ICT PROJECT 258109
Monitoring Control for Remote Software Maintenance

Project number: 258109
Project Acronym: FastFix
Project Title: Monitoring Control for Remote Software Maintenance
Call identifier: FP7-ICT-2009-5
Instrument: STREP
Thematic Priority: Internet of Services, Software and Virtualization
Start date of the project: June 1st, 2010
Duration: 30 months

D3.5: 1st Prototype of the Error Reporting

Lead contractor: INESC ID
Author(s): João Garcia, Paolo Romano, Luís Rodrigues, Nuno Coração, Nuno Machado, João Matos
Submission date: November 2011
Dissemination level: Public

Project co-founded by the European Commission under the Seventh Framework Programme
© S2, TUM, LERO, INESC ID, TXT, PRODEVELOP
Abstract: This document describes the design and usage of the communication system and error reporting components of the FastFix project’s platform. These components are used for client-server messaging and for sending error reports and auxiliary files to the maintenance team in a FastFix installation.

This document has been produced in the context of the FastFix Project. The FastFix project is part of the European Community's Seventh Framework Program for research and development and is, as such, funded by the European Commission. All information in this document is provided "as is" and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability. For the avoidance of all doubts, the European Commission has no liability in respect of this document, which is merely representing the authors view.
## Contents

1 Introduction

2 Communication System
   2.1 Design
   2.1.1 Security
   2.2 Usage
      2.2.1 Creating Messages
      2.2.2 Sending Messages
      2.2.3 Receiving Messages

3 Error Report Generation System
   3.1 Design
   3.2 Usage

4 Running FastFix Components
   4.1 Infrastructure for FastFix Development
   4.2 Build and Run FastFix
      4.2.1 Build
      4.2.2 Run existing bundles

Annex A - Overview of Error Reporting Packages and Classes
   A.1 Packages:
   A.2 TRAC server
      A.2.1 Core
      A.2.2 defaultClient
   A.3 Secure Copy Logs
      A.3.1 secureCopy
   4.3 reporting / reporting.internal
1 Introduction

One of the goals of FastFix is to automate the communication between client devices running centralized applications with FastFix support and server-side maintenance teams receiving error reports from those applications on the FastFix support platform. This is achieved through the ability to have online communication between both sides and to submit error reports for faulty client-side application. These functionalities are performed by the two components overviewed in this document: the FastFix communication system and the FastFix error report generation system. This is a companion document to the source code of these two systems in the FastFix platform. Since the platform is based on the OSGi framework and in it components are encapsulated in bundles, both the communication and error report generation systems are OSGi bundles both available at the FastFix code repository (https://repository.fastfixproject.eu/svn/fastfix/, authenticated access). The communication system is contained in a bundle for the client side, eu.fastfix.client.communication (8 classes, 20 methods, 115 lines of code), and a bundle for the server side, eu.fastfix.server.communication (11 classes, 36 methods, 166 lines of code). The error report generation system is contained in the eu.fastfix.client.user.reporting bundle (16 classes, 98 methods, 438 lines of code). The following sections detail the design and usage of these components.
2 Communication System

2.1 Design

The FastFix communication system (FCS) is a bidirectional message exchange mechanism. Any bundle on one of the sides of the client-server divide may send a message to another bundle on the other side. A message is a generic object containing an identifier of the destination bundle and an arbitrary object appended by the sender. Message sending is performed by a non-blocking to the FCS and receiving is done using a blocking call to that same system.

The transport mechanism that is used by in the FCS is TCP sockets with object serialization based on Java’s `java.io.ObjectOutputStream` and `java.io.ObjectInputStream` classes.

The FCS assumes that client-server connection is established by the client, e.g. Figure 1. The client does not need to know the FastFix server’s address, which is an installation configuration. Likewise, the server does not need to know the clients’ addresses: when it receives a message, the message contains the client’s identifier and the server can use this identifier to reply to this client.

![Diagram of message sending using FCS](image)

**Figure 1 Example of sending a message using FCS**

2.1.1 Security

Since the FastFix communication system will be used to communicate potentially private information regarding users’ devices, physical context (e.g. location) and personal data, the platform must ensure that the data transmission is secure, i.e. that malicious third parties cannot eavesdrop, modify or intercept messages from the client devices of FastFix users to
maintenance teams on the server side. Therefore the communication channels between FastFix clients and servers are authenticated and encrypted using Java Secure Socket Extension. FastFix servers are authenticated using X.509 certificates which are installed during the installation of the FastFix platform on the client device.

2.2 Usage

2.2.1 Creating Messages
FastFix messages are by default data transport envelopes devoid of payload. The FastFixMessage class only provides some methods for handling metadata information (addresses, identifiers, etc…). These metadata do not need to be accessed during standard communication. The only exceptional case being that, on the server side, one may want to know which client sent a particular message. This is done by calling getClientID method (see section 2.2.3) on the FastFixMessage object.

In order to create a FastFix message with meaningful content, it is necessary to extend the FastFixMessage class with data attributes that are the message’s payload. These classes must be included in the eu.fastfix.server.communication bundle in the messages package.

2.2.2 Sending Messages
In order to send a message to another bundle, it is necessary to first obtain a reference to the communication system by calling the getInstance method of the CommunicationFactory class. Then one must create an instance of the relevant derived class of FastFixMessage and then call the send method. This method will be called on the IServerDataSender interface of the eu.fastfix.server.communication bundle if the message is sent from server to client and on the IClientDataSender of the eu.fastfix.client.communication bundle if the message is sent in the reverse direction. When sending messages from a client to the server, calling send requires two parameters, m (a FastFixMessage object) and the bundleId. The m parameter is the message to be sent and the bundleId parameter designates the bundle that will receive the message. The bundleId identifies the destination bundle for the message. The possible values for the bundleId are listed in the eu.fastfix.server.communication.utils.Parameters.FastFixID.
enumerate type. If the message is sent from the server to a client there is an additional 3rd parameter, clientId which is the unique identifier of the destination client.

### 2.2.3 Receiving Messages

FastFix messages are received by first obtaining a reference to the communication system by calling the getInstance method of the CommunicationFactory class (eu.fastfix.server.communication bundle on the server side or eu.fastfix.client.communication on the client side). Messages can be received both synchronously and asynchronously.

Synchronous reception (see Figure 2) is blocking and performed by calling the receiveBlocking(FastFixID bundleID) method on the IClientDataReceiver interface of the bundle on the client and on the IServerDataReceiver interface of the eu.fastfix.server.communication bundle on the server. The possible values for the bundleID are listed in the eu.fastfix.server.communication.utils.Parameters.FastFixID enumerate type. This call is **blocking** and will block the calling thread until a message arrives for that bundle. When a message arrives for the requested bundle, the caller is unblocked and the message (FastFixMessage) is returned from the call. The received message contains a valid clientId (a unique identifier of the client) that can be read using the getClientID method.

![Figure 2 Synchronous message reception in FCS](image-url)
Asynchronous reception (see Figure 3) is performed by call register on the IClientDataReceiver or IServerDataReceiver (depending on whether the caller is on the client or server side). The parameter is an object that implements IAsynchDataReceiver which is an interface containing methods for providing its bundleid (FastFixID getBundleID()) and for accepting a message (void notify(FastFixMessage message)).

Figure 3 Asynchronous message reception in FCS
3 Error Report Generation System

3.1 Design

The error report generation system allows FastFix components to send error reports to the maintenance team. These reports have two parts: a report header with a description of the error, which is added to a ticket tracking system, and a set of report log files.

The report header contains information describing the error and is implementation dependent on the specific error database being used.

The log files are transparently handled by the error reporting system, which simply transfers them to the maintenance server and stores them in a folder associated with the specific error report. They can be any files that the component submitting the error report specifies.

3.2 Usage

An error report is generated by creating an ErrorReport object. This requires creating a report header and, if wanted, adding a list of report logs.

The report header (abstract ReportHeader class) is used to create an entry in a bug database. Currently, FastFix supports the TRAC bug-tracking server (trac.edgewall.org) via the TRACReportHeader class in the eu.fastfix.client.user.reporting bundle.

The TRACReportHeader contains a set of string attributes characterizing the error. Those attributes depend on the specific TRAC version and installation being used. For the current TRACReportHeader, those attributes are:

- summary: a general description of the error;
- description: a more detailed description of the error;
- type: error type (can be “defect”, “issue”, “task”, “enhancement”). In the case of FastFix, the reasonable value to use is “defect” since most error reports;
- priority: error priority (can be “blocker”, “critical”, “major”, “minor”, “trivial”);
- milestone: name of the application development milestone. Since FastFix targets production environments, can typically be left empty (“”);
- **component**: name of the component that failed. Depends on the particular application being monitored by FastFix;

- **version**: version of the application where the error happened;

- **keywords**: keywords for search purposes;

- **cc**: TRAC users who will be notified of the error;

- **owner**: TRAC user who owns the report.

The list of report logs is a simple `java.util.List` of strings containing the absolute paths on the client device of all the log files that should be transferred to the FastFix maintenance server.

Once the `ErrorReport` is created by calling one of its constructor (report header only or report header plus log files), it is submitted to the FastFix maintenance server by calling `generateErrorReport` on the `IErrorReportGenerator` interface of the `eu.fastfix.client.user.reporting` bundle. This call returns a `ReportStatus` object containing a global success flag, called `status` (true for success and false in case any of the steps failed), the bugtracker’s `ticketNumber` and a flag, `ticketCreated`, indicating whether the ticket was created or not. Additionally, the ticket status has properties describing the outcome of the log files transfer process:

- **CompleteTransfers**, the number of successfully transferred log files (obtained by the `getCompleteTransfers` method).

- **FailedTransfers**, the number of failed transfers of log files (obtained by the `getFailedTransfers` method).

- **CompleteTransferNames**, the list with the names of the successfully transferred log files (obtained by the `getCompleteTransferNames` method).

- **FailedTransferNames**, the list with the names of the log files whose transfers failed (obtained by the `getFailedTransferNames` method).
4 Running FastFix Components

Currently the FastFix platform is run as a set of OSGi components. The best way to test the components described in this document is in the context of the FastFix development environment.

4.1 Infrastructure for FastFix Development

The following software is required to develop FastFix components:

- Maven 3.0.3 (Homepage: [http://maven.apache.org/](http://maven.apache.org/))
- [http://m2eclipse.sonatype.org/sites/m2e/](http://m2eclipse.sonatype.org/sites/m2e/)
- M2Eclipse Maven Plug-In for Eclipse (Homepage: [http://m2eclipse.sonatype.org](http://m2eclipse.sonatype.org). Install via Updatesite [http://m2eclipse.sonatype.org/sites/m2e/](http://m2eclipse.sonatype.org/sites/m2e/): Help > Install New Software > Add... > Use Updatesite URL > Select all options)

4.2 Build and Run FastFix

4.2.1 Build

For each platform, client and server, there is a dedicated Maven Project called "parent". This is needed in order to build all required bundles for the client or the server automatically with Maven. The server parent bundle is called eu.fastfix.server.parent and the client parent project eu.fastfix.client.parent. To build the client bundles

1. Right-click on the client parent project and select "Run As... -> Maven Install".
2. Wait for Maven to build the packages and generate the binaries and Manifest.MF files. As a result, you should get something similar to this:
3. Refresh your package explorer by pressing F5 or right-click on it and press "Refresh" in order to prevent "Out of sync" while running the bundles.

Perform analogous steps for the server bundles.

4.2.2 Run existing bundles

To run the current bundles

1. Select one of the projects by right-clicking on it
2. Choose "Run As -> Run Configurations...".
3. Double-Click on "OSGi Framework" item from the list placed left. This will create a new configuration file for running the bundles.
4. In the "Arguments tab", replace the proposed "Program arguments" with
   1. "-os ${target.os} -ws ${target.ws} -arch ${target.arch} -nl ${target.nl} -console -clean"
5. Press "Run"
6. In the "Console" view, type "ss fastfix" and press enter in order to display all FastFix bundles. If any of the FastFix bundles has "Resolved" state, start it by typing "start xx" where "xx" is the number of the plugin which is not activated.

One only have to create a new configuration and specify arguments once. Later just hit on "Run as OSGi Framework".
Annex A - Overview of Error Reporting Packages and Classes

A.1 Packages:

- eu.fastfix.client.user.reporting
- eu.fastfix.client.user.reporting.internal
- eu.fastfix.client.user.reporting.internal.core
- eu.fastfix.client.user.reporting.internal.defaultClient
- eu.fastfix.client.user.reporting.internal.secureCopy

This implementation allows the client to insert a new ticket in a TRAC server and to secure copy log files to some remote location.

A.2 TRAC server

The packages containing the implementation of the TRAC ticket creation are the core package and the defaultClient package. The core package contains an abstract implementation of the functions required to create a ticket. Some are implemented but others are not. The defaultClient package provides implementations for the functions not implemented in the core package. The reason for this choice of implementation is simple, there are many ways to implement some functions and some may not be compatible with all TRAC servers, so they should be separated. If by any chance incompatibility issues arise, one only has to extend the core package.

A.2.1 Core

- TicketHeader: An abstract class that contains the fields of the ticket form.
- TracConnector: An abstract class to connect with a TRAC server.
- TicketStatus: A class containing the status or result of an insert operation.
- TicketFailException: An exception thrown when something went wrong with the creation of the ticket. It also provides details of the underlying failure and the TicketStatus.
A.2.2 defaultClient
- DefaultTicketHeader: The default extension of the TicketHeader abstract class.
- DefaultTracConnector: The default extension of the TracConnector abstract class.

A.3 Secure Copy Logs
A.3.1 secureCopy
- SecureCopyOperations: A class that provides functions to secure copy files to remote locations.
- SecureCopyManager: This class manages the methods from SecureCopyOperations.
- SecureCopyStatus: A class that holds the result of a secure copy operation.
- SecureCPFailException: An exception thrown when something went wrong with the secure copy operation. It also provides details of the underlying failure and the SecureCopyStatus.

4.3 reporting / reporting.internal
- ReportStatus: Contains both the TicketStatus and the SecureCopyStatus.
- ReportFailedException: Thrown if the error report operation failed. It contains details of the failure, TicketStatus and SecureCopyStatus.
- UserReportingServiceImpl: Contains the functions to submit an error report.
- ErrorReport: A class that joins the ticket header and the files to be secure copied.