Virus-Scanning on Performance....137
Getting Started....138
File Transfer SPI Contents....138
Building the Virus-Scanning DLL....138
General Considerations....138
Building and Installing for Windows....138
Building and Installing for IBM i....139
File Transfer SPI Parameters....139
Input Parameters....139
Return Messages....140
Constants....140
File Transfer SPI Functions....141
stDdaFtInit Function....141
stDdaFtScanFile Function....141
StDdaFtTerminate Function....141

Chapter 5. User Information SPI....143
UserInfo Application....143
Operations....143
Configuration Data in UserInfoConfig.xml....144
Working with Black Boxes....146
User Information API....147
Configuring the Servlet to Use the New Black Box....147
Admin Pages and UserInfoConfig.xml....147
Domino Configuration....149

About This Guide

This guide explains the IBM® Lotus® Sametime® Directory and Database Access Service Provider Interface (SPI).

Intended Audience

This guide is intended for developers deploying Sametime who want to customize the access level to the organization directory and/or database.

Requirements

For Information about software requirements for the Sametime Directory and Database Access Toolkit, see releaseNotes.txt, included with this toolkit.

How to Use this Guide

This guide contains the following main sections.

Table listing chapter number and title with a description of each chapter.

Chapter title	**Description**
Chapter 1: Introduction	Provides an overview of the logic and access layers.
Chapter 2: Chat Logging SPI	Provides an explanation of the Chat Logging SPI and its management. Includes function reference guide.
Chapter 3: Token Authentication SPI	Describes the Token Authentication SPI. Includes function reference guide.
Chapter 4: File Transfer SPI	Describes the File Transfer SPI. Includes function reference guide.
Chapter 5: User Information SPI	Describes the User Information SPI and the process of retrieving users' data from data storages.

Toolkit Contents

Documentation – Sametime Directory and Database Access Toolkit Developer's Guide (this document)

Chat Logging SPI sample

Chat Logging SPI template

Chat Logging SPI header files

Token Authentication SPI template

Token Authentication SPI header files

File Transfer SPI sample

User Information SPI sample

User Information SPI Javadoc

Related Documents

IBM Lotus Sametime Installation and Administration Guide

Additional Information

For more information on working with Sametime servers and other features described in this guide, refer to the Sametime information center.

See these Web sites for more information:

http://www.lotus.com/sametime

http://www.ibm.com/developerworks/lotus

You may also want to read "Working with the Sametime Client Toolkits," available at the IBM Redbooks site (http://www.redbooks.ibm.com).

Chapter 1. Introduction

As part of the Sametime server, the Community server provides services for awareness, chat, and meeting places.

Persistent Data in the Community Server

The Community server uses persistent data storage to save and retrieve persistent information in the implementation of these services.

This information can be divided into the following types:

- **Directory information** – Sametime requires access to an organization directory in order to identify different users, authenticate users, allow users to search in the directory, and perform other such actions. This directory may already exist and may be in use by other applications. Sametime requires only read access to this directory.
- **User information** – This is information about users that is specific to Sametime, and is written and used by Sametime.
- **Administration information** – This information is specific to Sametime and describes the community server(s) in the organization. It includes configuration details, log information, and other information that is used by the servers or is needed by Sametime administrators.

Access to the databases where this information is stored is implemented through the following layers:

- **Logical layer** – Implements the logic of accessing the databases without any dependency on the type of database that is used.
- **Access layer** – Implements the access to a specific database.

The logic layer is implemented once. The access layer is implemented separately for each database type. Sametime includes an implementation of this layer. Sametime also publishes the definition of the Chat Logging and Token Authentication SPIs so customers can adopt them to their environments.

Directory Access

The directory stores the organization's user information. You can deploy Sametime so it uses your existing organization directory or so it creates and uses its own built-in directory. If you already have a directory in the organization, it is assumed that you will want to use that directory.

You can use an existing directory in one of the following ways:

- **Using the "out of the box" access layer supplied by the Sametime product** – If you have a Lotus Notes® directory (NAB) or an LDAP directory, Sametime provides an access layer that functions with these directory types. If you are using an LDAP directory, you might need to tune the connection but you don't need to customize the access layer.
- **Customizing the access layer –** If you are not using a Notes directory or an LDAP directory, or if you are using an LDAP directory but would like to implement your own access layer, you can customize the access layer to do so.

Sametime servers only read information from the directory; they do not write it. Because they do not make any changes to the directory, you can safely use your organization's existing directory.

Sametime uses the directory for the following services:

- **Authenticating users** – Confirms that whoever is trying to log in to the Sametime community is authorized to do so.
- **Searching** – Allows people to search in the directory for other people or groups in the organization and add them to their contact list. When the end-user types a name or a part of a name, the server searches for matches in the directory.
- **Browsing** – Allows people to browse the contents of the directory. This service is complimentary to the search service. Browsing is used when the end-user knows part of the name and would like to find the correct match by browsing the entire directory.
- **Groups** – Provides the service for retrieving the contents of a public group.

In addition to the directory-related services above, there is one other service that is related to the directory information:

- **Token Authentication** – A component that handles token generation and verification. The Token Authentication SPI is included in this toolkit.

Sametime supports two types of authentication tokens out-of-the-box:

- Proprietary Sametime token
- LTPA token supported by Domino™ and WebSphere®

The Token Authentication SPI allows you to customize Sametime for a different kind of token generated in your deployment.

Database Access

The user information database contains Sametime-specific information about users. In addition to the user information in the organization directory, Sametime stores and uses this additional information about each user. This information is saved on the server to allow the user access to the information from any computer.

The user information is stored in a Notes database in a regular deployment; this storage is considered the "out of the box" implementation of the access layer. You can customize this implementation for a different database storage system, such as any SQL server.

Sametime stores two types of additional user information:

- **Privacy information** – Stores and retrieves the privacy information of a user.
- **User preferences** – Stores and restores the preferences of the user, including the contact list, messages, alarms, and so on. The only part of the user preference not stored on the server is the connectivity information.

The interface between the logical layer and the access layer allows you to set and retrieve the above information.

Configuration Data Access

Administration information databases contain information for configuration and logging settings. These databases are used by the administrator to modify the configuration of the community and to monitor the status of the servers. These databases include:

Server configuration – Contains information about Sametime servers and their configuration parameters. You can use the Sametime administration tool to modify this data. If you change the location where you store this information, you must then provide an alternative method for modifying the information.

Server log – Records events from the server. The set of events is defined in advance. You can choose which to record and which not to record.

Chat logging – Stores the chat transcripts for future reference. Administrators and supervisors can review chats even if they did not participate in the chat. The Chat Logging SPI is included in this toolkit. The toolkit can also be used to log the files being transferred through the server.

Thread Safety of Extensions

As many other products, all Sametime Server processes are multi-threaded.

In order for extensions not to compromise product stability or functionality, all extensions have to be coded in a thread safe manner, assuming the process is multi-threaded regardless of whether the extension creates additional threads.

In addition, extensions have to comply with the Sametime thread safety mechanism, which provides a safe way to use library functions not guaranteed to be safe.

Unsafe Standard Library Functions

Standards do not require all standard library functions to be thread-safe.

While some software vendors declare their libraries safe, not all are, and for the sake of uniform and portable stability, we recommend not to assume anything about the stability of those functions not guaranteed by the standards to be safe.

The following functions are known to be unsafe in some platforms:

- Environment-related, such as: `getenv()` and `putenv()`.
- Static storage return, such as time-related: `localtime()`, `gmtime()`, `asctime()` and `ctime()`.
- Functions relying on the environment to perform localization, such as those relying on the value of the TZ environment variable: `localtime_r()`, `gmtime_r()`, `asctime_r()` and `ctime_r()`.
- Any other, standard or 3rd party library function known to be calling one of the categories mentioned above.

The Sametime ThreadSafe C++ Library

The STTS_GUARD Macro

Protect every line of code calling an unsafe function by using the `STTS_GUARD()` macro inside a separate block of code, e.g.:

```
void mySafePutenv(const char* str)
{
     STTS_GUARD();
     return putenv(str);
}
```

Protect every line of code calling a static storage unsafe function, along with its usage, by using the `STTS_GUARD()` macro inside a separate block of code, e.g.:

```
void mySafePrintYear(time_t secondsSinceEpoch)
{
     int year;
     {
          STTS_GUARD(); //BEGIN SAFE BLOCK
          struct tm* time = localtime(&secondsSinceEpoch);
```

```
          year = time->tm_year;
//END SAFE BLOCK
     }
     printf("Year is %d.\n", year);
}
```

Avoid excessive locking, e.g. around blocking calls or heavy computation; for example:

```
void myBadGetHomedirAndSleep(char* out, size_t len)
{
     STTS_GUARD();
     const char* home = getenv("HOME");
     strncpy(out, home, len);
     sleep(10); //BAD: May hang all other threads
}
```

Instead, limit the guard to the exact necessary block, e.g.:

```
void myGoodGetHomedirAndSleep(char* out, size_t len)
{
     {
          STTS_GUARD();
          const char* home = getenv("HOME");
          strncpy(out, home, len);
}
     sleep(10); //OK: We are not holding the lock
}
```

How to Build Code Using This Library

In every source file where the `STTS_GUARD()` macro is used, include the following file:

```
#include <StThreadSafe.h>
```

Make sure to have the …`/dda/inc/common` dir in the include path.

When linking, use the STTS library: stts.lib for Windows, libstts.so on Unix-like platforms.

Make sure to have the appropriate platform under the …`/dda/lib/`*`OS`* dir in the link path.

Chapter 2. Chat Logging SPI

The Sametime Chat Logging Server Provider Interface (SPI) is a server-side Sametime feature. This feature can intercept all chat conversations at the server and store a record of them in a data store for future retrieval and reference. The intercepted conversation includes both text and data messages. The Chat Logging application can also apply content filters to the instant messages, and block session creation. The feature is provided as an incentive for developers to create their own DLL (or service program, for IBM i users) for storing chats. The DLL or service program you create implements a predefined SPI used by the server. As the developer, it is your responsibility to store the chats in a data store of your choice so you can retrieve and display them later.

The Chat Logging SPI sample is an example of the chat logging implementation and can be used as a template for building your customized chat logging DLL or service program.

Architecture

Sametime contains two mechanisms for text conversations between users:

- Instant messaging (IM)
- N-way chat and meetings
- Offline messages

When an entity (a user, server, or a server application) needs to communicate with another entity, it creates a channel to the other entity. The channel can be encrypted if necessary. The recipient can choose to accept or reject the channel. For more information on channel encryption, see the "___" and the "___" sections that follow.

Instant messaging

Instant messaging allows two people to exchange messages that they type via their computer keyboards. Instant messages are exchanged between two people only. When user A wants to have a chat with user B, user A selects user B, which causes the server to create a channel to user B. If approved, the server connects the two users and thereafter acts only as a router, transferring the messages from one user to the other. In Sametime 3.0 and above, the messages are encrypted by default. Starting from Sametime 8.5.2 any data is being sent from user A to user B and vice versa has been snatched by ChatLogging server application and then passed to ChatLogging BlackBox. The BlackBox invokes the DLL APIs to log the data.

N-way chat

An n-way chat is the exchange of online messages among more than two users. The messages are encrypted by default. An n-way chat is created when:

Two users in an instant meeting (IM) invite an additional user to the conversation.

One user invites at least two other users to a chat.

While instant messages are only one-to-one, n-way chats use a different mechanism called a "Place" to exchange messages among multiple participants.

Place

A Place is a server-side component where users meet and exchange messages. Users who want to participate in the conversation must enter the Place. Upon joining a Place, the users are assigned to sections. Each Place has one or more sections, and each section has zero or more users. Having sections enables the client applications to allow the user to send messages to different participants or scopes. (The scope refers to the number of participants who will receive a message.) The types of scopes are:

- Place scope – When a message is sent to the Place scope, all the participants in the Place receive it.
- Section scope – When a message is sent to the section scope, only the participants of the section receive it.
- Individual scope – When a message is sent to a specific participant in a Place, only that individual receives the message.

Activities

Activities represent server-side components that add value, content, and activity to the Place. They are a type of participant in a Place and, as participants, can send messages and receive messages that are sent to them or to the Place scope.

Logging announcements

The Chat Logging application can also log announcements. Announcements are text messages sent from the server or from a user to one or more users. The announcements are treated as a session with just one message sent to a number of users. It can be distinguished from a regular chat because the session id is prefixed with the string ANNOUNCE_.

Chat Logging Management

Chat logging management requires that you understand how chat logging works. The following topics provide a more detailed understanding of chat logging:

- Modes
- Chat logging in a Distributed Environment
- Synchronous and Asynchronous Implementation
- Chat Content Monitoring
- Error Handling
- Chat Logging Effect on Sametime Server Performance

Modes

The two modes of chat logging are:

- Strict chat logging – If the chat logging service is not present or an error message about the Chat Logging SPI functions is received, all existing chats and chats in Places will be disconnected, and users will not be able to start new chats.
- Relaxed chat logging – As long as the chat logging service is running, chats are logged, however if error message about the Chat Logging SPI function is received then the chats are not logged, but can be created or continue running.

Note See the section on Administration below for a more detailed explanation on how to configure the server correctly, in order to achieve the requested behavior.

Chat Logging in a Distributed Environment

When using the chat logging feature in a distributed or multi-server environment, an IM or chat in a Place can occur between two users connected to different Sametime servers. A distributed Sametime environment contains more than one Sametime server. Installing multiple Sametime servers provides several advantages related to load balancing and network usage, and can enhance meeting and server performance.

The chat transcript is logged for each server.

In an N-way chat or a meeting, the location of the server on which the chat is logged is the server on which the Place is handled. Other options are available through advanced configuration.

In an instant messaging session, the chat is logged onto the home server of each IM participant. This can be either on each server or only on the recipients according to server configuration.

Note If user A initiates a chat with user B, the chat is logged in the home server of the chat's recipient – user B. If in the course of the chat, user A closes the chat window, but user B continues to write, a new chat session is created in which user B is the initiator and user A is the recipient. From this point on the chat is logged on the home server of user A.

Synchronous and Asynchronous Implementation

A chat logging DLL or service program can be implemented asynchronously (the message is sent regardless of its logged state) or synchronously (the message is not sent until it is logged successfully). Sametime provides the Chat Logging SPI sample for synchronous implementation. You can develop your own chat logging DLL or service program with asynchronous implementation.

Chat Logging mechanism

The chat logging mechanism for chats in IM is the same as for *n*-way chats and meetings. Each chat message string is logged into the data store before it is forwarded to the recipient. If the chat logging DLL or service program is implemented synchronously, the chat message string is not forwarded to the recipient until this string is successfully logged in the data store. This delay slows down the chat operation. Asynchronous implementation of the chat logging DLL or service program does not affect the IM operation because the chat message string is forwarded to the recipient without waiting for it to be logged to the data store.

Retrieving Chats

The purpose of chat logging is to be able to later search the logged conversations and read them. Sametime does not provide such a tool. You will need to develop the tool that searches the data store and retrieves the logged conversations.

Administration

Use the Sametime System Console to configure chat logging as follows.

1. Select the relevant Sametime server, then select the CommunityServices tab.
2. Enable chat logging by choosing one of the following values for the Chat Logging Flag field:
   - Off – No chat logging will occur.
   - Relaxed – Chat logging will occur; if it does not work for any reason, chat is still enabled.
   - Strict – Chat logging will occur; if it does not work for any reason, chat is disabled.

     Note If chat logging is not enabled, the chat logging server application will not function.
3. For the Capture Service Type field, specify 0x1000 for chat logging in strict mode, in this case the server will enforce the chat logging, if application is not running then the server will decline any chats channels.

     Note In case of Relaxed mode "Capture Service Type" field must be set to 0, if it will be set to 0x1000, then the server will identify it as strict mode and enforce the service, once a chat is requested and chat logging service is not available – channel will be declined.

4. Restart the Sametime and Domino Server so that your changes will take effect immediately.

In a distributed community, the administrator should configure all the servers at once and configure them all the same way: either all have chat logging turned on or all do not. Also, they should all be running in the same mode: either relaxed mode or strict mode. (See the “Modes” section for more information.)

Note If all the servers are not running in the same mode, some chats will be logged and others will not. The logging occurs arbitrarily in this situation.

Chat Content Monitoring

The Chat Logging SPI can pass back information about the content changes, forced in the message. The SPI provides ability to the Chat Logging library to block messages or block chat initialization. The decision to block the chat can be based on the chat content or list of chat participants. The Chat Logging library can return a special Error Code, which will indicate that the original message should be filtered out or changed, and a new message (or several messages) should be sent instead.

Error Handling

The Chat Logging SPI can pass back information about its status in the form of a code and a message. Two variables are defined globally inside the calling application:

stDdaRetVal – An enumerator for the return state of the most recent SPI call.

stDdaRetStr - A character buffer for a string with a textual explanation of the return code.

The pointers to these variables are passed by the application to the Chat Logging SPI in the initialization call. For more information, see the "Chat Logging SPI Functions" section.

The table below defines the possible values for the enumerator stDdaRetVal:

Table listing the possible values for the enumerator stDdaRetVal .

Value	Description
ST_DDA_OK	The operation was completed successfully.
ST_DDA_INFO	The operation was completed successfully, but there is an informative message.
ST_DDA_WARNING	The operation was completed, but a warning was issued.
ST_DDA_ERROR	The operation failed.
ST_DDA_FATAL	The operation failed and the application should stop calling the specific SPI.
ST_DDA_SESSION_CREATION_DECLINED	The chat session initialization was declined by the Chat Logging library.
ST_DDA_SESSION_CLOSED_BY_SERVER	The chat session was closed by the Chat Logging library.
ST_DDA_MSG_CHANGED	The chat logging changed the original message. The new messages should be sent instead of the original message.

It is the responsibility of the developer implementing a chat logging DLL or service program to make sure that information is not lost. For example, if an informational message is created and later a fatal error occurs, the informational message might hold data crucial for understanding the cause of the fatal

condition but is not saved because of the fatal error. The messages should be concatenated to avoid overwriting of messages and to retain all message information.

Chat Logging Effect on the Sametime Server Performance

The effect of chat logging on the Sametime server depends on the chat logging operation and implementation modes. See the "Modes" and "Synchronous and Asynchronous Implementation" sections of this chapter for more information.

Chat Logging Operation Mode

In relaxed mode, if the chat logging application does not work, the malfunction will have no effect on the Sametime server. Users can create new Places and new IMs and send invitations; the existing Places and IMs will continue to function.

In strict mode, if one of the Chat Logging SPI functions returns errors, all existing IMs and chats in Places will be destroyed.

Chat Logging Implementation Mode

If the Chat Logging SPI is implemented synchronously, the operation of chats, n-way chats and meetings in instant messages is slower than usual.

If the Chat Logging SPI is implemented asynchronously, the operation of the chats in both instant messages and Places is not affected.

Getting Started

The Chat Logging SPI is part of the Directory and Database Access Toolkit. You can extract the files for this toolkit either on your local machine or (to make it available to other users) on your Sametime server.

To access the toolkit pages that include the toolkit documentation, samples, and binaries, open readme.html in the toolkit root folder.

Chat Logging SPI Contents

The contents of the Directory and Databases Access Toolkit related to the Chat Logging SPI are:

Sametime Directory and Database Access SPI documentation – This document.

Chat Logging SPI sample – This sample provides a file version example of the Chat Logging SPI implementation. It provides examples of constants, error messages, and chatlogging.ini configuration. In the file version, each chat is logged in a separate text file, located under a folder defined in the chatlogging.ini file. If screenshots were sent during the chat, each screenshot is saved to a separate file. For more information on the Chat Logging SPI sample, refer to the Chat Logging SPI Sample section of this document.

Chat Logging SPI template – This template or "dummy" version of the Chat Logging SPI does not return error messages (all return codes are OK) and does not log chat transcripts. Use this template version for building a new chat logging SPI implementation.

Chat Logging SPI header files – These files contain SPI definitions, syntax, variables, constants, return values, and so forth.

Building the Chat Logging DLL or Service Program or Shared Library

This toolkit does not provide a full version of the chat logging DLL or service program. Use the Chat Logging SPI template and the Chat Logging SPI sample to complete development of the chat logging SPI for your particular environment(s).

When you create a chat logging DLL or service program:

Follow the interface defined by the Chat Logging SPI.

Develop a mechanism for storing the text and data messages, and a mechanism for retrieving them at a later date.

To ensure proper operation of the chat logging DLL or service program, see the following topics:

- General Considerations
- Building and Installing a Chat Logging DLL for Windows
- Building and Installing a Chat Logging Shared Library for Linux
- Building and Installing a Chat Logging Service Program for IBM i
- Recommendations

General Considerations

The following chat logging features must be considered when planning the chat logging implementation.

When using chat logging in a distributed community:

- All servers must run the same way: either all have chat logging turned on or all have chat logging turned off.
- All servers must run in the same mode: either relaxed or strict.

  Note If all the servers are not running in the same mode, some chats will be logged and others will not. The logging occurs arbitrarily in this situation.

With an instant message, a chat is logged on the home server of the chat's recipient.

Synchronous implementation of the Chat Logging SPI affects the speed of chats in instant messaging and of chats in Places. For more information about the synchronous and asynchronous implementation of the Chat Logging SPI, see the Synchronous and Asynchronous Implementation section of this chapter.

When using chat logging in a hosted environment where multiple organizations use a single Sametime server, the stDdaClSessionStartedByOrgName function allows the SPI developer to perform certain preparations for new session logging. This function was added in Sametime 2.6 and is not required to be implemented in non-hosted environments. See the stDdaClSessionStartedByOrgName function for more information.

Building and Installing a Chat Logging DLL for Windows

Follow these general steps to build and install a chat logging DLL:

1. Study the provided sample, StChatLogFile.dll in the toolkit samples\chatlogging directory. For more information about the sample, refer to the Chat Logging SPI Sample section of this chapter.
2. Use the template version in the toolkit samples\chatlogging directory to create a new Chat Logging SPI implementation.

   Note: When you build your chat logging DLL, leave the sample unchanged.
3. When you create a new DLL, use the debug mechanisms and make sure debug messages are being saved in the trace files to facilitate troubleshooting during the development and use of the DLL.
4. Save the new DLL under the name StChatLog.dll.
5. Test the created DLL.

   Note: Do not install a new chat logging DLL until you have tested it thoroughly; DLLs directly affect the operation of server components.
6. Replace the dummy chat logging DLL on the Sametime server with the newly created DLL (StChatLog.dll). The new DLL must be placed in the default directory that contains all other DLLs. The default directory is C:\Sametime. If more than one server is included in the community, replace the DLL for each server.
7. Make the necessary changes in the configuration of each Sametime server in the community. For more information, see the Administration section of this chapter.
8. To activate the new chat logging DLL you created, start and stop the Sametime server on which the DLL is installed. If the Sametime community contains more than one Sametime server, start and stop all servers in the community.

Building and Installing a Chat Logging Shared Library for Linux

Follow these general steps for building and installing a chat logging .SO:

9. Study the provided sample, libChatLogging.so in the toolkit samples/chatlogging directory. For more information about the sample, refer to the "Chat Logging SPI Sample" section of this chapter.
10. Use the template version in the toolkit samples/chatlogging directory to create a new Chat Logging SPI implementation.
11. Build using the `make` command in the following way. makefile is present in the same folder as sources (samples/chatlogging). The libstChatlogging.so will be created in the same directory as the makefile.

    ```
    make clean all
    ```

    Note: When building your chat logging .SO, leave the sample unchanged

    When creating a new .SO, use the debug mechanisms and make sure debug messages are being saved in the trace files to facilitate troubleshooting during the development and use of the .SO.
12. Test the created .SO.

Note: Do not install a new chat logging .SO until you have tested it thoroughly; .SOs directly affect server components operation.

13. Replace the dummy chat logging .SO on the Sametime server with the newly created .SO (libChatLogging.so). The new .SO must be placed in the default directory that contains all other .SOs. The default directory is /opt/ibm/lotus/notes/latest/linux. If more than one server is included in the community, replace the .SO for each server. Make sure that the owner and the group for the newly copied .so are “notes” and “notes” respectively. If not, login as root and run the following commands:

```
chown notes libChatLogging.so
chgrp notes libChatLogging.so
```

14. Similarly check the permissions for the .so. It should –rwxrwxr-x. If not set properly, change the permissions in the following way.

```
chmod 775 libChatLogging.so
```

15. Make the necessary changes in the configuration of each Sametime server in the community. For more information, see the "Administration" section of this chapter.
16. To activate the created chat logging .SO, start and stop the Sametime server on which server, where the .SO is installed. If the Sametime community contains more than one Sametime server, start and stop all the servers in the community.

Starting and stopping a Sametime server on Linux

Issue a quit command on the domino console.

When the server stop completes, restart the server using ./ststart

Building and Installing a Chat Logging Service Program for IBM i

It is best to follow these general steps for building and installing a chat logging service program:

Study the source code provided in the toolkit samples\chatlogging directory. In this Chat Logging SPI sample, each chat transcript is logged in a separate text file under a directory defined in the chatlogging.ini file. For more information about the sample, refer to the “Chat Logging SPI Sample” section of this chapter.

To create a new custom Chat Logging SPI implementation, use the template version in the toolkit templates\chatlogging directory.

When you create a new service program, use the debug mechanisms and make sure debug messages are being saved in the trace files to facilitate troubleshooting during the development and use of the service program.

To build and install the Chat Logging SPI sample that logs each chat transcript to a text file under a directory defined in the chatlogging.ini file, do the following:

1. Copy the following toolkit files from your local machine to a directory (for example, “/STToolkit”) on your IBM i:

```
samples\chatlogging\ChatResource.cpp
inc\chatlogging\ChatResource.h
samples\chatlogging\ChatSessionTable.cpp
inc\chatlogging\ChatSessionTable.h
```

```
samples\chatlogging\Disclaimer.cpp
samples\chatlogging\Image.cpp
samples\chatlogging\ParseDataMsg.cpp
samples\chatlogging\RichTextMgr.cpp
samples\chatlogging\stDdaClApi.cpp
samples\nonwin32\utilities.cpp
samples\nonwin32\utilities.h
samples\nonwin32\debug.h
samples\nonwin32\winprofile.cpp
inc\chatlogging\stDdaClApi.h
inc\chatlogging\stDdaClCodes.h
inc\chatlogging\Disclaimer.h
inc\chatlogging\Image.h
inc\chatlogging\ParseDataMsg.h
inc\chatlogging\RichTextMgr.h
templates\authtoken\stAuthTokenApi.cpp
inc\authtoken\stAuthTokenApi.h
inc\common\stDdaApiDefs.h
inc\common\nonwin32\windows.h
```

2. Add the ASCII C/C++ Runtime Development kit to your IBM i system. Refer to the following Web site:

   http://www-03.ibm.com/servers/enable/site/asciirt/

3. Create a library to hold your custom service programs and modules.

```
CRTLIB CHATAPILIB
```

4. Compile the chatlogging source files using the following commands:

```
CRTCPPMOD MODULE(CHATAPILIB/CHATRSC)
SRCSTMF('/STToolkit/ChatResource.cpp') DBGVIEW(*SOURCE)
DEFINE('QSRCSTMF' '_UNIX' 'OS400' 'qadrt_use_ctype_inline'
'V5R2_COMPILER' 'UNIX_TOOLKIT_COMPILE') TERASPACE(*YES)
STGMDL(*INHERIT) INCDIR('/STToolkit'
'/qibm/proddata/qadrt/include') TGTCCSID(819)

CRTCPPMOD MODULE(CHATAPILIB/CHATSESTBL)
SRCSTMF('/STToolkit/ChatSessionTable.cpp') DBGVIEW(*SOURCE)
DEFINE('QSRCSTMF' '_UNIX' 'OS400' 'qadrt_use_ctype_inline'
'V5R2_COMPILER' 'UNIX_TOOLKIT_COMPILE') TERASPACE(*YES)
STGMDL(*INHERIT) INCDIR('/STToolkit'
'/qibm/proddata/qadrt/include') TGTCCSID(819)

CRTCPPMOD MODULE(CHATAPILIB/UTILITIES)
SRCSTMF('/STToolkit/utilities.cpp') DBGVIEW(*SOURCE)
DEFINE('QSRCSTMF' '_UNIX' 'OS400' 'qadrt_use_ctype_inline'
'V5R2_COMPILER' 'UNIX_TOOLKIT_COMPILE') TERASPACE(*YES)
STGMDL(*INHERIT) INCDIR('/STToolkit'
'/qibm/proddata/qadrt/include') TGTCCSID(819)

CRTCPPMOD MODULE(CHATAPILIB/STDDACLAPI)
SRCSTMF('/STToolkit/stDdaClApi.cpp') DBGVIEW(*SOURCE)
DEFINE('QSRCSTMF' '_UNIX' 'OS400' 'qadrt_use_ctype_inline'
```

```
'V5R2_COMPILER' 'UNIX_TOOLKIT_COMPILE') TERASPACE(*YES)
STGMDL(*INHERIT) INCDIR('/STToolkit'
'/qibm/proddata/qadrt/include') TGTCCSID(819)

CRTCPPMOD MODULE(CHATAPILIB/WINPROFILE)
SRCSTMF('/STToolkit/winprofile.cpp') DBGVIEW(*SOURCE)
DEFINE('QSRCSTMF' '_UNIX' 'OS400' 'qadrt_use_ctype_inline'
'V5R2_COMPILER' 'UNIX_TOOLKIT_COMPILE') TERASPACE(*YES)
STGMDL(*INHERIT) INCDIR('/STToolkit'
'/qibm/proddata/qadrt/include') TGTCCSID(819)
```

5. Create a service program called STCHATLOG in library CHATAPILIB:

```
CRTSRVPGM SRVPGM(CHATAPILIB/STCHATLOG) MODULE(CHATAPILIB/CHATRSC
CHATAPILIB/CHATSESTBL CHATAPILIB/UTILITIES CHATAPILIB/STDDACLAPI
CHATAPILIB/WINPROFILE) BNDSRVPGM(QADRTTS) EXPORT(*ALL)
OPTION(*DUPVAR *DUPPROC)
```

6. Change the object owner of the new STCHATLOG service program to QNOTES:

```
CHGOBJOWN OBJ(CHATAPILIB/STCHATLOG) OBJTYPE(*SRVPGM)
NEWOWN(QNOTES)
```

7. Remove the symbolic link /QIBM/Userdata/lotus/notes/STCHATLOG.SRVPGM via the CL command:
   `RMVLNK OBJLNK('/qibm/userdata/lotus/notes/STCHATLOG.SRVPGM')`

8. Create a new link /QIBM/Userdata/lotus/notes/STCHATLOG.SRVPGM that points to your new STCHATLOG service program in CHATAPILIB via the CL command:

```
ADDLNK OBJ('/qsys.lib/CHATAPILIB.lib/STCHATLOG.SRVPGM')
NEWLNK('/qibm/userdata/lotus/notes/STCHATLOG.SRVPGM')
```

9. Change the authority on the new link via the CL command:

```
CHGAUT OBJ('/QIBM/Userdata/lotus/notes/STCHATLOG.SRVPGM')
USER(QNOTES) DTAAUT(*RWX) OBJAUT(*ALL)
```

10. Make the necessary changes in the stconfig.nsf file of each Sametime server in the community. For more information, see the Administration section of this chapter.

11. To activate the new chat logging service program you created, stop and start the Sametime server on which the service program is installed. If the Sametime community contains more than one Sametime server, stop and start all servers in the community.

For more information on stopping and starting Sametime servers, refer to the Sametime

Recommendations

When building the chat logging DLL, follow the provided template and sample as closely as possible. Follow these additional recommendations to ensure the success of your application.

Chat Logging SPI functions

Follow these standards to ensure proper operation of the Chat Logging SPI:

The value of the libversion parameter is saved in the Sametime log file (sametime.log). Every time the Chat Logging SPI version is changed, its number is incremented and the information in the Sametime log file is updated accordingly. Use this parameter to keep track of the used versions. See the stDdaClInit function in the Chat Logging SPI Functions section for more information on the libversion parameter.

The retcode and retmsg parameters provide information about the application status. Retcode is returned from each function and a retmsg matches the specific retcode. The value of retmsg is saved in the log or trace files. Use this standard to continue receiving updates on the statuses of the application. See the appRetCode parameter of the stDDaClinit function in the Chat Logging SPI Functions section for more information.

Note To prevent information loss, concatenate the messages.

Logging and retrieving chat transcripts

The chat transcripts can be stored in a database, in a file, or in a set of files. It is best to run a global directory of all logged chats or log chat transcripts in one database, which is replicated by the servers.

It is possible to develop a mechanism for filtering the messages and chat transcripts so that only predefined chats are logged.

Chat Logging SPI Sample

The Chat Logging SPI sample provides an example of synchronous implementation of the Chat Logging SPI feature. It can be found on the Directory and Database Access SPI Toolkit home page. For more information, see the Chat Logging SPI Contents section of this document.

In the Chat Logging SPI sample, each chat transcript is logged in a separate text file under a folder defined in the chatlogging.ini file. If some screenshots are sent during the chat, each screenshot picture is saved into a separate file in the same folder. Display the ability to send a warning to the sender of the replaced message, both in ASCII and RichText. The file version provides examples of constants, error messages, and additional chatlogging.ini flags.

Note Constants, error messages, and chatlogging.ini flags in the Chat Logging sample are optional.

Chatlogging.ini Flags for the Chat Logging SPI Sample

The following chatlogging.ini flags have been specially created for the Chat Logging SPI sample. There is an option to set messages with rich text or plain text:

```
BB_CL_TRACE flag – used for recording the Chat Logging SPI sample debug messages
in trace files. To enable the recording, set BB_CL_TRACE to 1.

BB_CL_LIBRARY_PATH flag (in the [Library] section) – used to specify the
location for logging chat transcripts. In this sample, chat transcripts are
saved in the sametime\CLData folder.

CL_BAN_CHAT_BETWEEN_USERS – used to set two names of users, to imitate session
initialization forbidding.
Examples:
CL_BAN_CHAT_BETWEEN_USERS=CN=John Smith/O=Ubique,CN=Peter Pan/O=Ubique
```

For LDAP configuration: CN_BAN_CHAT_BETWEEN_USERS="uid=john Smith,ou=sametime,dc=ibm,dc=com","uid=Tom Peters,ou=sametime,dc=ibm,dc=com".

In case the names contain commas, you can bound the names by the quotes. Example: CL_BAN_CHAT_BETWEEN_USERS="CN=John Smith,O=Ubique","CN=Peter Pan,O=Ubique"

CL_BAN_CHAT_WARNING_MSG – used to supply a message, which will be sent to the client, when his session initialization attempt is forbidden. Example: CL_BAN_CHAT_WARNING_MSG=Chat initialization was banned by ChatLogging

CL_REPLACE_STRINGS_IN_MSG – used to set two strings, when the first string is a pattern to be searched and replaced by the second string. Example: CL_REPLACE_STRINGS_IN_MSG=June,July

Note: In case the strings contain commas, you can bound the strings by the quotes. Example: CL_REPLACE_STRINGS_IN_MSG="June, June","July,July"

CL_DELETE_MSG_WITH_STRING – used to set a sub string, that when found in a message, the message will be deleted. Example: CL_DELETE_MSG_WITH_STRING=August

CL_WARNING_ON_DELETED_MSG – used to set a warning message, which will be sent when the original message is deleted. Example: CL_WARNING_ON_DELETED_MSG=The last message was deleted by ChatLogging

CL_WARNING_ON_DELETED_MSG_RICH_TEXT – used to set a warning message in rich text format, which will be sent when the original message is deleted. Example: CL_WARNING_ON_DELETED_MSG_RICH_TEXT= The last message was deleted by ChatLogging

CL_CHAT_START_DISCLAIMER – used to set a warning message, which will be sent when chat starts. Example: CL_CHAT_START_DISCLAIMER= This chat would be saved

CL_CHAT_START_DISCLAIMER_RICH_TEXT – used to set a warning message in rich text format, which will be sent when chat starts. Example: CL_CHAT_START_DISCLAIMER_RICH_TEXT= This chat would be saved

CL_REPLACE_STRINGS_IN_MSG_WARNING – used to set a warning message, which will be sent when the original message is replaced. Example: CL_REPLACE_STRINGS_IN_MSG_WARNING=Warning, your message was replaced by the server

CL_REPLACE_STRINGS_IN_MSG_WARNING_RICH_TEXT - used to set a warning message in rich text format, which will be sent when the original message is replaced. Example: CL_REPLACE_STRINGS_IN_MSG_WARNING_RICH_TEXT=Warning, your message was replaced by the server.

Chat Logging SPI Sample Parameters

This section describes the following.

- Chat Logging SPI Sample Error Messages

The following error messages were developed specifically for the Chat Logging SPI sample. For general chat logging return and error messages, refer to the Chat Logging SPI Return Messages section of this document.

Table listing error messages for the Chat Logging SPI sample.

Message	Value	Description

ST_DDA_CL_SESSION_ALREADY_EXISTS	0x1002	The operation could not be performed because the specified session ID is already in use.
ST_DDA_CL_SESSION_DOES_NOT_EXIST	0x1003	The indicated session ID does not exist.
ST_DDA_CL_DB_ERROR	0x1004	Access to the data store was denied.
ST_DDA_SESSION_CREATION_DECLINED	0x1006	The chat session initialization is declined by the Chat Logging.
ST_DDA_SESSION_CLOSED_BY_SERVER	0x1007	The chat session is closed by the Chat Logging.
ST_DDA_MSG_CHANGED	0x1008	The original message was changed or replaced by the Chat Logging.

Chat Logging SPI Sample Constants

The following constants were developed specifically for the Chat Logging SPI sample. For information about general chat logging constants, refer to the Chat Logging SPI Reference section of this document.

Table listing constants for the Chat Logging SPI sample.

Message	Value	Description
ST_DDA_MAX_MSG_LEN	11000 characters	The maximum length of the message string that can be logged.
ST_DDA_MAX_NAME_LENGTH	256 characters	The maximum length of the login name, user name and group name.
ST_DDA_MAX_STR_LEN	1024 characters	The maximum length of the string.
MESSAGE_SIZE	2048	The maximum message size of a path or a file name.
CL_SUFFIX	.dat	The file type of the files where chat transcripts are saved.
LIBRARY_SECTION	Library	The section of a chatlogging.ini file where the DEFAULT_LIB_PATH flag is written.
DEFAULT_LIB_PATH	CLData/	The default location of the folder where the chat transcripts are saved.
CL_LIBRARY_PATH	BB_CL_LIBRARY_PATH	The name of the debug flag in the chatlogging.ini file. This flag determines the folder where chat transcripts are saved.
CL_INI_FILE_NAME	chatlogging.ini (example)	The name of the Chat Logging SPI configuration file.
CL_TRACE_FILE_NAME	trace/StChatLog.txt	The name and location of the trace files where the debug messages are recorded.
CL_PREFIX_NAME	StChatLog	The prefix for the name of the trace file.

Chat Logging SPI Reference

This section describes the Chat Logging SPI Parameters and the Chat Logging SPI Functions.

Chat Logging SPI Parameters

This section contains information about:

- Chat Logging SPI Return Messages
- Chat Logging SPI Constants

The table below lists and describes the return messages common to all Sametime Toolkit SPIs, APIs, and DLLs, including the Chat Logging SPI.

Table listing return messages common to all Sametime Toolkit SPIs, APIs, and DLLs.

Return message	Value	Description
ST_DDA_API_OK	0x0000	Returned by the SPI functions when the requested operation succeeds
ST_DDA_API_VERSION_MISMATCH	0x0001	Returned by the initialization function if version incompatibility is found
ST_DDA_API_INVALID_PARAM	0x0002	Returned by the SPI functions when an operation cannot be performed because one or more of the parameters does not match the specifications (for example, wrong length of user name, wrong format of token, and so on)
ST_DDA_API_INTERNAL_ERROR	0x0003	Returned by the SPI functions when an operation fails due to a problem other than those in this table
ST_DDA_API_NOT_SUPPORTED	0x0004	Returned by the initialization function if the chat logging library does not support the SPI version

Chat Logging SPI Constants

The table below lists and describes the constants defined in stDdaApiDefs.h that are common to all Sametime toolkit SPIs and APIs, including the Chat Logging SPI.

Table listing constants defined in stDdaApiDefs.h common to all Sametime toolkit SPIs and APIs.

Name	Value	Description
ST_DDA_MAX_NAME_LENGTH	256 characters	The maximum length of the login name, user name, and group name.
ST_DDA_MAX_ID_LENGTH	256 characters	The maximum length of a user ID or group ID.
ST_DDA_MAX_DESCRIPTION_LENGTH	256 characters	The maximum length of a user description or a group description string.

ST_DDA_API_NOTES, ST_DDA_API_LDAP, ST_DDA_API_DB2	0x0002, 0x0004, 0x0008, respectively	Indicate which platform interfaces were initialized. No other types of platform interfaces are specified. If the platform interface in use is not defined, use a higher bit (for example, 0x08 or 0x10).

Chat Logging SPI Functions

This section describes the Chat Logging SPI functions provided by this toolkit.

The Sametime Chat Logging SPI provides the following functions:

```
stDdaClInit
stDdaClTerminate
stDdaClTimerEvent
stDdaClSessionStarted
stDdaClSessionStartedByOrgName
stDdaClSessionEnded
stDdaClEntity
stClEntityType
stClSrvMsgDestination
stDdaClJoiningSession
stDdaClSessionPolicycheckFunc
stDdaClLeavingSession
stDdaClSessionMsg
stDdaClSessionDataMsg
stDdaClJoiningSessionWithSrvMsg
stDdaClLeavingSessionWithSrvMsg
stDdaClSessionMsgWithSrvMsg
stDdaClSessionDataMsgWithSrvMsg
stDdaClCleanSrvMsgs
```

stDdaClInit

Overview

The stDdaClInit function allows the Chat Logging SPI to perform preparations that are necessary to support its regular functionality. The function also checks for anything preventing the SPI from supplying this functionality.

The syntax for the initialization call is:

```
int ST_DDA_API
stDdaClInit (/*in/out*/int* prVersion,
```

```
/*in*/      int  initializedOutside,
/*out*/     int* initializedInside,
/*out*/     char* dirType,
/*out*/     char* libVersion,
/*out*/     StDdaRetVal* appRetCode,
/*out*/     char* appRetMsg ) ;
```

Description

The stDdaClInit function initializes the chat logging library and verifies that it can handle the specified version of the SPI. The calling application places the SPI version number in the prVersion parameter. The chat logging library adjusts its SPI version number to the version of calling application.

If the version numbers are equal, the initialization continues.

If the SPI version number of the chat logging library is higher than the SPI version number of the calling application, the Chat Logging SPI decides whether it can work with an SPI of a lower version. Depending on the particular configuration, the initialization either continues or an error message is returned.

If the SPI version number of the chat logging directory is lower than the SPI version number of the calling application, the calling application decides whether it can work with an SPI of a lower version. Depending on the particular configuration, the initialization either continues, or an error message is returned.

The initializeOutside parameter contains information about platform interfaces that are initialized by this calling application.

Note One server application can use several libraries, and some platform interfaces must be initialized only once per server application.

If the initializeOutside parameter indicates that the platform interface was already initialized, the initialization process does not continue and an OK message is returned.

If the platform interface was not initialized, the chat logging library performs the initialization and sets one of the bits in the initializedInside mask accordingly. The initializeInside parameter contains information about platform interfaces that must be initialized and it keeps an internal record of the initialized interfaces.

Parameters – Input

Table listing inputs for the stDdaClInit function.

perversion	Determines which SPI versions are used by the calling application (Chat Logging server application) and the chat logging library. This parameter is also an output parameter. Sametime 8.x supports prVersion =11, 12, 13 and 14 Sametime 9.0 supports prVersion =16 The library that declares its version as 13 and higher **must** implement stDdaClSessionDataMsg method See stDdaClSessionStartedByOrgName and stDdaClSessionDataMsg for specific information about the prVersion parameter pertaining to these functions.

initializedOutside	Describes the platform interfaces initialized by other libraries that this server application might have used before calling this function. Examples of this parameter are ST_DDA_API_NOTES for the Notes platform interface and ST_DDA_API_LDAP for the LDAP platform interface.

Parameters – Output

Table listing outputs for the stDdaClInit function.

initializedInside	This bit mask describes platform interfaces that were initialized by this library. It is used in conjunction with the InitializedOutside parameter; the function decides what interfaces still need to be initialized based on the value of the InitializedOutside parameter. The function then keeps an internal record of the interfaces that it initialized and sets the InitializedInside parameter accordingly.
dirType	This parameter identifies the library and is recorded in the sametime.log. Any change in the library type is updated in the sametime.log.
libVersion	This string identifies the library version. The libVersion parameter is recorded in the sametime.log. Any change in the library version is updated in the sametime.log
appRetCode	This pointer points to a variable that holds the return code from the most recent call to one of the library's functions.
appRetMsg	This pointer points to a string variable (null-terminated) that holds a message containing any information about what happened during the most recent call to one of the library's functions. **Note** The maximum length of this message is 1023 characters (buffer maximum length is 1024).

Return Values

Table listing return values for the stDdaClInit function.

ST_DDA_API_OK	0x0000	Indicates that the initialization succeeded
ST_DDA_AIP_VERSION_MISMATCH	0x0001	Indicates that the library does not support the given version of the calling application
ST_DDA_API_INTERNAL_ERROR	0x0003	Indicates that the initialization failed
ST_DDA_API_NOT_SUPPORTED	0x0004	Indicates that the library does not support this SPI version of the calling application

stDdaClTerminate

Overview

The stDdaClTerminate function allows the developer to perform an efficient shutdown by freeing memory and any other resources that were being used by the application.

The syntax for the termination call is:

```
void ST_DDA_API
stDdaClTerminate()
```

Description

This function terminates any platform interface initialized by the library. It uses a copy of the value of the initializedInside parameter that was saved when stDdaClInit was called.

Parameters – Input

None

Parameters – Output

None

Return Values

None

stDdaClTimerEvent

Overview

The stDdaClTimerEvent function allows the library to perform actions that are time-based, such as refreshing a cache or reconnecting to the database.

The syntax for the timer event call is:

```
void ST_DDA_API
stDdaClTimerEvent(void)
```

Description

This function is called at the defined interval (for example, once per minute) by the calling application, to enable time-based actions.

Parameters – Input

None

Parameters – Output

None

Return Values

None

stDdaClSessionStarted

Overview

The stDdaClSessionStarted function allows the developer to perform certain preparations for new session logging.

The syntax for this call is:

```
int ST_DDA_API
stDdaClSessionStarted (/*in*/ const char* sessionId)
```

Description

This function is called when a new chat session is started in an instant message or a Place (a meeting or n-way chat).

Parameters – Input

Table listing inputs for the stDdaClSessionStarted function.

sessionId	This parameter is an identifier of the chat session. It is used in subsequent calls relating to the same session. The format of this field may vary depending on the session's type, and usually will have the following prefixes to indicate the session's type: • IM_ - A two-way chat session. • ANNOUNCE_ - Represents an announcement or an admin message. • SVC_ - Any other service type between two users that does not have a session prefix. • FT_ - A file transfer session between two users, when file transfer is enabled via the Community Server and not peer to peer. • 0#PLC$ – A group chat, this value is determined by the Sametime client and could be different with older Sametime clients. Notes: • There might be other session formats, as some of them are created by other parties developing for Sametime using the ST Java toolkit. • The session ID will be the same across servers for two-way sessions which include IM_ and SVC. Announcements will also have the same session ID excluding admin messages.

Parameters – Output

None

Return Values

Table listing return values for the stDdaClSessionStarted function.

ST_DDA_API_OK	0x0000	Indicates that the action succeeded
ST_DDA_CL_SESSION_ALREADY_EXIST	0x1002	Indicates that the ID defined for a new session

		is already in use
ST_DDA_CL_DB_ERROR	0x1004	Indicates a failure to write to the data store

Note If the server is configured to work in strict mode and the Chat Logging SPI function returns errors, existing chat session will be destroyed.

stDdaClSessionStartedByOrgName

Overview

The stDdaClSessionStartedByOrgName function allows the developer to perform certain preparations for new session logging. This function is similar to stDdaClSessionStarted, and allows for chat logging in a hosted environment where multiple organizations use a single Sametime server.

The syntax for the session started call is as follows:

```
int ST_DDA_API
stDdaClSessionStartedByOrgName (/*in*/ const char* sessionId
                                /*in*/ const char* organization)
```

Description

This function is called when a new chat session is started in an instant message or a Place.

Parameters – Input

Table listing inputs for the stDdaClInit function.

sessionId

An identifier of the chat session. This ID is used in subsequent calls relating to the same session.

The format of this field may vary depending on the session's type, and usually will have the following prefixes to indicate the session's type:

- IM_ - A two-way chat session.
- ANNOUNCE_ - Represents an announcement or an admin message.
- SVC_ - Any other service type between two users that does not have a session prefix.
- FT_ - A file transfer session between two users, when file transfer is enabled via the Community Server and not peer to peer.
- 0#PLC$ – A group chat, this value is determined by the Sametime client and could be different with older Sametime clients.

Notes:

- There might be other session formats, as some of them are created by other parties developing for Sametime using the ST Java toolkit.
- The session ID will be the same across servers for two-way sessions

	which include IM_ and SVC. Announcements will also have the same session ID excluding admin messages.
organization	The organization's name.

Parameters – Output

None

Return Values

Table listing return values for the stDdaClInit function.

ST_DDA_API_OK	0x0000	Indicates that the action succeeded
ST_DDA_CL_SESSION_ALREADY_EXIST	0x1002	Indicates that the ID defined for a new session is already in use
ST_DDA_CL_DB_ERROR	0x1004	Indicates a failure to write to the data store

Note If the server is configured to work in strict mode and the Chat Logging SPI function returns errors, existing chat session will be destroyed.

stDdaClSessionEnded

Overview

The stDdaClSessionEnded function allows the developer to perform certain operations for ending the session logging.

The syntax for this call is:

```
int ST_DDA_API
stDdaClSessionEnded(/*in*/ const char* sessionId);
```

Description

This function is called when a session is finished because the IM channel is closed or a Place is closed.

Parameters – Input

Table listing inputs for the stDdaClSessionEnded function.

sessionId	An identifier of the chat session

Parameters – Output

None

Return Values

Table listing return values for the stDdaClSessionEnded function.

ST_DDA_API_OK	0x0000	Indicates that the action succeeded

ST_DDA_CL_SESSION_NOT_EXIST	0x1003	Indicates that the defined session ID does not exist

Common Structures

The following common structures are use by the SPI methods:

stDdaClEntity

StClEntityType

StClSrvMsgDestination

stDdaClEntity

The following structure describes a participant in a chat session:

```
struct StDdaClEntity {
 char id[ST_DDA_MAX_NAME_LENGTH];
 StClEntityType type;
};
```

StClEntityType

This enumerator defines the different types of participants:

```
typedef enum {
 ST_DDA_CL_NO_TYPE,
 ST_DDA_CL_USER,
 ST_DDA_CL_SECTION,
 ST_DDA_CL_ACTIVITY,
 ST_DDA_CL_SESSION,
 ST_DDA_CL_EXTERNAL_USER,
 ST_DDA_CL_OFFLINE_USER
} StClEntityType;
```

StClSrvMsgDestination

This enumerator defines different types of server message destination

typedef enum {

ST_DDA_CL_SENDER,

ST_DDA_CL_RECEIVER

} StClSrvMsgDestination;

StClSrvMsgType

This enumerator defines different types of server messages

typedef enum {

```
  ST_DDA_CL_TEXT_MSG,
  ST_DDA_CL_DATA_MSG,
  ST_DDA_CL_SRV_MSG
} StClSrvMsgType;
```

StDdaClSrvMsg

This structure defines the server message

```
struct StDdaClSrvMsg {
  StClSrvMsgType srvMsgType;
  StClSrvMsgDestination srvMsgDestination;
  unsigned long msgLen;
  char* msg;
};
```

stDdaClJoiningSession

Overview

The stDdaClJoiningSession function is used to identify the scope that each new participant is joining. The new participant can be a user, a section, or an activity. The participant can:

```
Send and receive messages.
Receive messages sent directly to the participant or to the scope in which the
participant is included.
```

The syntax for this call is:

```
  int ST_DDA_API stDdaClJoiningSession(/*in*/ const char* sessionId,
            /*in*/ const StDdaClEntity *entity,
            /*in*/ const StDdaClEntity *scope)
```

Description

This function is called when a participant creates or joins an IM or Place chat session.

Parameters – Input

Table listing inputs for the stDdaClJoiningSession function.

sessionId	An identifier of the chat session
entity	An identifier of the new participant, consisting of a type and an ID. A new participant can be a section, an activity, or a user.
scope	An identifier of the scope that the new participant joins. The scope can be a user, a section, a session, or an activity.

Parameters – Output

None

Return Values

Table listing return values for the stDdaClJoiningSession function.

ST_DDA_API_OK	0x0000	Indicates that the action succeeded.
ST_DDA_CL_DB_ERROR	0x1004	Indicates that the access to the data store was denied. Failure to write to the data store.
ST_DDA_CL_SESSION_NOT_EXIST	0x1003	Indicates that the session with a defined ID does not exist.

Note If the server is configured to work in strict mode and the Chat Logging SPI function returns errors, existing chat session will be destroyed.

stDdaClJoiningSessionWithSrvMsg

Overview

The stDdaClJoiningSessionWithSrvMsg function is used to identify the scope that each new participant is joining. The new participant can be a user, a section, or an activity. The participant can:

```
Send and receive messages.
```

```
Receive messages sent directly to the participant or to the scope in which the
participant is included.
```

The syntax for this call is:

```
int ST_DDA_API stDdaClJoiningSessionWithSrvMsg(
        /*in*/ const char* sessionId,
        /*in*/ const StDdaClEntity *entity,
        /*in*/ const StDdaClEntity *scope,
        /*out*/ StDdaClSrvMsg** srvMessages,
        /*out*/ unsigned long* srvMessagesLen)
```

Description

This function is called when a participant creates or joins an IM or Place chat session.

Parameters – Input

Table listing inputs for the stDdaClJoiningSessionWithSrvMsg function.

sessionId	An identifier of the chat session
Entity	An identifier of the new participant, consisting of a type and an ID. A new participant can be a section, an activity, or a user.

Scope	An identifier of the scope that the new participant joins. The scope can be a user, a section, a session, or an activity.

Parameters – Output

Table listing outputs for the stDdaClJoiningSessionWithSrvMsg function.

srvMessages	Server messages – server message that were generated by the Chat Logging library and will be sent to the session initiator when the session initiation is declined (`ST_DDA_SESSION_CREATION_DECLINED` return code is received). The message will be released by the Chat Logging server application.
srvMessagesLen	Server Messages array length

Return Values

Table listing return values for the stDdaClJoiningSessionWithSrvMsg function.

ST_DDA_API_OK	0x0000	Indicates that the action succeeded.
ST_DDA_CL_DB_ERROR	0x1004	Indicates that the access to the data store was denied. Failure to write to the data store.
ST_DDA_CL_SESSION_NOT_EXIST	0x1003	Indicates that the session with a defined ID does not exist.
ST_DDA_SESSION_CREATION_DECLINED	0x1006	Indicates that the session creation was declined by the server.
ST_DDA_MSG_CHANGED	0x1008	Indicates that there are server messages for the joining user. Could be used to send disclaimer message.

Note If the server is configured to work in strict mode and the Chat Logging SPI function returns errors, existing chat session will be destroyed.

stDdaClLeavingSession

Overview

The stDdaClLeavingSession function is used to identify the participant who is leaving a session. This participant can no longer send or receive messages.

The syntax for this call is:

```
int ST_DDA_API
stDdaClLeavingSession( /*in*/ const char* sessionId,
          /*in*/ const StDdaClEntity *entity)
```

Description

This function is called when a participant leaves a chat session.

Parameters – Input

Table listing inputs for the stDdaClLeavingSession function.

sessionId	An identifier of the chat session.
entity	An identifier of the leaving participant. The participant can be a section, an activity, or a user.

Parameters – Output

None

Return Values

Table listing return values for the stDdaClLeavingSession function.

ST_DDA_API_OK	0x0000	Indicates that the action succeeded.
ST_DDA_CL_DB_ERROR	0x1004	Indicates that access to the data store was denied. Failure to write to the data store.
ST_DDA_CL_SESSION_NOT_EXIST	0x1003	Indicates that the session with a defined ID does not exist.

stDdaClLeavingSessionWithSrvMsg

Overview

The stDdaClLeavingSessionWithSrvMsg function is used to identify the participant who is leaving a session. This participant can no longer send or receive messages.

The syntax for this call is:

```
int ST_DDA_API
stDdaClLeavingSessionWithSrvMsg(/*in*/ const char* sessionId,
                /*in*/ const StDdaClEntity *entity,
                /*out*/ StDdaClSrvMsg** srvMessages,
                 /*out*/ unsigned long* srvMessagesLen)
```

Description

This function is called when a participant leaves a chat session.

Parameters – Input

Table listing inputs for the stDdaClLeavingSessionWithSrvMsg function.

sessionId	An identifier of the chat session.
entity	An identifier of the leaving participant. The participant can be a section, an activity, or a user.

Parameters – Output

Table listing outputs for the stDdaClLeavingSessionWithSrvMsg function.

srvMessages	Server messages – server message that were generated by the Chat Logging library and will be sent to both parties of chat session, when the session is left. The array will be released in the Chat Logging Server application and should be allocated as one memory block.
srvMessagesLen	Server Messages array length

Return Values

Table listing return values for the stDdaClLeavingSessionWithSrvMsg function.

ST_DDA_API_OK	0x0000	Indicates that the action succeeded.
ST_DDA_CL_DB_ERROR	0x1004	Indicates that access to the data store was denied. Failure to write to the data store.
ST_DDA_CL_SESSION_NOT_EXIST	0x1003	Indicates that the session with a defined ID does not exist.

stDdaClSessionMsg

Overview

The stDdaClSessionMsg function is used to identify the source destination and content of the message.

The syntax for this call is:

```
int ST_DDA_API
stDdaClSessionMsg(/*in*/ const char* sessionId,
        /*in*/ const StDdaClEntity *sender,
        /*in*/ unsigned long msgLen,
        /*in*/ const char *msg,
        /*in*/ const StDdaClEntity *receiver);
```

Description

This function is called when a participant receives a message.

Parameters – Input

Table listing inputs for the stDdaClSessionMsg function.

sessionId	An identifier of the chat session.
sender	An identifier of the sender of the message. The sender can be a user, a section, a session, or an activity.
msgLen	The length of the message being read from the message buffer.
msg	The body of the message.
receiver	An identifier of the receiver of the message. The receiver can be a user, a section, a session, or an activity.

Parameters – Output

None

Return Values

Table listing return values for the stDdaClSessionMsg function.

ST_DDA_API_OK	0x0000	Indicates that the action succeeded.
ST_DDA_CL_DB_ERROR	0x1004	Indicates that the access to the data store was denied. Failure to write to the data store.
ST_DDA_MSG_TOO_LONG	0x1005	Indicates that the length of the message exceeds the defined length.
ST_DDA_CL_SESSION_NOT_EXIST	0x1003	Indicates that the session with a defined ID does not exist.

Note If the server is configured to work in strict mode and the Chat Logging SPI function returns errors, existing chat session will be destroyed.

stDdaClSessionMsgWithSrvMsg

Overview

The stDdaClSessionMsgWithSrvMsg function is used to identify the source destination and content of the message.

The syntax for this call is:

```
int ST_DDA_API
stDdaClSessionMsgWithSrvMsg(/*in*/ const char* sessionId,
              /*in*/ const StDdaClEntity *sender,
              /*in*/ unsigned long msgLen,
              /*in*/ const char *msg,
              /*in*/ const StDdaClEntity *receiver,
              /*out*/ StDdaClSrvMsg** srvMessages,
               /*out*/ unsigned long* srvMessagesLen)
```

Description

This function is called when a participant receives a message.

Parameters – Input

Table listing inputs for the stDdaClSessionMsgWithSrvMsg function.

sessionId	An identifier of the chat session.
sender	An identifier of the sender of the message. The sender can be a user, a section, a session, or an activity.
msgLen	The length of the message being read from the message buffer.

msg	The body of the message.
receiver	An identifier of the receiver of the message. The receiver can be a user, a section, a session, or an activity.

Parameters – Output

Table listing outputs for the stDdaClSessionMsgWithSrvMsg function.

srvMessages	Server messages – server message that were generated by the Chat Logging library and will be sent to both parties of chat session, when the session is left. The array will be released in the Chat Logging Server application and should be allocated as one memory block.
srvMessagesLen	Server Messages array length

Return Values

Table listing return values for the stDdaClSessionMsgWithSrvMsg function.

ST_DDA_API_OK	0x0000	Indicates that the action succeeded.
ST_DDA_CL_DB_ERROR	0x1004	Indicates that the access to the data store was denied. Failure to write to the data store.
ST_DDA_MSG_TOO_LONG	0x1005	Indicates that the length of the message exceeds the defined length.
ST_DDA_CL_SESSION_NOT_EXIST	0x1003	Indicates that the session with a defined ID does not exist.
ST_DDA_SESSION_CLOSED_BY_SERVER	0x1007	Indicates that the session was closed by the server.
ST_DDA_MSG_CHANGED	0x1008	Indicates that the original message was changed and the server messages should be sent instead.

Note If the server is configured to work in strict mode and the Chat Logging SPI function returns errors, existing chat session will be destroyed.

stDdaClSessionDataMsg

Overview

The stDdaClSessionDataMsg function is used to identify the source destination and content of the data message.

The syntax for this call is:

```
int ST_DDA_API
stDdaClSessionDataMsg(/*in*/ const char* sessionId,
        /*in*/ const StDdaClEntity *sender,
        /*in*/ unsigned long dataType,
        /*in*/ unsigned long dataSubType,
        /*in*/ unsigned long msgLen,
```

```
/*in*/ const char *msg,
/*in*/ const StDdaClEntity *receiver);
```

Description

This function is called when a participant receives a data message.

Parameters – Input

Table listing inputs for the stDdaClSessionDataMsg function.

sessionId	An identifier of the chat session.
Sender	An identifier of the sender of the message. The sender can be a user, a section, a session, or an activity.
dataType	The data type of the message.
dataSubType	The data subtype of the message.
msgLen	The length of the message being read from the message buffer.
msg	The body of the message.
receiver	An identifier of the receiver of the message. The receiver can be a user, a section, a session, or an activity.

Parameters – Output

None

Return Values

Table listing return values for the stDdaClSessionDataMsg function

ST_DDA_API_OK	0x0000	Indicates that the action succeeded.
ST_DDA_CL_DB_ERROR	0x1004	Indicates that the access to the data store was denied. Failure to write to the data store.
ST_DDA_MSG_TOO_LONG	0x1005	Indicates that the length of the message exceeds the defined length.
ST_DDA_CL_SESSION_NOT_EXIST	0x1003	Indicates that the session with a defined ID does not exist.

Note If the server is configured to work in strict mode and the Chat Logging SPI function returns errors, existing chat session will be destroyed.

stDdaClSessionDataMsgWithSrvMsg

Overview

The stDdaClSessionDataMsgWithSrvMsg function is used to identify the source destination and content of the data message.

The syntax for this call is:

```
int ST_DDA_API
stDdaClSessionDataMsgWithSrvMsg (/*in*/ const char* sessionId,
        /*in*/ const StDdaClEntity *sender,
        /*in*/ unsigned long dataType,
        /*in*/ unsigned long dataSubType,
        /*in*/ unsigned long msgLen,
        /*in*/ const char *msg,
        /*in*/ const StDdaClEntity *receiver);
```

Description

This function is called when a participant receives a data message.

Parameters – Input

Table listing inputs for the stDdaClSessionDataMsgWithSrvMsg function.

sessionId	An identifier of the chat session.
Sender	An identifier of the sender of the message. The sender can be a user, a section, a session, or an activity.
dataType	The data type of the message.
dataSubType	The data subtype of the message.
msgLen	The length of the message being read from the message buffer.
msg	The body of the message.
receiver	An identifier of the receiver of the message. The receiver can be a user, a section, a session, or an activity.

Parameters – Output

Table listing outputs for the stDdaClSessionDataMsgWithSrvMsg function.

srvMessages	Server messages – server message that were generated by the Chat Logging library and will be sent to both parties of chat session, when the session is left. The array will be released in the Chat Logging Server application and should be allocated as one memory block.
srvMessagesLen	Server Messages array length

Return Values

Table listing return values for the stDdaClSessionDataMsgWithSrvMsg function.

ST_DDA_API_OK	0x0000	Indicates that the action succeeded.
ST_DDA_CL_DB_ERROR	0x1004	Indicates that the access to the data store was denied. Failure to write to the data store.
ST_DDA_MSG_TOO_LONG	0x1005	Indicates that the length of the message exceeds the defined length.
ST_DDA_CL_SESSION_NOT_EXIST	0x1003	Indicates that the session with a defined ID does

		not exist.
ST_DDA_SESSION_CLOSED_BY_SERVER	0x1007	Indicates that the session was closed by the server.
ST_DDA_MSG_CHANGED	0x1008	Indicates that the original message was changed and the server messages should be sent instead.

Note: If the server is configured to work in strict mode and the Chat Logging SPI function returns errors, existing chat session will be destroyed.

stDdaClCleanSrvMsgs

Overview

The stDdaClCleanSrvMsgs function is used to delete the server messages and free the allocated memory.

The syntax for this call is:

```
stDdaClCleanSrvMsgs(StDdaClSrvMsg* srvMessages,
       unsigned long srvMessagesLen);
```

Description

This function is called when there were previously allocated server messages.

Parameters – Input

Table listing inputs for the stDdaClCleanSrvMsgs function.

srvMessages	Pointer to array of server messages. All the server messages must be allocated in a single call to allocating methods.
srvMessagesLen	Length of server messages array.

Return Values

Table listing return values for the stDdaClCleanSrvMsgs function.

ST_DDA_API_OK	0x0000	Indicates that the action succeeded.
ST_DDA_ERROR	0x0003	Indicates that the action failed.

stDdaClSessionPolicycheckFunc

Overview

This function is used to enforce banning in 1 on 1 chat.

The syntax for this call:

```
int ST_DDA_API
stDdaClSessionPolicycheckFunc(/*in*/ const StDdaClEntity *entity,
```

```
/*in*/ const StDdaClEntity *toEntity,
/*out*/ StDdaClSrvMsg** srvMessages,
/*out*/ unsigned long* srvMessagesLen);
```

Description

This function checks the chat between entity and toEntity is allowed.

Parameters – Input

Table listing inputs for the stDdaClSessionPolicycheckFunc function.

entity	The first entity in the policy check.
toEntity	The second entity in the policy check.

Parameters – Output

Table listing outputs for the stDdaClSessionPolicycheckFunc function.

srvMessages	Server messages – server message that were generated by the Chat Logging library and will be sent to both parties of chat session, when the session is left. The array will be released in the Chat Logging Server application and should be allocated as one memory block.
srvMessagesLen	Server Messages array length.

Return Values

Table listing return values for the stDdaClSessionPolicycheckFunc function.

ST_DDA_API_OK	0x0000	Indicates that chat between enity and toEntity is allowed.
ST_DDA_SESSION_CREATION_DECLINED	0x1006	Indicates that enity and toEntity are not allowed to participate in the chat session.
ST_DDA_API_INTERNAL_ERROR	0x0003	Indicates that the action failed.

Chapter 3. Token Authentication SPI

Sametime uses authentication by token to authenticate connections that occur after a user has authenticated once using basic password authentication. Authentication by token prevents a user from having to re-enter authentication credentials.

Sametime supports two types of authentication tokens out-of-the-box:

Proprietary Sametime token

LTPA token supported by Domino and WebSphere

The Sametime Token Authentication Server Provider Interface (SPI) is a server-side Sametime feature that allows you to customize Sametime for a different kind of token generated in your deployment by developing a DLL or service program that implements the specified SPI.

Getting Started

The Token Authentication SPI is part of the Directory and Database Access Toolkit. You can extract the files for this toolkit either on your local machine or (to make it available to other users) on your Sametime server.

To access the toolkit pages that include the toolkit documentation, samples, and binaries, open readme.html in the toolkit root folder

Token Authentication SPI Contents

The contents of the Directory and Databases Access Toolkit that concern the Token Authentication SPI are:

- Sametime Directory and Database Access SPI documentation – This document
- Token Authentication SPI template – This template or "dummy" version of the Token Authentication SPI does not verify a token or generate a token, and it always returns OK. Use this template version for building a new Token Authentication SPI implementation.
- Token Authentication SPI header files – These files contain SPI definitions, syntax, variables, constants, return values, and so on.

Building the Token Authentication DLL or Service Program

This toolkit does not provide a full version of the Token Authentication DLL or service program. Use the Token Authentication SPI template to complete development of the Token Authentication SPI for your particular environment(s).

When creating the Token Authentication DLL or service program:

- Follow the interface defined by the Token Authentication SPI.
- Develop a mechanism for token generation and verification.

To ensure proper operation of the Token Authentication DLL or service program, see the following topics:

- General Considerations
- Building and Installing a Token Authentication DLL for Windows
- Building and Installing a Token Authentication Service Program for IBM i
- Recommendations

General Considerations

When deploying a Token Authentication DLL or service program in a distributed community, each server must be able to verify a token generated by others.

Building and Installing a Token Authentication DLL for Windows

Follow these general steps for building and installing a Token Authentication DLL:

1. Use the template version in the toolkit templates\authtoken directory to create a new Token Authentication SPI implementation.
2. When creating a new DLL, use the debug mechanisms and make sure debug messages are being saved in the trace files to facilitate troubleshooting during the development and use of the DLL.
3. Save the new DLL under the name StAuthToken.dll.
4. Set the ST_AUTH_TOKEN setting under the [ST_BB_NAMES] section of sametime.ini to N/A
5. Test the created DLL.

   Note Stop the Sametime server on which the DLL is installed. If the Sametime community contains more than one Sametime server, stop all servers in the community.
6. Replace the existing Token Authentication DLL on the Sametime server with the newly created DLL (StAuthToken.dll). The new DLL must be placed in the default directory that contains all other DLLs. The default directory is C:\Sametime. If more than one server is included in the community, replace the DLL for each server.
7. Start the Sametime server on which the DLL is installed. If the Sametime community contains more than one Sametime server, start all servers in the community.

Building and Installing a Token Authentication Service Program for IBM i

Follow these general steps for building and installing a Token Authentication service program:

To create a new custom Token Authentication SPI implementation, use the template version in the toolkit templates/authtoken directory.

When creating a new service program, use the debug mechanisms and make sure debug messages are being saved in the trace files to facilitate troubleshooting during the development and use of the service program.

To build and install the Token Authentication SPI sample, do the following:

1. If you have not already done so:
   - Copy the following toolkit files from your local machine to a directory (for example, "/STToolkit") on your IBM i:

     ```
     samples\chatlogging\ChatResource.cpp
     samples\chatlogging\ChatResource.h
     samples\chatlogging\ChatSessionTable.cpp
     samples\chatlogging\ChatSessionTable.h
     samples\chatlogging\stDdaClApi.cpp
     samples\nonwin32\utilities.cpp
     samples\nonwin32\utilities.h
     samples\nonwin32\debug.h
     samples\nonwin32\winprofile.cpp
     inc\chatlogging\stDdaClApi.h
     inc\chatlogging\stDdaClCodes.h
     templates\authtoken\stAuthTokenApi.cpp
     inc\authtoken\stAuthTokenApi.h
     inc\common\stDdaApiDefs.h
     inc\common\nonwin32\windows.h
     ```

   - Add the ASCII C/C++ Runtime Development kit to your system:

     http://www-03.ibm.com/servers/enable/site/asciirt/

2. Create a library to hold your custom service programs and modules

   ```
   CRTLIB ATHTKNLIB
   ```

3. Compile the authentication source files using the following commands:

   ```
   CRTCPPMOD MODULE(ATHTKNLIB/STAUTHTAPI)
   SRCSTMF('/STToolkit/stAuthTokenApi.cpp') DBGVIEW(*SOURCE)
   DEFINE('QSRCSTMF' '_UNIX' 'OS400' 'V5R2_COMPILER'
   'UNIX_TOOLKIT_COMPILE') TERASPACE(*YES) STGMDL(*INHERIT)
   INCDIR('/STToolkit' '/qibm/proddata/qadrt/include') TGTCCSID(819)

   CRTCPPMOD MODULE(ATHTKNLIB/UTILITIES)
   SRCSTMF('/STToolkit/utilities.cpp') DBGVIEW(*SOURCE)
   DEFINE('QSRCSTMF' '_UNIX' 'OS400' 'V5R2_COMPILER'
   'UNIX_TOOLKIT_COMPILE') TERASPACE(*YES) STGMDL(*INHERIT)
   INCDIR('/STToolkit' '/qibm/proddata/qadrt/include') TGTCCSID(819)
   ```

```
CRTCPPMOD MODULE(ATHTKNLIB/WINPROFILE)
SRCSTMF('/STToolkit/winprofile.cpp') DBGVIEW(*SOURCE)
DEFINE('QSRCSTMF' '_UNIX' 'OS400' 'V5R2_COMPILER'
'UNIX_TOOLKIT_COMPILE') TERASPACE(*YES) STGMDL(*INHERIT)
INCDIR('/STToolkit' '/qibm/proddata/qadrt/include') TGTCCSID(819)
```

4. Create a service program called STAUTHTKN in library ATHTKNLIB:

```
CRTSRVPGM SRVPGM(ATHTKNLIB/STAUTHTKN) MODULE(ATHTKNLIB/STAUTHTAPI
ATHTKNLIB/UTILITIES ATHTKNLIB/WINPROFILE) BNDSRVPGM(QADRTTS)
EXPORT(*ALL) OPTION(*DUPVAR *DUPPROC)
```

5. Change the object owner of the new STAUTHTKN service program to QNOTES:

```
CHGOBJOWN OBJ(ATHTKNLIB/STAUTHTKN) OBJTYPE(*SRVPGM) NEWOWN(QNOTES)
```

To activate this service program on all Sametime servers on your IBM i system:

1. Remove the symbolic link /QIBM/Userdata/lotus/notes/STAUTHTKN.SRVPGM via the CL command:

```
RMVLNK OBJLNK('/qibm/userdata/lotus/notes/STAUTHTKN.SRVPGM').
```

2. Create a new link /QIBM/Userdata/lotus/notes/STAUTHTKN.SRVPGM that points to your new STAUTHTKN service program in ATHTKNLIB via the CL command:

```
ADDLNK OBJ('/qsys.lib/ATHTKNLIB.lib/STAUTHTKN.SRVPGM')
NEWLNK('/qibm/userdata/lotus/notes/STAUTHTKN.SRVPGM')
```

3. Change the authority on the new link via the CL command:

```
CHGAUT OBJ('/QIBM/Userdata/lotus/notes/STAUTHTKN.SRVPGM')
USER(QNOTES) DTAAUT(*RWX) OBJAUT(*ALL)
```

To activate this service program on a single Sametime server on your IBM i system:

1. Create a new link /yourstserverdatadir/STAUTHTKN.SRVPGM that points to your new STAUTHTKN service program in ATHTKNLIB via the CL command (where yourstserverdatadir is the data directory of the Sametime server on which to deploy the service program):

```
ADDLNK OBJ('/qsys.lib/ATHTKNLIB.lib/STAUTHTKN.SRVPGM')
NEWLNK('/yourstserverdatadir/STAUTHTKN.SRVPGM')
```

2. Change the authority on the new link via the CL command

```
CHGAUT OBJ('/yourstserverdatadir/STAUTHTKN.SRVPGM') USER(QNOTES)
DTAAUT(*RWX) OBJAUT(*ALL)
```

To activate the created Token Authentication service program, stop and start the Sametime server(s) on which the service program is installed.

Recommendations

When building a Token Authentication DLL or service program, follow the provided template as closely as possible.

Token Authentication SPI Reference

This section describes the Token Authentication SPI parameters and the Token Authentication SPI functions.

Token Authentication SPI Parameters

This section contains information about:

- Token Authentication SPI Return Messages
- Token Authentication SPI Constants

Token Authentication SPI Return Messages

These return messages are common to all Sametime Toolkit SPIs, APIs, and DLLs, including the Token Authentication SPI.

Table listing Authentication SPI return messages.

Return message	Value	Description
ST_DDA_API_OK	0x0000	Returned by the SPI functions when the requested operation succeeds
ST_DDA_API_VERSION_MISMATCH	0x0001	Returned by the initialization function if version incompatibility is found
ST_DDA_API_INVALID_PARAM	0x0002	Returned by the SPI functions when an operation cannot be performed because one or more of the parameters does not match the specifications (for example, wrong length of user name, wrong format of token, and so on)
ST_DDA_API_INTERNAL_ERROR	0x0003	Returned by the SPI functions when an operation fails due to a problem other than those in this table
ST_DDA_API_NOT_SUPPORTED	0x0004	Returned by the initialization function if the token authentication library does not support the SPI version
ST_DDA_AUTH_BAD_LOGIN	0x1002	Returned by the SPI functions when the verification failed and, as a result, the user login will fail.
ST_DDA_API_SSO_ERROR	0x0006	Returned by the SPI functions when the verification failed due to SSO error

Token Authentication SPI Constants

These constants, defined in stDdaApiDefs.h, are common to all Sametime toolkit SPIs, APIs, and DLLs, including the Token Authentication SPI.

Table listing Authentication SPI constants.

Name	Value	Description
ST_DDA_MAX_NAME_LENGTH	256 characters	This constant defines the maximum length of the login name, user name, and group name.
ST_DDA_MAX_ID_LENGTH	256 characters	This constant defines the maximum length of a user ID or group ID.
ST_DDA_MAX_DESCRIPTION_LENGTH	256 characters	This constant defines the maximum length of a user description or a group description

		string.
ST_DDA_API_NOTES, ST_DDA_API_LDAP	`0x0002`, `0x0004` respectively	These constants are used to indicate which platform interfaces were initialized. No other types of platform interfaces are specified. If the platform interface in use is not defined, use a higher bit, such as 0x08 or 0x10.
ST_DDA_MAX_TOKEN_LENGTH	16384	Maximum allowed token length
ST_DDA_MAX_NUM_OF_TOKENS	3	The maximum number of the tokens that can be passed for authentication or to be generated

Token Authentication SPI Functions

This section describes the Token Authentication SPI functions provided by this toolkit.

The Sametime Token Authentication SPI provides the following functions:

`stTokenInit`

`stTokenTerminate`

`stGetToken`

`stVerifyToken`

`stGetTokens`

`stVerifyTokenAndExtractUserId`

stTokenInit

Overview

The stTokenInit function initializes the Token Authentication SPI module.

The syntax for this call is:

```
int ST_DDA_API
stTokenInit(/*in*/ int initializedOutside,
     /*out*/ int* initializedInside )
```

Description

The stTokenInit function initializes the Token Authentication SPI module.

Parameters – Input

Table listing inputs for stTokenInit function.

initializedOutside A bit mask describing the platform interfaces initialized by other directory

	access libraries before the call to this function was made. The values ST_DDA_API_SYBASE, ST_DDA_API_NOTES, and ST_DDA_API_LDAP are predefined. New implementations of the directory access libraries can use other value definitions for other directory/database platforms.

Parameters – Output

Table listing outputs for stTokenInit function.

initializedInside	A bit mask describing the platform interfaces initialized by this function. It is used by the function in conjunction with the initializedOutside parameter; the function decides what interfaces still need to be initialized based on the value of initializedOutside. It keeps an internal record of the initialized interfaces and sets the initializedInside parameter accordingly.

Return Values

Table listing return values for stTokenInit function.

ST_DDA_API_OK	0x0000	Initialization succeeded
ST_DDA_API_INTERNAL_ERROR	0x0003	Initialization failed

stTokenTerminate

Overview

The stTokenTerminate function is called when the process is terminated. It can do the necessary cleanup before the process termination. For example, in the Lotus Notes environment this function can terminate the Notes API.

The syntax for this call is:

```
void ST_DDA_API
stTokenTerminate()
```

Description

This function terminates any platform interface initialized by the library. It uses the copy of the value of initializedInside that it saved when stTokenInit was called.

Parameters – Input

None

Parameters – Output

None

Return Values

None

stVerifyToken

Overview

The stVerifyToken function attempts to verify the authentication token.

The syntax for this call is:

```
int ST_DDA_API
stVerifyToken(/*in*/ const char* loginName,
/*in*/ const char* token,
/*in*/ int tokenLength )
```

Description

This function is called when a user authenticates by token. The stVerifyToken function verifies that the token provided as input is valid, matches the provided login name, and has not expired. If the token is valid, the function returns 0.

Parameters – Input

Table listing inputs for stVerifyToken function.

loginName	String containing user's login name
token	Pointer to a buffer with the token
tokenLength	Denotes the length of the token to be read from the token buffer

stVerifyTokenAndExtractUserId

Overview

The stVerifyTokenAndExtractUserId function replaces the deprecated stVerifyToken method. It receives a list of authentication tokens and attempts to validate them.

The syntax for this call is:

```
int ST_DDA_API
stVerifyTokenAndExtractUserId(/*in*/ const TokenData tokenArr[],
                              /*in*/ const unsigned int &numOfTokens,
                              /*in/out*/ char* userId)
```

Description

This function is called when a user authenticates by one or more tokens. This function verifies that at least one of the tokens provided as input is valid, and has not expired. In this case it returns 0. In addition it extracts the user ID from the valid token so it can be used later in the login process. In case that the token type doesn't allow extraction of user ID the value of the userId argument isn't updated and returned as is.

Parameters – Input

Table listing inputs for stVerifyTokenAndExtractUserId function.

tokenArr	Array of TokenData objects. Each object contains a token of a user.
	TokenData object describes a single token. It contains a token string and type of token. TypeOfToken type definition becomes the following values:
	typedef unsigned short TypeOfToken;
	const TypeOfToken ST_SECTOKENFORMAT_UNKNOWN = 0x0000;
	const TypeOfToken ST_SECTOKENFORMAT_LTPATOKEN = 0x0001;
	const TypeOfToken ST_SECTOKENFORMAT_LTPATOKEN2 = 0x0002;

numOfTokens	Number of tokens in the passed array
userId	String to contain the extracted from the token user ID. The parameter can't be null.

Parameters – Output

userId	String to contain user ID is extracted from the token

Return Values

Table listing return values for stVerifyToken function.

ST_DDA_API_OK	0x0000	Indicates that the action succeeded.
ST_DDA_AUTH_BAD_LOGIN	0x1002	Indicates that the verification failed and, as a result, the user login will fail.
ST_DDA_API_INTERNAL_ERROR	0x0003	The function tried to verify token but failed.
ST_DDA_API_INVALID_PARAM	0x0002	A passed parameter is invalid
ST_DDA_API_SSO_ERROR	0x0006	SSO error occurred

stGetToken

Overview

The stGetToken function retrieves the authentication token.

The syntax for this call is:

```
int ST_DDA_API
stGetToken(/*in*/ const char* userId,
```

```
            /*out*/ char* token,
            /*out*/ int* tokenLength)
```

Description

The stGetToken is depreciated function. It generates a token for the provided user ID. With this token the client can log in at a later time without requiring a password. For example, a client application can send a request to get a token when it re-logs in or launches another client application that must log in to the Sametime server. With the token, the client application can log in without challenging the user for a password.

The generated token is valid for only a limited period of time.

Parameters – Input

Table listing inputs for stGetToken function.

UserID	The user's unique ID in the community,such as the user's particular name in Notes or LDAP

Parameters – Output

Table listing outputs for stGetToken function.

token	Pointer to a buffer containing the token generated by the function
tokenLength	Pointer to buffer containing the token length

Return Values

Table listing return values for stGetToken function.

ST_DDA_API_OK	0x0000	Indicates that the action succeeded.
ST_DDA_API_INTERNAL_ERROR	0x0003	The stGetToken function tried to get a token but failed.

stGetTokens

Overview

The stGetTokens function retrieves one or more authentication tokens according to the server configuration.

The syntax for this call is:

```
int ST_DDA_API
stGetTokens(/*in*/ const char* userId,
            /*out*/ TokenData tokenArr[],
            /*in*/ const unsigned int &tokenArrSize,
            /*out*/ unsigned int &numOfTokens)
```

Description

The stGetTokens function generates one or more tokens for the provided user ID based on the server configuration. With these tokens the client can log to this or other servers in at a later time without requiring a password. The token will be valid for only a limited period of time.

Parameters – Input

Table listing inputs for stGetTokens function.

UserID	The user's unique ID in the community, such as the user's particular name in Notes or LDAP
tokenArrSize	The size of the tokenArr array passed to the function

Parameters – Output

Table listing outputs for stGetTokens function.

tokenArr	Array of generated tokens for the user. The array is initializes from the application level and should be filled with one or more tokens by the function.
numOfTokens	Amount of generated tokens that was put into tokenArr

Return Values

Table listing return values for stGetTokens function.

ST_DDA_API_OK	0x0000	Indicates that the action succeeded.
ST_DDA_API_INTERNAL_ERROR	0x0003	The function tried to get a token but failed.

Chapter 4. File Transfer SPI

The Sametime File Transfer service intercepts all file transfer transactions at the server and accumulates the transferred files for virus scanning. The Server Provider Interface (SPI) is provided for you, the developer. It allows you to create your own DLL to implement the virus scanning itself. The DLL you create implements a predefined SPI used by the server for the virus scanning.

The File Transfer SPI sample is an example of a dummy implementation, in the sense of receiving the file and sending errors by a predefined parameter. You can use the sample as a template for building your customized file transfer DLL, however it does not implement the core of the scanning, (it does not implement any virus scanning).

Architecture

When an entity (user, server, or server application) needs to communicate with another entity, it creates a channel to the other entity. The channel can be encrypted if necessary.

The option of file transfer is enabled only between two users, once both of them are displayed as active users. Furthermore, the option of file transfer enabling is affected by the server configuration you may configure whether file transfer is allowed or not, allowed file size to transfer.

Instant Messaging

Instant messaging allows two people to exchange messages that they type via their computer keyboards. Instant messages are exchanged between two people only. When User A wants to have a chat with User B, User A selects User B, which causes the server to open a channel to User B. If approved, the server

connects the two users and thereafter acts only as a router, transferring the messages from one user to the other. In Sametime 3.0 and above, the messages are encrypted by default.

File Transfer Management

File transfer management requires that you understand how the file transfer works. The topics in this section can help you to study under the hoods of the File Transfer:

- Modes
- Multi-thread running
- Synchronous implementation
- Configuration and policy
- Error handling
- File transfer effect on Sametime server performance

Modes

The three modes of file transfer virus scanning:

- Always mode – a file is transferred successfully only if it was scanned for viruses and found clean. If the file transfer service is not present or if the virus-scanning DLL that implements the File Transfer SPI functions is not present, no file transfers are allowed. This is also known as Strict mode.
- WhenAvailable mode – a file is transferred successfully even when a virus-scanning DLL that implements the File Transfer SPI functions is not present. However, if the virus-scanning DLL does exist, the file is not transferred when a virus is found. This is also known as Relax mode.
- Never mode – the File Transfer DLL is not loaded at all. This is also known as Off mode.

In all three modes, the file transfer service enforces a policy checking for each file transfer.

Multi-Thread Running

The file transfer application is multi-threaded. It can call the scanFile method of the virus-scanning DLL multiple times, in parallel, from multiple threads.

Synchronous Implementation

All the methods of the virus-scanning DLL that implements the File Transfer SPI functions are synchronous. The file transfer application waits until the method returns.

Configuration and Policy

File Transfer Configuration

Use the Sametime StConfig.nsf file to configure the file transfer. To enable a file transfer:

1. Create or open the CommunityClient document.
2. Set "Allow file transfer" to true.
3. Define the maximum allowed file size:

1. Create or open the CommunityClient document.
2. Set "Maximum Transfer File Size" to the appropriate size in KB.

4. Choose the virus-scanning mode:
    1. Create or open the CommunityClient document.
    2. Enable the file transfer by choosing one of the following values for the File Transfer Flag field:
        - Never mode – File Transfer is allowed and the policy is verified. However, once the policy is verified, software no longer snatch the channel and the DLL does not load at all.
        - whenAvailable mode – Virus scanning occurs; if virus scanning does not work for any reason, the file is still transferred, however once virus is found we do not transfer the file as in Always mode.
        - Always mode – Virus scanning occurs; if it does not work for any reason, the file transfer is disabled.
5. To prevent file transfer when the file transfer service is not available, set the value of the Capture Service Type field to 0x00000038. If this field already holds a value, add 0x00000038 to field, separated by a comma.
6. Restart the http server and then the configuration service so the changes can take effect.
7. In a distributed community, the administrator should configure all the servers at once, and configure them all the same way:

    Either all servers have file transfer turned on, or all do not.

    All servers should be running in the same mode: either whenAvailable mode or Always mode. (See the ___ section.) As well as having the same configuration settings, such as allowing file transfer, maximum size allowed.

    Attention: If not all the servers are running in the same mode, some files are transferred and others are not. In this situation, the transfers occur arbitrarily.

    An additional method of configuring the file transfer is via the admin User interface:

    1. Enter the http://server_name\stcenter.nsf.
    2. Click the Administer the server button.
    3. Log in by typing the administrator user and password.
    4. Click **Configuration** and select Community Services.
    5. Set the parameters using similar attributes to the ones discussed above:
        - A checkbox to allow file transfer
        - A text area for defining the maximum allowed file size
        - A radio button for selecting the virus scan mode (Always = Strict, WhenAvailable = Relax, Never = Off)
    1. Restart the http server and then the configuration service so your changes can take effect.

Note There is no way to configure the "Capture Service Type field" other than through StConfig.nsf.

File Transfer Policy

Each user has a default or pre-defined policy. This policy includes:

- An allow file transfer flag
- The maximum file size
- A flag, indicating whether to use an exclude extension list for this user
- A list, indicating if this user is prevented from sending extensions

Policy is received via the policy server application. If no policy is running and Strict mode is defined, the application does not allow file transfers as the policy cannot be enforced.

Note The policy is enforced on the sender side only. In a distributed environment, the virus scanning and policy enforcement are performed on the server where the sender is logged in.

Error Handling

The File Transfer SPI can return information about the status of the scanning in three ways:

Use a return code from the call to the SPI, which indicates the scanning status

If a virus is found, two fields can be edited with the virus type and description. The pointers to these variables are passed by the application to the File Transfer SPI in the scanFile call.

The table below defines the possible values for the return code.

Table listing possible values for return codes.

Description	Value
The operation was completed successfully.	RC_OK
Scanning was not able to be completed	SCAN_ERR
A virus has been detected.	VIRUS_FOUND
This file does not require scanning	NO_NEED_SCAN

When you implement a file transfer DLL, it is your responsibility to ensure that information is not lost and that it is all transferred back to the application.

Effect of Virus-Scanning on Performance

How the file transfer, and especially virus scanning, affects performance on the Sametime server depends on the file transfer operation and implementation modes:

- In Always mode, a virus scanner is always running. This directly affects the performance due to the constant writing and reading I/O operations from the transferred files, alongside with the scanning itself.
- In whenAvailable mode, if there is a valid scanner (SPI), the functionality is the same as for the Always mode. However, if no SPI implementation exists, there are no I/O operations, which eases the load.
- When the mode is Never, only the policy is checked and then the channel is not captured any more, so performance is not affected.

Getting Started

File Transfer SPI Contents

The following contents of the Directory and Databases Access Toolkit are relevant to the File Transfer SPI :

- Sametime Directory and Database Access SPI documentation – this document.
- File Transfer SPI sample – provides a basic implementation of the SPI. It does not scan for viruses, and it only returns an error code for debugging issues, according to a file extension.
- File Transfer SPI header files – contain SPI definitions, syntax, variables, constants, return values, and so on.

Building the Virus-Scanning DLL

This toolkit does not provide a full version of the File Transfer DLL or service program. An example is provided for basic use, however there is no implementation for scanning itself.

When creating the virus-scanning DLL that implements the File Transfer SPI:

1. Follow the interface defined by the File Transfer SPI.
2. Develop a mechanism for reading a file from the file system and running a virus scan.
3. Read the General Considerations topic to ensure proper operation of the file transfer DLL.

General Considerations

These file transfer features must be considered when planning the file transfer implementation for a distributed community:

- All servers must run the same way: either all have file transfer turned on, or all have file transfer turned off.
- All servers must run in the same mode: either whenAvailable or Always.

Note If all the servers are not running in the same mode, some files are transferred and others are not. In this situation, the file transfers occur arbitrarily.

Synchronous implementation of the virus-scanning SPI affects the speed of file transfers in instant messaging.

Building and Installing for Windows

Follow these general steps for building and installing a file transfer DLL for Windows:

1. Study the provided sample, StFileTransfer.dll, in the toolkit samples\filetransfer directory.

   Note When building your file transfer DLL, leave the sample unchanged. You can use the sample project to verify that you have implemented all that was required.To activate your implementation of the virus-scanning DLL, in the sametime.ini file, in the [FileTransfer] section, set VIRUS_SCAN_DLL with your DLL name.

2. Do not install a new file transfer DLL until you have tested it thoroughly; DLLs directly affect server component operation.
3. Ensure you place the new DLL in the default directory that contains all other DLLs. The default directory is C:\Program Files\Lotus\Domino. If more than one server is included in the community, replace the DLL for each server.
4. Make the necessary changes in the stconfig.nsf file of each Sametime server in the community. For more information, see the ___ section.
5. To activate the created file transfer DLL, start and stop the Sametime server on which the DLL is installed. If the Sametime community contains more than one Sametime server, start and stop all Sametime servers in the community.

Building and Installing for IBM i

The Sametime server runs on multiple platforms. For more information regarding the list of platforms available for Sametime, refer to the Sametime Detailed System Requirements.

File Transfer SPI Parameters

This section contains information about:

- File Transfer SPI input parameters
- File Transfer SPI return messages
- File Transfer SPI constants

Input Parameters

This section describes the parameters that are transferred from the application to the SPI. Some of them are for SPI usage as input parameters, and some of are output parameters to be returned from the SPI to the application, in case of virus detection, to receive additional information.

```
typedef struct _STSession
{
      // sender name
      char sender_usename[MAX_USER_NAME_LENGTH];
      // receiver name
      char  receiver_username[MAX_USER_NAME_LENGTH];
      // session ID (channel ID)
      unsigned long  session_id;
} STSession;
/**
 * typedef the scan disk dll parameters
 */
typedef struct _ScanDiskInfo
{
```

```
        // pointer to the scanned file
        std::fstream* file;
        // file name
        char fileName[MAX_PATH_NAME_SIZE];
        // file size (bytes)
        int fileSize;
        // fstream size   (bytes)
        int   fstreamSize;
        // virus type – output parameter
        int   virustype;
        // virus description – output parameter
        char virustypedescription[MAX_BUF_SIZE];
        // trend needs it for some kind of traces
        STSession session;
} ScanParaInfo;
```

Both output parameters can have any kind of value, according to the virus detection. The values are transferred directly to the end user, and it is your responsibility to enter valid values for these parameters, as required.

Return Messages

The Table below describes the return values.

Table listing the values returned by the SPI functions.

Value	Return message	Description
0	RC_OK	Returned by the SPI functions when the operation completes successfully.
1	SCAN_ERR	Returned by the SPI functions when the scanning cannot be completed.
2	VIRUS_FOUND	Returned by the SPI functions when a virus is detected.
3	NO_NEED_SCAN	Returned by the SPI functions when scanning is not required.

Constants

The table below describes the constants defined in scanDiskApi.h.

Table listing the constants defined in scanDiskApi.h.

Value	Message	Description
256 characters	MAX_USER_NAME_LENGTH	Identifies the maximum length of the user name.
256 characters	MAX_BUF_SIZE	Indicates the maximum string length of the virus type description.
256 characters	MAX_PATH_NAME_SIZE	Identifies the maximum length of the path.

File Transfer SPI Functions

This section describes the File Transfer SPI functions provided by this toolkit:

- stDdaFtInit
- stDdaFtScanFile
- stDdaFtTerminate

stDdaFtInit Function

The stDdaFtInit function allows the File Transfer SPI to perform preparations that support its regular functionality. The function also checks for anything preventing the SPI from supplying this functionality.

The prVersion parameter is for forward compatibility. The virus-scanning DLL returns the current version, which is 12 in the field.

The syntax for the initialization call is:

```
int ST_DDA_API stDdaFtInit(ScanDiskInitParams *params);
```

When ScanDiskInitParams, is currently defined as:

```
typedef struct _ScanDiskInitParams
{
      // The ScanFile version
      int prVersion;
} ScanDiskInitParams;
```

stDdaFtScanFile Function

The stDdaFtScanFile function allows you to perform the virus-scanning method itself. This function is called synchronously from the application, as follows:

The application opens a thread for each virus-scanning call.

The application calls this method and waits until the method ends.

The syntax for the scanning call is:

```
int ST_DDA_API stDdaFtScanFile(ScanParaInfo* info);
```

For more information regarding ScanParaInfo, see the ____ section.

Input Parameters: ScanParaInfo*

Output Parameters: As described in ____, we return the virus type and description via the Input parameter.

Return Values: All values as described in the ____ section.

StDdaFtTerminate Function

The stDdaFtTerminate function allows the File Transfer SPI to perform a clear termination for the current scanning. This function is called once the application terminates.

The syntax for the terminate call is:

```
void ST_DDA_API stDdaFtTerminate(void);
```

Input Parameters: None.

Output Parameters: None.

Return Values: None.

Chapter 5. User Information SPI

UserInfo is a servlet application used by the Sametime server to retrieve users' personal details from one or more data repositories. This application is loaded by the http server and accepts http requests. UserInfo also functions as a Sametime server application that handles requests sent via Sametime channels.

Presenting user data when hovering over a buddy's name or when viewing a business card involves the following process:

1. A client request is sent to the UserInfo application.
2. The UserInfo application processes the request and retrieves the requested data from data resources.
3. The application sends an XML response to the client.

The UserInfo application should be installed on a Domino 7 server or later.

Every Sametime server should have its own installation of UserInfo. The XML files are not replicated between servers in a distributed environment.

The Domino server should be restarted after every change to the configuration XML for the changes to take effect.

UserInfo Application

UserInfo is a servlet application loaded by the Domino http task and terminated when the http task is restarted or stopped. When the servlet initializes, the server application mechanism is initialized as well.

The UserInfo application handles requests asking for details of a specific user. These requests may be either http requests or requests sent via Sametime channels.

Operations

The requests sent to the application support three types of operations:

1. **Requesting specific details for a specific user ID**. The servlet sends a query for only these details and returns an answer with the requested items (if available).

   Parameters:

   operation=1 – operation ID

   userId=<userId> – user ID for which information is requested

   FIELD1= <fieldname>

   FIELDn=<fieldname>

2. **Requesting the names of details in a specific set**. The request can contain the ID of a predefined set determined by the administrator. The UserInfo application responds with the list of details in the given set.

   Parameters:

operation=2 – operation ID

setId=<setId> – ID of the requested set

3. **Requesting the values of all details listed in a specific set for a specific user ID**. The UserInfo application queries the data resources for the given user ID and for the items listed only for the specified ID set ID.

   Parameters:

   operation=3 – operation ID

   userId=<userId> – user ID for which information is requested

   setId=<setId> – ID of the requested set

Configuration Data in UserInfoConfig.xml

The servlet accesses the UserInfoConfig.xml configuration XML file, which holds connectivity settings. This XML is constructed and populated with relevant data during the installation process, based on the administrator's choice of directory:

- If a Domino server is being used, UserInfoConfig.xml is populated with Domino-related settings.
- If an LDAP server is being used, LDAP settings are written to UserInfoConfig.xml.

While the application uses this XML file for configuration settings, it also reads updates resulting from administrator changes done via Sametime System Console (SSC). SSC allows managing of UserInfo settings that are stored in STConfig.nsf database on Community server. UserInfo service receives configuration updates from StConfiguration service that reads the relevant changes from StConfig.nsf.

The servlet reads any configuration settings when it loads. Therefore, for the servlet to process new configuration updates, the HTTP task should be restarted.

Note Updates via SSC refer only to the attributes retrieved from the Sametime directory (LDAP or Notes).

Resources Section

The XML file contains a "Resources" section that includes information for storage used in this deployment:

- If Sametime is configured to work with LDAP, the "Resources" section contains a "Storage" section whose type is "LDAP". The Storage section contains LDAP-specific parameters such as host name, base DN, port, etc.
- If the storage in use is the Domino directory, the "Storage" section type is defined as "NOTES".

Storage Sections

Every Storage section holds a list of details that can be retrieved from the relevant storage. The list provided in the installed version is the list of details supported by Sametime 7.5 and higher clients. Details can be added or changed but the Sametime client only supports the provided list. Every detail that can be retrieved from the storage is described in a separate Detail tag. The Detail tag contains the following properties:

- ID – name used by applications (client)
- FieldName – name of the attribute field in Notes, LDAP, or any other data resource
- Type – describes the MIME type: text/* for non-binary types

Every Storage section can contain a CommonField tag. This tag is provided for synchronization of the user IDs between two or more black boxes.

Since the user ID used by the first black box may not necessarily be stored in the other data repository, we define a common field.

To synchronize between storage types, a specific field must hold the same value in all data repositories. This can be a different field in each data source, but it should be mapped to the same ID in UserInfoConfig.xml

The value of the CommonFieldName property is the ID property in the Detail tag, which serves as a shared field between all used black boxes.

Each data repository should set the same value in the field mapped to the CommonFieldName value.

Example

Let's say we have a server using the Domino directory, and the Notes black box is the first black box. The commonFieldName property of the CommonField tag in the Storage section is set to "MailAddress".

In the NOTES storage section, the Detail tag whose ID is "MailAddress" is mapped to "internetAddress", which is the field that holds mail addresses in the Domino directory. Supposing an LDAP server is also used, the LDAP server should have a field that contains the same mail address as the one stored in Domino for the internetAddress field.

The LDAP server has a "mail" field that holds mail addresses. To allow synchronization of these two storage areas, make sure that the internetAddress field for a specific user in Domino contains the same value as the mail field in LDAP.

This mapping enables the application to search the new storage according to a value that is uniquely stored in it. This value can be configured based on the added black box.

The default value for commonFieldName is MailAddress. If the commonField tag does not appear in the Storage section of the black box queried first, CommonFieldName is set to MailAddress.

ParamsSets Section

The XML file contains a ParamsSets section that holds the predefined sets. A predefined set holds a list of parameters to retrieve in a certain order. The set for displaying business card information is Set #1.

The first set, Set #0, is a computed set that the servlet builds from all the ID properties in the Detail tags in all specified Storage sections. Detail sets can be added or changed.

BlackBoxConfiguration Section

The BlackBoxConfiguration section lists the black boxes used to retrieve users' details. Each BlackBox tag contains the following properties:

- **Type** – storage type this black box can handle. The value of this property should be similar to the value of the type property in the Storage opening tag. This value should be unique; each

BlackBox tag should have a different value for this property. The property can be omitted for newly added black boxes.

- name – name of the class that implements the relevant black box

  com.ibm.sametime.userinfobb.UserInfoLdapBB – LDAP black box

  com.ibm.sametime.userinfobb.UserInfoNotesbb – Notes black box

- MaxInstances – maximum number of instances of this black box that are created and maintained

Working with Black Boxes

This section explains how the UserInfo servlet works with black boxes. The servlet reads the black box names from the BlackBoxConfiguration section in UserInfoConfig.xml and initializes a number of instances of this black box as specified by the MaxInstances parameter in the black box tag.

The servlet sends a request to all the loaded black boxes and aggregates the responses received by each of them according to the black box priority. The priority of a black box is defined by its listing order. Therefore, a value retrieved from a black box that is listed first is sent in the response although a value for this detail was also received from the black box listed second.

Suppose, for example, we have a Sametime server that uses an LDAP server, and the UserInfo servlet is configured to use two black boxes. The first black box is the LDAP black box, as set by the installation, and the second is a user-defined black box. If we send a query to the servlet asking for the Title and MailAddress values for the user user@company_name.com and we receive a value for Title from both black boxes, then the Title that will be sent as a response is the Title retrieved from the LDAP directory, since the black box that handles data retrieval from LDAP was listed first.

The XML file created during installation contains the name of the black box that can retrieve data from the Sametime directory. This black box should always be listed first. Any other additional black boxes can be listed after it.

The servlet searches the first directory by sending a query to the first black box listed in the BlackBoxConfiguration section. Since Sametime's client sends the name that is found in the Sametime directory as the user ID parameter, the first black box queried is the black box that retrieves data from this directory. This black box is listed in the Black Box Configuration section during installation.

Additional black boxes may be listed after this black box.

Since only one set of parameters is provided for querying all black boxes, we use the commonField tag. A common field is a field that has the same unique value for each user in all data repositories. This tag has a CommonFieldName property whose value is the ID property of a detail tag.

The default common field is "MailAddress". In the Sametime directory (e.g., Domino directory), it is configured to the field internetAddress. In your data repository, it can be configured to the appropriate field that holds the same mail address as the internetAddress field in the Sametime directory.

If an additional black box is added, the servlet first tries to query it using the user ID parameter it receives from the client. If that call returns no values, a second query is sent to that black box using the value retrieved from the first black box for the CommonFieldName property. For example, if the common field is MailAddress and the first black box is Notes, the second query to the additional black box is sent with the value retrieved from the Domino directory for MailAddress.

User Information API

For a new black box to work correctly with the UserInfo application, the black box should implement the User Information API. The UserInfo application can load and process data received from any class that implements this API. By doing this, UserInfo allows data retrieval from a variety of data repositories.

A new black box for the UserInfo application implements the UserInfoBlackBoxAPI interface. The constructor of this class has no parameters. The interface methods include the following:

- init – for initialization
- terminates – for cleanup
- processRequest – the main method, which receives a RequestContext object with the user ID and the requested parameters. It returns a Response object that holds the values for the fetched details. The response object has a data member of type Map that is filled with the returning data. The key for the Map is the detail name and the value is a new DetailItem object, whose members are as follows:
- ID – the name that identifies the detail item. Its value is specified in the request as a detail Id parameter. Set its value using the setId method.
- Field Name – Stores the appropriate field name in the data storage that holds information for this detail item.
- Type – the appropriate MIME type. Set using the setType method ("text/plain" for string values, "image/x" for images where x is the type of image, such as gif, jpeg, etc).
- Value – an object that holds the data retrieved from the data storage. This value may hold text data or binary data. Use the methods setBinaryValue and setTextValue to populate this data member.
- setUserFound(true) – Called if the response object from processRequest contains a value for any of the requested details or if the user is found in storage but none of the requested details are found.

Refer to the Javadoc documentation for a complete description of the API classes and methods.

A sample black box is included in this package. While the sample returns hard-coded values for every request, it shows how to handle the request and the response objects.

To compile the sample black box, copy the UserInfo.jar from 9.0.0 Sametime Community server to the sample folder in the SDK. Then attach the UserInfo.jar to the sample project as an external library.

To configure the sample black box to run with the UserInfo application, follow the steps described in the next section.

Configuring the Servlet to Use the New Black Box

Admin Pages and UserInfoConfig.xml

For the servlet to load a new black box, you need to add an additional BlackBox tag to UserInfoConfig.xml. Add the tag under the Black Box Configuration section, after the black box that is

set by the installation. For the servlet to work with the connect client, ensure the default installed black box is listed first.

BlackBox tag properties:

- type – string that identifies the black box
- name – new class name, including its package
- MaxInstances – number of available objects for this type. If this value is x, the servlet creates x instances of this black box.

The response sent to the client aggregates the data returned from the different storage types according to the priority of the black box handling each data repository. To retrieve data from one resource instead of another:

1. Use UserInfoConfig.xml to configure the detail item that is defined in the CommonField tag. If you use the default value for the common field (MailAddress by default), add the corresponding field name for this item. Configure additional details to retrieve from the Sametime directory.

   Do not configure details you wish to retrieve by using the additional black box.

   Restart Domino when you finish updating the Admin pages.
2. Make sure that UserInfoConfig.xml contains a commonField definition in the Storage section that describes the Sametime directory parameters. If this tag is omitted, the value for this common field is MailAddress.
3. Make sure that the Storage section corresponding to the first black box in UserInfoConfig.xml does not include "Detail" tags for the items you intend to retrieve using the additional black box.
4. Add a BlackBox tag as described above.
5. Ensure that the code of the new black box queries the data resource for all requested detail items.
6. Restart the http service.

The storage details that now appear in UserInfoConfig.xml are only read by our black boxes.

Adding additional storage details in the XML is only necessary if your code is designed to read them from there.

If additional fields (other than the fields read by the installed black box) are to be retrieved from the additional black box and your code is intended to use Set #0, the newly added storage section should contain a name for the type property of the Storage opening tag and a Details section with Detail tags for each of the relevant fields. In each Detail tag, define the following:

- The ID that serves as the identifier name of this item
- The FieldName that serves as the corresponding field name in the data repository
- The type property (for client use): "text/plain" for text, "Image/gif", "image/jpeg", etc. for photo.

Domino Configuration

To complete the Domino configuration, add the name of the jar that contains the new black box implementation to the JavaUserClassesExt section in the notes.ini file, and restart the Domino server.

If JavaUserClassesExt contains variables that point to the name of a specific jar, the added black box must also follow this convention.

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk. IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you. Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
5 Technology Park Drive
Westford Technology Park
Westford, MA 01886

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only. All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp.
Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

These terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM

AIX

DB2

DB2 Universal Database Domino

Domino

Domino Designer

Domino Directory

i5/OS

iSeries

Lotus

Notes

OS/400

Sametime

System i

WebSphere

AOL is a registered trademark of AOL LLC in the United States, other countries, or both.

AOL Instant Messenger is a trademark of AOL LLC in the United States, other countries, or both.

Google Talk is a trademark of Google, Inc, in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Microsoft, and Windows are registered trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.