



FastFix Platform User Manual

Authors:

Dennis Pagano, Tobias Roehm, Emitzá Guzmán,
Sergio Zamarripa, João García, Benoit Gaudin, Javier
Cano, Christophe Joubert, Walid Maalej



Project co-founded by the European Commission
under the Seventh Framework Programme

© S2, TUM, LERO, INESC ID, TXT, PRODEVELOP

Abstract: This document describes the final version of the FastFix platform. It is a supplement to the source code, which can be accessed on the FastFix open source project repository at SourceForge. It gives a conceptual overview of the platform as a whole, and describes how the FastFix platform is typically deployed, set up, and used by end-users and developers.

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 | 5 |
| 2 | Platform Overview | 7 |
| 2.1 | eu.fastfix.client | 7 |
| 2.2 | eu.fastfix.common | 8 |
| 2.3 | eu.fastfix.targetapplication | 8 |
| 2.4 | eu.fastfix.server | 9 |
| 2.5 | eu.fastfix.dependencies | 9 |
| 2.6 | Summary | 10 |
| 3 | User Manual | 11 |
| 3.1 | Deployment and Setup | 11 |
| 3.1.1 | FastFix Configuration | 12 |
| 3.1.2 | FastFix Client Setup | 24 |
| 3.1.3 | FastFix Server Setup | 25 |
| 3.1.4 | FastFix Sensors Setup | 26 |
| 3.2 | Platform Usage | 29 |
| 3.2.1 | Context Observation | 29 |
| 3.2.2 | Event Correlation | 30 |
| 3.2.3 | Pattern Mining | 33 |
| 3.2.4 | Error Reporting | 36 |
| 3.2.5 | Fault Replication | 37 |
| 3.2.6 | Patch Generation and Self-Healing | 38 |
| 4 | Developer Manual | 40 |
| 4.1 | Configuration Extensions | 40 |
| 4.2 | Feature Extensions | 40 |
| 4.2.1 | Context Observation | 40 |
| 4.2.2 | Event Correlation | 42 |
| 4.2.3 | Pattern Mining | 42 |
| 4.2.4 | Error Reporting | 43 |
| 4.2.5 | Fault Replication | 44 |
| 4.2.6 | Patch Generation and Self-Healing | 44 |
| 5 | Summary | 45 |
| | Bibliography | 46 |



List of Figures

| | | |
|------|----------------------------------------------------------------|----|
| 2.1 | Conceptual overview of the FastFix platform. | 7 |
| 2.2 | FastFix client platform UI. | 8 |
| 2.3 | FastFix server platform UI. | 9 |
| 3.1 | Deployment of FastFix Platform. | 11 |
| 3.2 | Configuration of Application Bridge | 12 |
| 3.3 | Sensor Configuration | 13 |
| 3.4 | Datastore Configuration | 13 |
| 3.5 | Event correlation configuration element | 15 |
| 3.6 | Pattern mining configuration element | 17 |
| 3.7 | Ontology configuration element | 18 |
| 3.8 | Communication configuration element | 18 |
| 3.9 | Issue tracker configuration element | 19 |
| 3.10 | Ticket browser configuration element | 21 |
| 3.11 | FastFixSH configuration attributes | 22 |
| 3.12 | Sample Expert File | 23 |
| 3.13 | FastFixSH configuration within FastFix Client. | 24 |
| 3.14 | Starting registered sensors via the FastFix client UI. | 25 |
| 3.15 | Event Correlation Menu | 30 |
| 3.16 | General Information Tab | 31 |
| 3.17 | Symptoms Tab | 32 |
| 3.18 | Cause Tab | 32 |
| 3.19 | List of existing patterns of error | 33 |
| 3.20 | Pattern Mining menu | 34 |
| 3.21 | Detail of a mined pattern | 35 |
| 3.22 | Dialog showing a new mined pattern | 35 |
| 3.23 | Validating error patterns | 36 |
| 3.24 | Viewing an error report | 37 |
| 3.25 | GUIAnon usage instructions | 38 |
| 3.26 | FastFix Self-Healing plugin menu | 39 |



1 Introduction

This document describes the final version of the FastFix platform. It is a supplement to the FastFix source code, which can be accessed in the FastFix source code repository on SourceForge¹. The document first gives a general conceptual overview of the platform as a whole and summarizes current source code metrics. Then, it describes how FastFix can be used from the point of view of two actors. First, the FastFix *users*, i.e. maintenance engineers who want to remotely maintain their software with FastFix, and need to deploy and setup the FastFix platform. Second, *developers*, who want to develop for and extend FastFix, for instance by writing additional sensors, adding rules, or adding possibilities to configure the platform.

For more details on specific aspects and functions of the FastFix platform, we refer the reader to the corresponding deliverables, as summarized in the following list:

- First Prototype of Context Observer [9]
- First Prototype of the User Profiler [13]
- First Prototype of the Error Reporting System [2]
- Refined and Integrated Version of Context Observer, User Profiler and Error Reporting [12]
- First Prototype of the Event Processor [1]
- First Prototype of the Pattern Mining Module [15]
- Second refined prototype of the event correlation component [14]
- First and Second Prototype of the Execution Recorder and Replayer [3, 4]
- First, Second, Third, and Fourth Prototype of the Self-Healing and Patch Generation Component [7, 6, 5, 8]

This document does not describe in detail how to set up a development environment in order to build the FastFix platform from source code. FastFix is a relatively big system, and the project utilizes specialized technologies for the build process. Therefore, in this document we concentrate on giving additional, conceptual information on the source code, and illustrate deployment and usage scenarios. For information on how to start developing for FastFix, we refer the reader to [10] and to the project's Wiki documentation²³⁴.

In Section 2, we give a conceptual overview of the final FastFix platform. We illustrate how and why bundles are distributed among different namespaces and give details using

¹<https://svn.code.sf.net/p/fastfixrsm/code/trunk>

²http://fastfixproject.eu/wiki/Howto:_Set_up_development_infrastructure

³http://fastfixproject.eu/wiki/Howto:_Main_development_use_cases

⁴http://fastfixproject.eu/wiki/Howto:_Start_FastFix_from_Eclipse



code metrics. In Section 3, we describe how the FastFix platform can be deployed, set up, and used for remote maintenance. In Section 4, we describe, how developers can configure, extend, and change the FastFix functionality, in order to be able to provide remote maintenance services tailored to their customers' requirements. Section 5 summarizes and concludes this document.



2 Platform Overview

FastFix is a *remote* software maintenance platform. As Figure 2.1 illustrates, its components are distributed among two environments. First, the environment where the target application is running, and second the maintenance environment where development, maintenance, and testing tools are installed. Consequently, the FastFix system consists of two main components: the FastFix client and the FastFix server. Both server and client communicate and exchange data with each other.

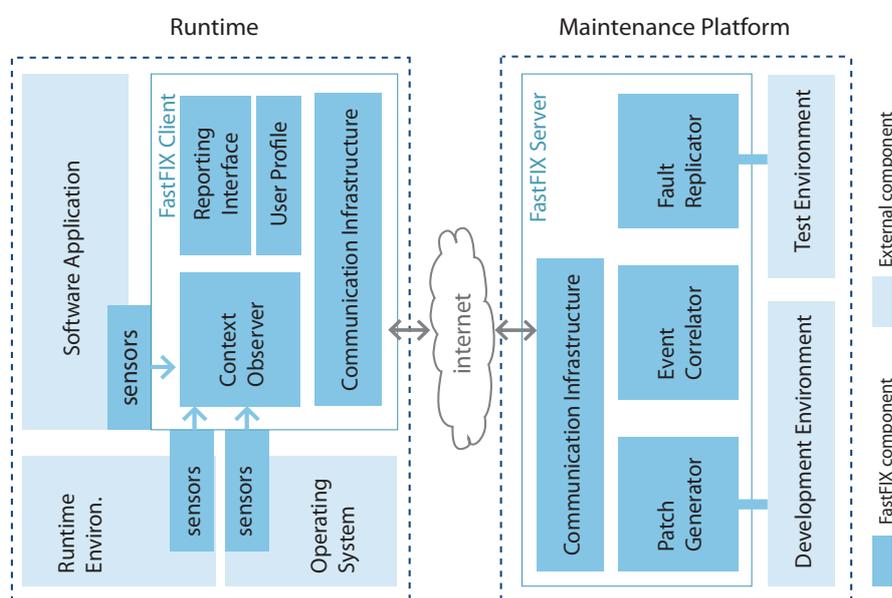


Figure 2.1: Conceptual overview of the FastFix platform.

As described in [11], FastFix components are structured into namespaces according to their responsibility in the platform. The namespace of each code bundle can be identified from the bundle name. Currently, the integrated platform contains 79 bundles¹ belonging to 5 different namespaces. In the following, we briefly explain all five namespaces and give additional details in terms of code metrics. Details on the source code in terms of interfaces and packages can be found in [11] as well as in the source code documentation² (JavaDoc).

2.1 eu.fastfix.client

The *client* namespace contains bundles constituting the FastFix client component. The FastFix client runs in the environment of the target application (i.e. in the application

¹Including bundles needed to group other bundles (“parent bundles”)

²The project source code is available at SourceForge: <https://svn.code.sf.net/p/fastfixrsm/code/trunk>

usage environment). Its main purposes are (a) to collect context information monitored by sensors, (b) to perform data pre-processing tasks before sending information to the maintenance site (for security and performance), (c) to access supervisor mechanisms in the target application (for self-healing), and (d) to provide a user interface to allow users to regulate the FastFix functionality (e.g. start and stop sensors).

The *client* namespaces currently contains 100 classes with 504 methods in 37 packages (10 bundles), with a total of 4,632 lines of code in 110 files. Figure 2.2 shows a screenshot of the FastFix client UI including three registered sensors.

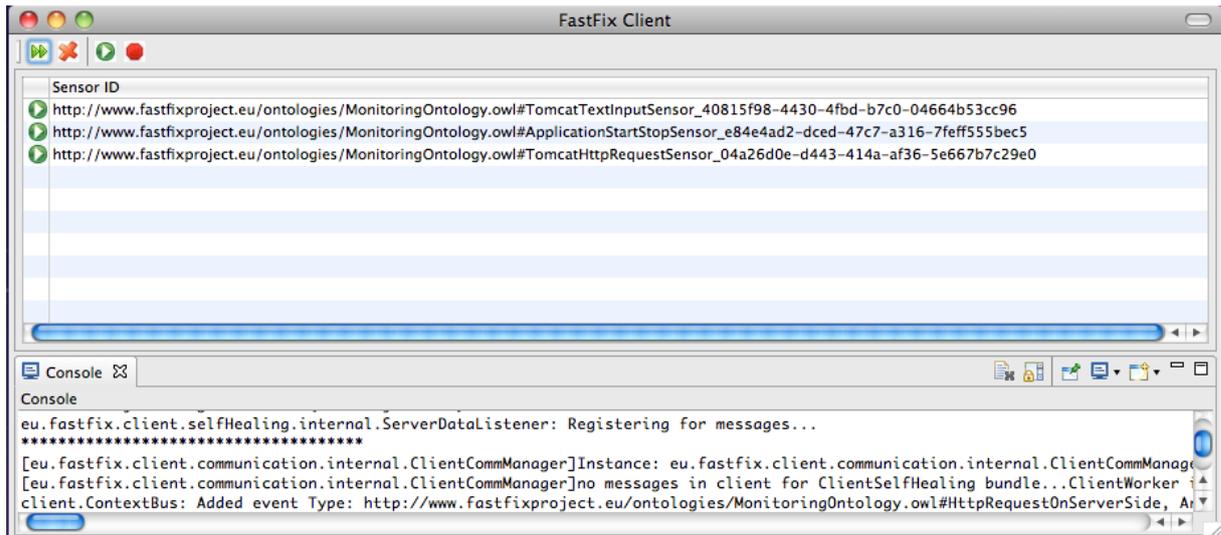


Figure 2.2: FastFix client platform UI.

2.2 eu.fastfix.common

The *common* namespace contains bundles which are needed by both the FastFix client and server. In the current state of the system, the FastFix common namespace includes bundles from nine different areas of concern, including context observation, persistency, error reporting, fault replication, and communication between client and server. More details can be found in [11].

The *common* namespaces currently contains 340 classes with 2,113 methods in 56 packages (12 bundles), with a total of 17,894 lines of code in 343 files.

2.3 eu.fastfix.targetapplication

The *targetapplication* namespace contains bundles that are supposed to run in the runtime environment of the target application. Typically such components are sensors (or actuators). Bundles in this namespace communicate with the FastFix client via the interfaces of the application bridge (cf. [11]).

The *targetapplication* namespaces currently contains 117 classes with 824 methods in 32 packages (11 bundles), with a total of 9,267 lines of code in 114 files.

2.4 eu.fastfix.server

The *server* namespace contains bundles constituting the FastFix server component. The FastFix server runs in the maintenance environment (or in the application engineering environment). Its main purposes are (a) to collect information sent by FastFix client, (b) to investigate this information and detect performance degradation trends, errors, and possible causes, (c) to create and update user profiles, (d) to provide access to issue trackers, (e) to allow maintenance engineers to replay errors, (f) to create patches and send these patches to clients to self-heal them, and (g) to provide a user interface to allow maintenance engineers to access the FastFix functionality (e.g. error replay and patch generation). Figure 2.3 shows a screenshot of the FastFix server UI, which mainly consists of a log and specific FastFix menus.

The *server* namespaces currently contains 415 classes with 2,826 methods in 115 packages (21 bundles), with a total of 40,164 lines of code in 410 files.

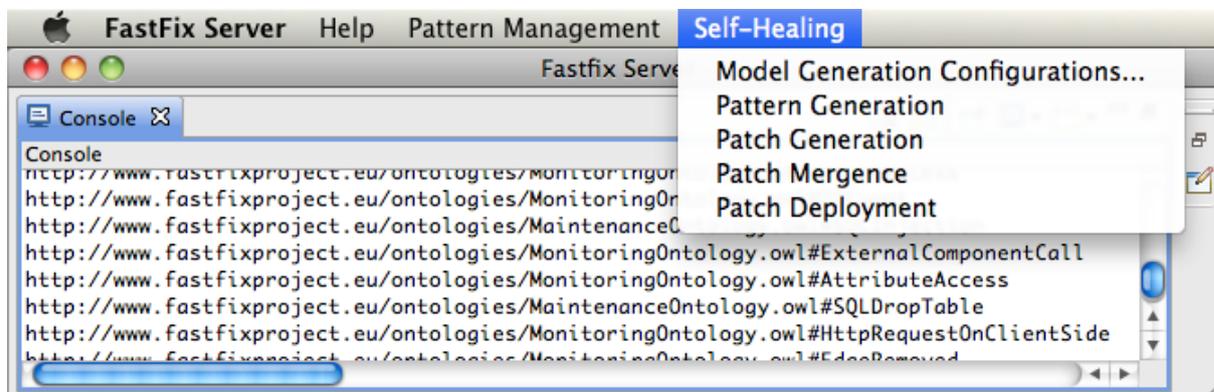


Figure 2.3: FastFix server platform UI.

2.5 eu.fastfix.dependencies

The dependencies namespace contains third-party libraries which we have wrapped in OSGi bundles. This process is sometimes necessary to be able to use specific libraries in an OSGi context. We created OSGi wrappers for the following 9 third party components:

- eu.fastfix.dependencies.apache.http – wrapping org.apache.http
- eu.fastfix.dependencies.axis – wrapping org.apache.axis
- eu.fastfix.dependencies.drools – wrapping org.drools
- eu.fastfix.dependencies.flexjson – wrapping net.sf.flexjson
- eu.fastfix.dependencies.javassist – wrapping javassist
- eu.fastfix.dependencies.jena2 – wrapping com.hp.hpl.jena
- eu.fastfix.dependencies.jpj – wrapping Java PathFinder
- eu.fastfix.dependencies.mysql – wrapping mysql jdbc connector
- eu.fastfix.dependencies.soot – wrapping soot, polyglot, and jasmin

2.6 Summary

Table 2.1 summarizes the code metrics of the FastFix open source remote maintenance platform.

Table 2.1: Code Metrics of Integrated FastFix Platform.

| | # bundles | # packages | # classes | # methods | # LOC | # files |
|------------------------------|-----------|------------|-----------|-----------|--------|---------|
| eu.fastfix.client | 10 | 37 | 100 | 504 | 4,632 | 110 |
| eu.fastfix.common | 12 | 56 | 340 | 2,113 | 17,894 | 343 |
| eu.fastfix.targetapplication | 11 | 32 | 117 | 824 | 9,267 | 114 |
| eu.fastfix.server | 21 | 115 | 415 | 2,826 | 40,164 | 410 |
| eu.fastfix.dependencies | 9 | - | - | - | - | - |
| sum | 63 | 240 | 972 | 6,267 | 71,957 | 977 |

3 User Manual

In this section, we provide a user manual for FastFix, i.e. we describe how the FastFix platform is typically deployed, which steps are necessary to set the platform up, and how it can be used for remote maintenance.

3.1 Deployment and Setup



Figure 3.1: Deployment of FastFix Platform.

Figure 3.1 illustrates how FastFix is deployed in a typical remote maintenance scenario (components, machines, and connections needed for FastFix are shown in blue). The “Client Machine”¹ refers to any machine hosting parts of the target application. In the case of a simple desktop application, this might just be a standard desktop client. In the case of a three tier application, it might refer to the presentation tier (desktop clients), logic tier (server hosting business logic), and data tier (server hosting database) machines respectively. For FastFix, a “Client Machine” refers to any machine where data about the application, runtime environment, or user can and shall be monitored. To this end, the FastFix client and sensors are deployed onto these “Client Machines”. The “Server Machine” is a (logical) additional machine added to the application deployment scenario, which hosts the FastFix server application. This machine (and the FastFix server) is connected to the clients via internet². The “Tracker Machine” denotes the machine where the issue tracker is running (typically a stand-alone server). The FastFix server connects at runtime to this server in order to access the issue tracker.

The following sections describe on a high level how the FastFix client, server, and sensors have to be installed. Depending on the specific platform and target application the details of these steps may vary, and additional steps may be required.

¹We put the term “Client Machine” in quotation marks to indicate that these machines are not necessarily clients in terms of the target application language, but in terms of FastFix.

²Note that the FastFix Server application might in theory also be installed on the “Tracker machine” or the “Client machine”.

In Section 3.1.1, we show how to configure the FastFix platform components. In the following two sections, we describe how to obtain and run the FastFix client (Section 3.1.2) and server (Section 3.1.3). Finally, in Section 3.1.4 we illustrate how to set up and connect FastFix sensors.

3.1.1 FastFix Configuration

Both FastFix server and client should be configured with a configuration file in the FastFix home directory³. The configuration file allows maintenance engineers to specify the behavior of the following components⁴:

3.1.1.1 Context Observation

Communication between sensors and FastFix client The application bridge is the component of the FastFix client to which the sensors communicate. The configuration parameters shown in Figure 3.2 are used both by the FastFix client and the sensors to communicate. In the following we briefly describe each configuration parameter.

- **sensorsAutoStart** (boolean):
Determines if sensors are started automatically when they are registered or if this has to be done manually by the user via the FastFix client.
- **RMIBridge active** (boolean):
Determines whether the RMI bridge, i.e. the component of the FastFix client used for communication via RMI, is available or not.
- **RMIBridge host** (Hostname), **port** (Port), and **name** (String):
Setup for the RMI connection. This is used by the FastFix client to provide services via RMI and the sensors to connect to FastFix client and consume the services.
- **HTTPBridge active** (boolean):
Determines whether the HTTP bridge, i.e. the component of the FastFix client used for communication via HTTP, is available or not.
- **HTTPBridge host** (Hostname), and **port** (Port):
Setup for the HTTP connection.

```
<!-- The ApplicationBridge element has configuration about the RMIBridge and the HTTPBridge. -->
<ApplicationBridge id="fastfix_applicationbridge_configuration" sensorsAutoStart="true">
  <!-- RMI bridge in use or not, with location (host and port) and name to locate it -->
  <RMIBridge active="true" host="localhost" port="1099" name="RMIBridge" />
  <!-- HTTP bridge in use or not, with location (host and port) -->
  <HTTPBridge active="true" host="localhost" port="1180" />
</ApplicationBridge>
```

Figure 3.2: Configuration of Application Bridge

³The FastFix home directory is (a) the folder specified by the Java environment property “fastfix.home” or (b) the folder “fastfix” in the user’s home directory. FastFix expects read and write permissions in all the FastFix home directory and subdirectories and files.

⁴More information can be found in the project wiki under http://fastfixproject.eu/wiki/Configuration_file_structure

Sensor Behavior The behavior of sensors can be configured using the following parameters. An example configuration is shown in Figure 3.3.

- **heartbeatInterval** (int):
The time interval (in milliseconds) in which sensors send heartbeat commands to the FastFix client to indicate that they are still alive and to receive instructions. Used by sensors that are implemented in own threads and not as listeners.
- **registerSensorsOnAppStart** (boolean):
Determines whether sensors should be registered automatically on application start or by explicit trigger of the user.

```
<!-- Parameters used by the sensors. -->  
<Sensors heartbeatInterval="60000" registerSensorsOnAppStart="true" />
```

Figure 3.3: Sensor Configuration

Datastore Monitored events are stored in the datastore, i.e. a MySQL database holding event data. The following parameters can be used to configure the database access for FastFix client and FastFix server. An example configuration is shown in Figure 3.4.

- **driver** (String):
The database driver that should be loaded and used (in Java).
- **url** (String):
The url under which the database can be accessed.
- **db** (String):
The name of the database in the MySQL server instance.
- **user** (String), **password** (String):
Credentials for MySQL server.
- **cache** (int):
The size of the event cache for database access, i.e. the number of events that are cached by FastFix client and FastFix server and written to database together.

```
<!-- The DataStore element has parameters to configure the datastore component of FastFix. -->  
<Datastore id="fastfix_datastore_configuration">  
  <!-- The Connection element has parameters to connect to a database, both in client and server. -->  
  <Connection driver="com.mysql.jdbc.Driver" url="jdbc:mysql://localhost/FastFixServer"  
    db="FastFixServer" user="DbUsername" password="DbPassword" cache="1"/>  
</Datastore>
```

Figure 3.4: Datastore Configuration

3.1.1.2 Event Correlation and Pattern Mining

Event correlation component FastFix event correlation system will automatically detect an error pattern during the application's operation, generating automatic tickets and gathering all the information that can be useful for further analysis. In order to properly configure the event correlation components, we need to define the file system folders where the required files are located. This configuration is defined by specifying several file paths, which are located under the FastFix home directory. Event correlation configuration allow the developer to disconnect the module from both the context and the report system.

- **DRL (String):** Both items, `path` and `url`, must point to the same resource: `maintenance-rules.drl`. This file contains the rules which are used to detect patterns of error, and it's an internal, i.e. it is not recommended to change the default value, since it must be in the classpath of FastFix. It is used by different classes, which use different methods to load the file. So, it is necessary to use two different parameters.
- **Changeset (String):** The item `path` points to the folder to be scanned in order update the rule engine with changes in the rules. It's also an internal file, so it is not recommended to change the value.
- **ContextSystem:**
 - **plugged (boolean):** This field is used for development purposes. If its value is *true*, the event correlation module will be fed with real context events. However, sometimes simulating the events is useful, specially for testing new event correlation features, in this case, *false* is the appropriate value. The correct value for production environments is *true*.
 - **nonAnonymousEvents (String):** It refers to the context event types that contain information about the user who create it. For example: `http://www.fastfixproject.eu/ontologies/MonitoringOntology.owl#TextInput+http://www.fastfixproject.eu/ontologies/MonitoringOntology.owl#HttpRequestOnServerSide`
- **UserReportingSystem**
 - **plugged (boolean):** This field is used for development purposes. If its value is *true*, the event correlation module inserts a ticket in the issue tracker (if a fault is detected). However, in development phase, it's recommended change the value to *false*, in order to not flood the tracking system. The correct value for production environments is *true*.
 - **post (boolean):** It refers to the granularity of the information shown in reports, i.e.: if its value is true, the module includes extra information about some context events.
- **CorrelationData**
 - **path (String):** It refers to the folder where the collected results are going to be stored, in other words, the location of the file containing information about the error being detected, as well as the client name and other configurable fields described in the configurable fields described in `generalConf` and `rdfConf`.

- **generalConf** (String): It is the folder of the properties file describing the general fields to be retrieved from the events associated to the current detected error. The general fields consist of a group of default fields (**OntologyId**, **ClientName**, **UID**) and a list of configurable fields, defined inside the properties file. The properties file is a list of name-value pairs, where the name represents the column in the generated csv file, while the value represents the event data property to be retrieved from the FastFix events.
- **rdfConf** (String): It refers to the folder of the properties file describing the rdf fields to be retrieved from the server datastore, corresponding to previous events that might be associated with the current situation of error. Again, this properties file represents several name-value pairs, where the first element describes the name of the column in the resulting csv file, while the second is the event data property to be retrieved.

The following figure represents an example of the event correlation configuration element of the FastFix configuration file:

```
<!-- The EventCorrelator element has parameters to configure the event correlator
components of FastFix. -->
<EventCorrelator id="fastfix_event_correlator_configuration">
  <DRL path="classpath:drl/maintenance-rules.drl" url="file:///./drl/maintenance-rules.drl" />
  <Changeset path="." />
  <ContextSystem plugged="true" nonAnonymousEvents="http://www.fastfixproject.eu/ontologies/MonitoringOntology.owl#TextInput" />
  <UserReportingSystem plugged="true" post="false" />
  <CorrelationData path="eventCorrelation/collectedFields.csv"
    generalConf="eventCorrelation/generalFields.properties" rdfConf="eventCorrelation/dataStoreFields.properties" />
</EventCorrelator>
```

Figure 3.5: Event correlation configuration element

Pattern mining component Apart from the predefined patterns of error of FastFix, correlation system provides a pattern mining module to suggest unknown patterns. The process to mine them is made by two sub-processes: the learning and mining procedures. Pattern mining configuration is key for having good results. In this section, the meaning of each field is explained, and some tips to correctly configure them are explained in section 3.2.3.

- **EventStream**: Pattern mining procedures use context event streams as input data . This configuration element concerns the parameters of the set of context events to be analyzed by pattern mining module.
 - **eventTimeWindow** (long): It refers to the value, in milliseconds, of the time window of the event stream, i.e. if maintenance engineers team want analyze the context events occurred in last 4 hours, the value of this field would be 14400000 milliseconds.
 - **mac** (String): The MAC address of the user whose events would be analyzed by the module. If no MAC address is specified, all the events are taken into account.
 - **pluggedServerDataStore** (boolean): This field is used for development purposes. If its value is *true*, the pattern mining module will be fed with real

context events, provided by the server data store component. However, sometimes simulating the events is useful, specially for testing new pattern mining features, in this case, *false* is the appropriate value. The correct value for production environments is *true*.

- **Algorithm:** This configuration element refers to the parameters used by the learning and mining procedures.
 - **chosenAlgorithm** (String): It refers to the algorithm used in the learning phase. Possible values are “PrefixSpan” and “CSPADE”
 - **support** (long): The value of this field represents the minimum number of occurrences (in percent) of a sequence of context events to be considered as frequent.
 - **minimumSequenceSize** (int): It refers to the minimum number of context events needed to become a sequence.
 - **sequenceTimeWindow** (long): It refers to the value, in milliseconds, of the time window of a sequence.
 - **itemSetTimeWindow** (long): It refers to the value, in milliseconds, of the time window of a itemSet. A sequence consists of one or more item sets, so the value of *sequenceTimeWindow* must be higher than *itemSetTimeWindow*.
 - **converterOutput** (String): The stream of events provided by the server data store must be serialized using a concrete format, in order to be processed later by learning process. This field contains the path of the file used for this purpose.
 - **sessionOutput** (String): Once the events have been formatted and serialized, the context events are grouped into sequences. The result of this process is stored in a file, which will be read later by the chosen algorithm. This field contains the path of this file.
 - **preprocessedOutput** (String): This field means the same than *converterOutput*, but for the mining process.
 - **machineLearning** (String): It refers to the path where the executable jar for learning process is located. This jar contains PrefisSpan and CSPADE algorithms.
- **LaunchProcess:** Learning and mining process can be launched in two modes: on demand or using a task scheduler. This configuration element is used for this purpose.
 - **automatic** (boolean): If the value of this parameter is *true*, both procedures will be performed according to the cron expression specified in *learningSchedule* and *miningSchedule*. If the value is *false*, both process should be launched using the FastFix server interface, as it is detailed in section 3.2.3.
 - **learningSchedule** (String): This field contains the cron expression used to launch the learning process in automatic mode.
 - **miningSchedule** (String): This field contains the cron expression used to launch the mining process in automatic mode.

- Results

- `normalBehaviorPatterns` (String): It refers to the path of the file where the results of the learning process are stored, i.e. the sequences of events (patterns) which represents the normal behavior of the target application.
- `minedPatterns` (String): It refers to the path of the file where the results of the mining process are stored. i.e. the patterns of error identified. Maintenance team can view them using FastFix interface, as it's detail in section 3.2.3.

```

<!-- The PatternMining element has parameters to configure the pattern mining
components of FastFix. -->
<PatternMining id="fastfix_pattern_mining_configuration">
  <EventStream
    eventTimeWindow = "86400000"
    mac = ""
    pluggedServerDataStore="true"/>
  <Algorithm
    chosenAlgorithm= prefixpan
    support = "0.2"
    minimumSequenceSize= "2"
    sequenceTimeWindow = "3000"
    itemSetTimeWindow= "1500"
    converterOutput="patternMining/converterOutput.txt"
    sessionOutput="patternMining/sessionOutput.txt"
    preprocessedOutput="patternMining/preprocessedOutput.txt"
    machineLearning="patternMining/machineLearning/learningProcess_v1302191653.jar"/>
  <LaunchProcess
    automatic= "true"
    learningSchedule = "0 0 2 * * ?"
    miningSchedule = "0 0 5 * * ?"/>
  <Results
    normalBehaviorPatterns="patternMining/normalBehaviorPatterns/normalPatternBehavior.txt"
    minedPatterns="patternMining/minedPatterns/minedPatterns.owl"/>
/>

```

Figure 3.6: Pattern mining configuration element

- Known ontologies

Focusing on the ontology configuration, FastFix currently supports two ontologies, one representing the monitoring events and other representing all the concepts associated to software errors, the maintenance ontology. In order to properly configure the ontologies location, FastFix unified configuration uses the “**Ontologies**” element to describe the location and main properties of these ontologies:

```

<!-- The Ontologies element has parameters and locations for the ontologies
component in FastFix. -->
<Ontologies id="fastfix_ontologies_configuration">
  <!-- An Ontology is identified by its nsUri and is located in a path relative
to the FastFix home folder. -->
  <Ontology
    nsUri="http://www.fastfixproject.eu/ontologies/MonitoringOntology.owl"
    path="ontologies/MonitoringOntology.owl" useReasoner="true" />
  <Ontology
    nsUri="http://www.fastfixproject.eu/ontologies/MaintenanceOntology.owl"
    path="ontologies/MaintenanceOntology.owl" useReasoner="false"
    modelSpec="OWL_DL_MEM" />
</Ontologies>

```

Figure 3.7: Ontology configuration element

3.1.1.3 Error Reporting and Fault Replication

Communication Configuring the FastFix communication subsystem is critical to a correct operation of FastFix. This component manages all connections between FastFix clients and servers. Figure 3.8 highlights the element in the FastFix XML configuration which control the communication system. The server's **host** and **port** must be the same on the client FastFix configuration and on the server's. The port choice depends only on the server's administrative restrictions. The client **host** and **port** are only used on the FastFix client and are freely chosen barring any administrative restrictions. Finally, SSL can be used for securing the communication channel between client and server. To use SSL, the **active** property must be set to *true* (**keypath** and **keyvalue** are unused). The remaining properties configure the usual SSL parameters. For the FastFix server, the **trustStore** and **trustStorePassword** configure the location and password to access the list of the trusted clients. And, for the client, the **keystore** and **keyStorePassword** configure the location and password to access the file with the client's own SSL certificate. Obviously, each FastFix client must have its certificate added to the keystore file and the FastFix server must add each client's certificate to its truststore file.

```

<!-- The Communication element deals with general communication channels
for communication between client and server. -->
<Communication id="fastfix_communication_configuration">
  <!-- Server location, by host and port -->
  <Server host="localhost" port="5555" />
  <!-- Client location, by host and port -->
  <Client host="localhost" port="5556" />
  <!-- Secure connection active or not, with parameters to configure the
secure channel in java -->
  <SSL active="false" keyPath="communication/ssl/keys/" keyValue="ABase64EncodedStringAsTheKey"
    trustStore="communication/sslStore" keyStore="communication/sslStore"
    trustStorePassword="123456" keyStorePassword="123456" />
</Communication>

```

Figure 3.8: Communication configuration element

Issue Tracker FastFix uses an issue tracker to store any error reports that it generates. Therefore, the server uses the FastFix configuration to configure the connection to the

issue tracker. Figure 3.9 highlights the element in the FastFix XML configuration which configures the issues tracker connection on the FastFix server. The `Ticket` element has the `attachmentMaxSize` parameter to limit the size of error report attachments. The `TracConnector` element configures the remaining parts of the issue tracker configuration (current parameters are used for the TRAC issue tracker but could be used with alternative meanings with other issue trackers):

- The `trac_server_main_uri`, `trac_login_uri` and `new_ticket_uri` parameters reference the URIs of the three issue tracker webpages, respectively, the entry page, the login page and the ticket insertion page.
- The `username` and `password` are the credentials to login into the issue tracker.
- The `trac_get_login` parameters informs FastFix whether the issue tracker login page requires an HTTP GET or PUT call.
- The `useEmbeddedTruststoreFile` identifies if set to `false` that an external file is used as a truststore to obtain the certificate of the issue tracker server. The `truststore` parameter points to the location of the truststore, the `trac_authorization_domain` identifies the domain of the issue tracker certificate and the `trac_truststore_password` is the password to open the truststore.
- Finally, the `TicketBrowser` element has as its only parameter the `TicketURLPrefix` which is the URL used to access the page describing a particular ticket in the issue tracker. It assumes that if a ticket number is appended to this string, this will result in a valid page describing that ticket.

```
<!-- The issue tracker has parameters to configure the issue tracker to
which the FastFix server can connect. -->
<IssueTracker id="fastfix_issuetracker_configuration">
  <!-- The Ticket element has parameters to configure several aspects of
the tickets for the target issue tracker. -->
  <Ticket attachmentMaxSize="2048" />
  <TracConnector trac_server_main_uri = "https://traclabs.fastfixproject.eu"
trac_login_uri="https://traclabs.fastfixproject.eu/login"
new_ticket_uri="https://traclabs.fastfixproject.eu/newticket"
username="████████████████████"
password="██████"
trac_get_login="true"
useEmbeddedTruststoreFile="true"
truststore="/resources/tracConnector.truststore"
trac_authorization_domain="traclabs.fastfixproject.eu"
trac_truststore_password="FastFix"/>
  <TicketBrowser TicketURLPrefix="https://traclabs.fastfixproject.eu/ticket/" />
</IssueTracker>
```

Figure 3.9: Issue tracker configuration element

Fault Replication The last configuration in terms of fault replication is the configuration of the fault replication component contained in the FastFix configuration file as well and shown below in Figure 3.10.

The `LogStore` element configures (for the FastFix client) the connection to the machine where any auxiliary logs of the error reports are to be stored. This is the machine where error reports are to be replayed, normally the FastFix Server. This connection is defined by the `host` and `port` of the machine. In order to store the logs, a username (`user`), `password` and `location` for storage of these files is provided.

The `TargetApplications` element configures needed at the client about the applications using FastFix fault replication. Its parameters are:

- `AwtInstrumentedClassesSubdir`: The folder containing the instrumented versions of AWT applications used with fault replication at this client.
- `ModelSubdir`: The folder containing the GUI model description of AWT applications used with fault replication at this client.
- `InstructionsFile`: The instructions file.
- `EnableAWTAnonymizer`: Indicates whether GUI anonymization should be used. If set to true, the error reports of AWT applications are reduced to a minimum.

Each sub-element of `TargetApplications` represents an application using FastFix (in the sample file in the image, `Moskitt`, `myJPass` and `Robot`). This part of the configuration is used by the server. Each application has a version and a type, either SWT or AWT. If it's SWT, the `location` parameter indicates where, at the FastFix server, the code for that particular version of that application is located. If it is AWT, the `mainClassPath` contains the path to the main class of the application and the `mainClassName` contains the name of the application's main class' name.

The `SwtRecorder` element is used to contain the `SwtRecordedEventsPath` which points to the location at the client where events logged by FastFix-enabled SWT applications can be logged. The `AwtRecorder` element is used to contain the `AwtRecordedEventsPath` which points to the location at the client where events logged by FastFix-enabled AWT applications can be logged. The `FilePrefix` element designates the prefix used to name these logs.

Finally, the `Reap` configuration element configures the anonymization of user input text: `reap_dir` points to the location, at the client, of the anonymization bundle `eu.fastfix.targetapplication.sensor.reap` and `workspace_dir` should point to a folder on the client that can be used by the anonymizer to store temporary files.

```
<!-- The FaultReplication element has parameters for the fault replication
component of FastFix -->
<FaultReplication id="fastfix_fault_replication_configuration">
  <!-- The LogStore element has parameters to copy the logs to a remote server. -->
  <LogStore host="test.domain.com" port="22" user="fastfix_replication"
    password="IHaveNone" location="/fault_replication/logs" />
  <!-- The TargetApplications section provides information about Applications which are used to replay faults. -->
  <TargetApplications
    AwtInstrumentedClassesSubdir="InstrumentResult/"
    ModelSubdir= "structure/Model.txt"
    InstructionsFile="/Users/fastFIX/targetApp.txt"
    EnableAwtAnonymizer="true">
    <Moskitt version="version" type= "SWT" location="/pathToMoskitt"/>
    <myJpass version="1.0" type="AWT" mainClassPath="pathToMyJpass" mainClassName= "MainClassName"/>
    <Robot version="1.0" type="AWT" mainClassPath="pathToRobot" mainClassName= "MainClassName"/>
  </TargetApplications>
  <SwtRecorder SwtRecordedEventsPath="/Users/fastFIX/SwtRecorded/guievents"/>
  <AwtRecorder AwtRecordedEventsPath="/Users/fastFIX/AwtRecorded/" FilePrefix="ListenersSequence"/>
  <Reap reap_dir="/vorkspace_fastFIX/eu.fastfix.targetapplication.sensor.reap"
    workspace_dir="/vorkspace_fastFIX">
    <LinkProperties target_project="Robot" properties_file="reap.properties"/>
  </Reap>
</FaultReplication>
```

Figure 3.10: Ticket browser configuration element

3.1.1.4 Patch Generation and Self-Healing

Several configuration parameters can help tune the Patch Generation and Self-Healing component. These parameters are illustrated in Figure 3.11 for the server side, as they appear in the FastFix maintenance environment extension for Eclipse. The *Parameters and Attributes* option decides on whether models should take method parameters and attributes into account. When not selected, the supervisor is less accurate but more efficient at runtime. This option can be useful whenever the runtime overhead is expected to be high. *Automatic Entry Point* and *ExpertFile Entry Point* both relate to the selection of the methods to be instrumented and for which models are extracted. The automated option computes entry points automatically, performing static analysis of the source code. The ExpertFile option allows developers to describe relevant methods to be instrumented. Expert files are xml files as presented in Figure 3.12.

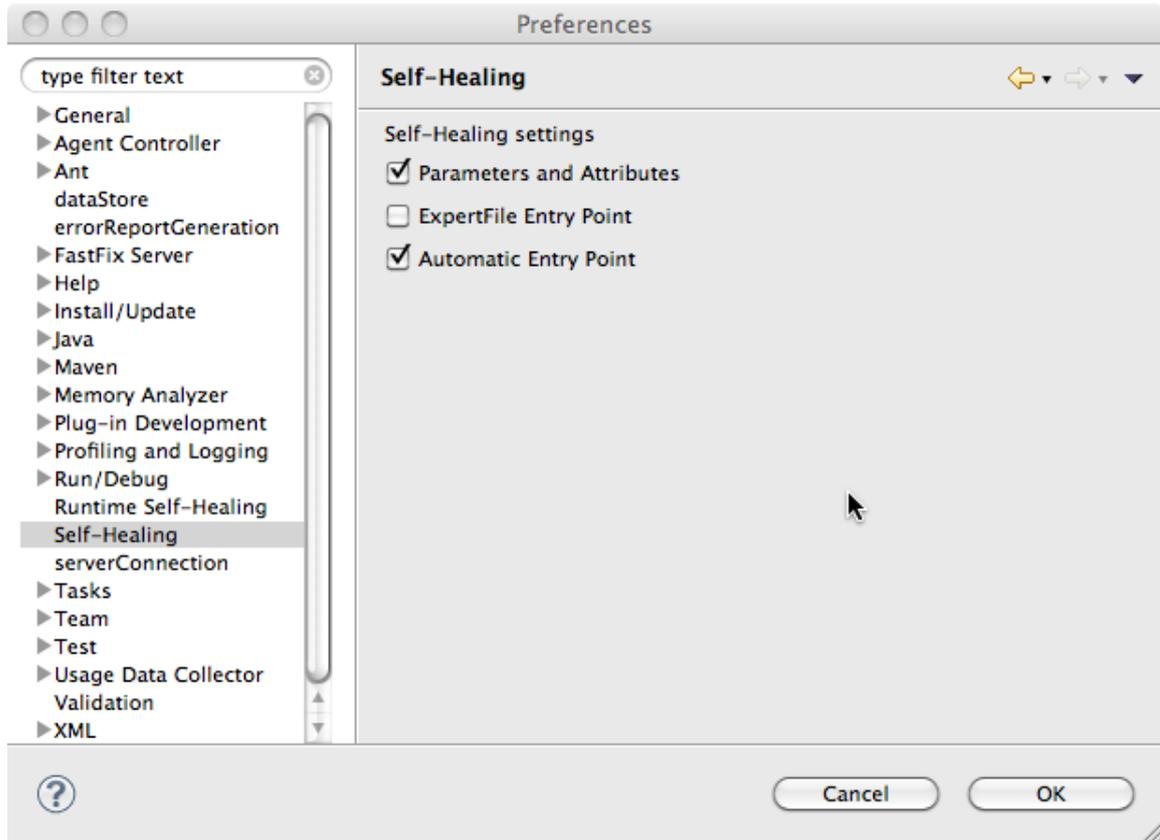


Figure 3.11: FastFixSH configuration attributes

Expert files describe sets of method declarations. Method name, declaring class name, declaring parent class name and declaring package name is the information that can be used in the file. Each set of methods is described within the `<method></method>` tag. `<methodName></methodName>` is a sub-tag that can contain a regular expression that relevant methods name must fulfill. `<class></class>` tag contains information about the declaring class of the relevant methods. It is possible to enter a regular expression in the `<className></className>` tag in order to filter the declaring classes according to their names.

```

<method>
  <!-- Specify the name of the Method 'methodA' -->
  <methodName>actionPerformed</methodName>
  <!-- Specify the information of the Class initiated the 'methodA' -->
  <class>
    <className/>
    <parentClassName>ActionListener</parentClassName>
    <packageName/>
  </class>
</method>
<!-- The method that you want to instrument as EntryPoints,
before extracting FastFix models -->
<method>
  <!-- Specify the name of the Method 'methodA' -->
  <methodName>run</methodName>
  <controllable>true</controllable>
  <!-- Specify the information of the Class initiated the 'methodA' -->
  <class>
    <className/>
    <parentClassName>Thread</parentClassName>
    <packageName/>
  </class>
</method>
<method>
  <!-- Specify the name of the Method 'methodA' -->
  <methodName>newFilter</methodName>
  <controllable>>false</controllable>
  <!-- Specify the information of the Class initiated the 'methodA' -->
  <class>
    <className>ControlWindow</className>
    <parentClassName/>
    <packageName>gui</packageName>
  </class>
</method>
</expertFile>

```

Figure 3.12: Sample Expert File

It is also possible to use the `<parentClassName></parentClassName>` to enter a regular expression that one of the parent classes or interfaces name must fulfill. Finally the `<packageName></packageName>` tag contains a regular expression that the package in which the method is declared must fulfill. All these tags work in conjunction, i.e. only methods that fulfill all the conditions are selected. However if tag values are left empty, then they have no impact in the filtering process. This is useful for instance for methods declared in anonymous class. In this case the `<className></className>` tag should remain empty. Finally, several sets of methods can be described in an expert file, using several `<method></method>` tags.

Finally, configuration parameters are also available on the client side and can be set through the FastFix client (Figure 3.13). *Activate Controller* allows to disable monitoring for self-healing, and consequently any action that this component could make. This is only useful in case runtime overhead becomes an issue, in order to disable any of the behavior brought through code instrumentation. *Supervisor Buffer Size* represents how

many method calls are kept in memory at most. Therefore it also represents the size of the trace logged when an exception is raised and caught by the self-healing component. *Request Period* represents the frequency at which the FastFix client sends a request to the server for new patches. Finally, *Testing Mode for Trace Collection* modifies the behaviors of the self-healing component at runtime. In this mode, every method calls that is instrumented is logged in a file. This is useful to collect sample of traces. This traces can for example be generate when the application is under test and therefore represent a set of good behaviors of the system (corresponding to traces collected from passing tests). These traces can later be used for patch validation.

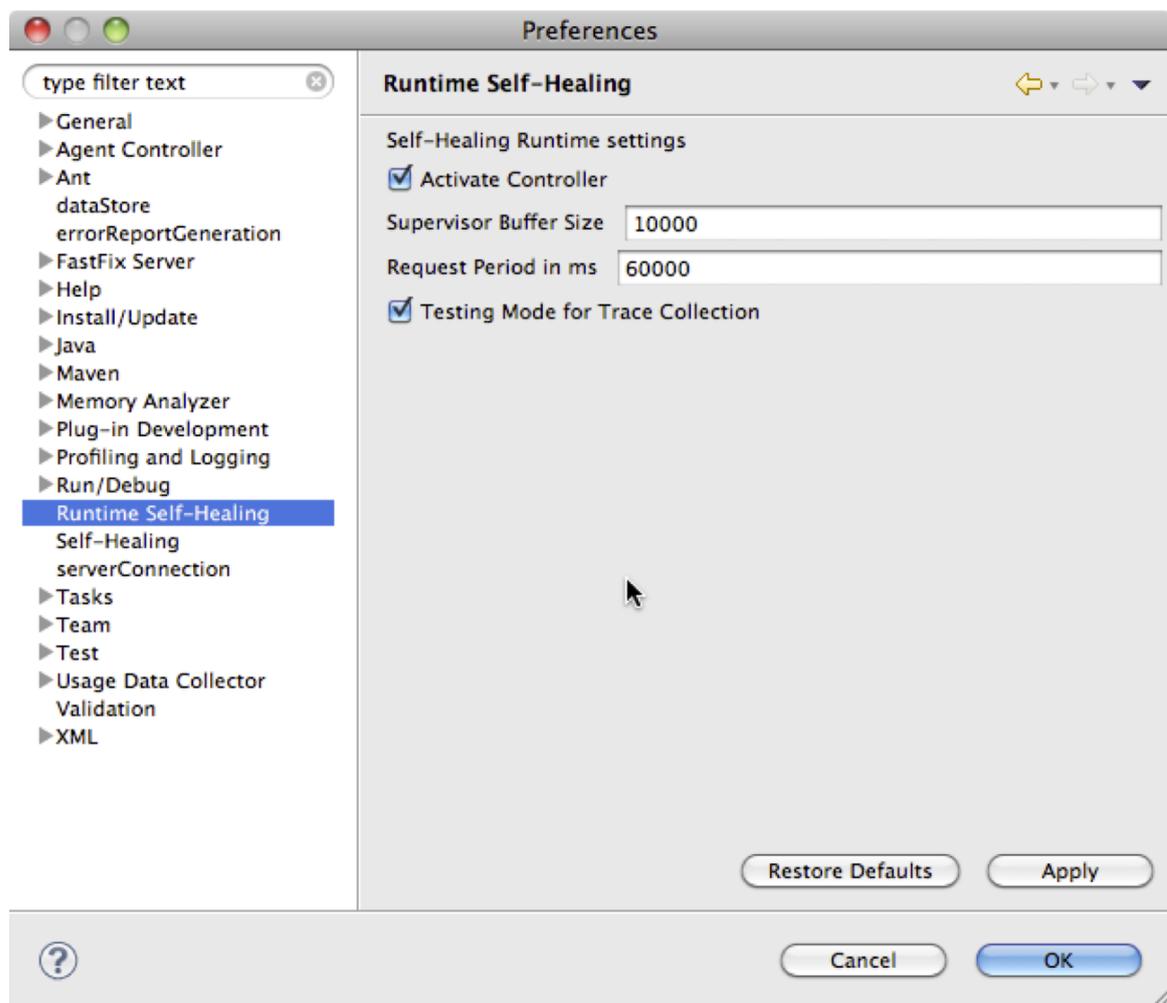


Figure 3.13: FastFixSH configuration within FastFix Client.

3.1.2 FastFix Client Setup

The FastFix client is implemented as an Eclipse RCP (Java) application. It can also be run in console mode for systems where no graphical user interface is available. The latest binary version of the FastFix client can be accessed in the FastFix repository at SourceForge ⁵. To run the FastFix client, the following steps have to be accomplished:

⁵<https://sourceforge.net/projects/fastfixrsm/files/client/>

1. Download the latest build of the FastFix client and unzip into a folder on the *client machine*.
2. Configure the FastFix client (see Section 3.1.1).
3. Start the client (usually by double-clicking).

The FastFix client is now ready and waits for sensors to register⁶⁷. As soon as sensors register at the FastFix client, they show up in the client UI. Sensors are started automatically once registered, unless the FastFix client is not configured to a manual start mode. In the latter case, sensors can be started from the client UI, as illustrated in Figure 3.14.

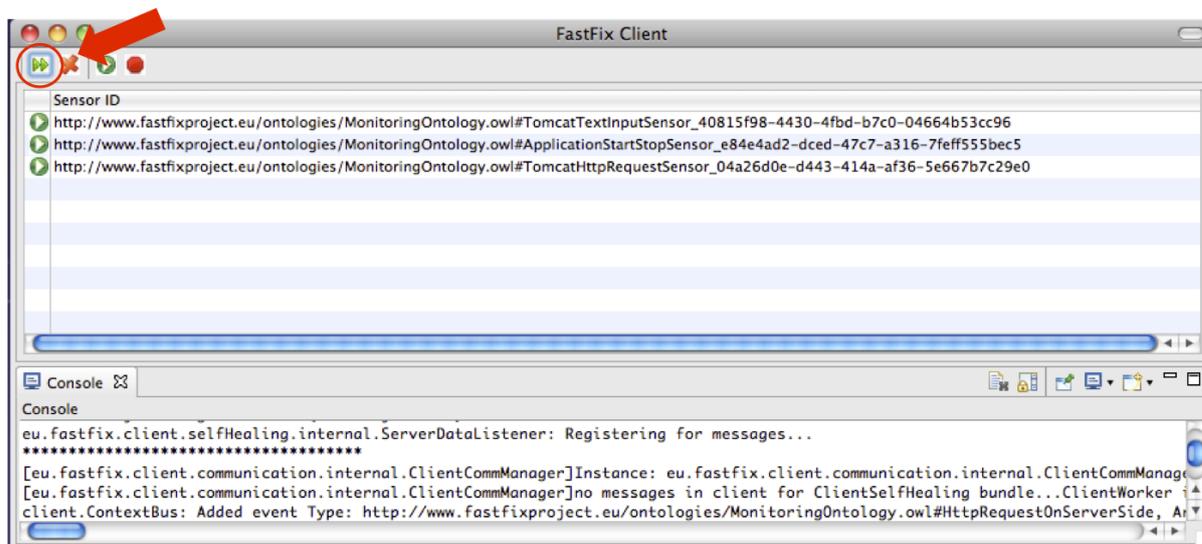


Figure 3.14: Starting registered sensors via the FastFix client UI.

3.1.3 FastFix Server Setup

The FastFix server is implemented as an Eclipse RCP (Java) application, similar to the FastFix client. It can also be run in console mode for systems where no graphical user interface is available. The latest binary version of the FastFix server can be accessed in the FastFix repository at SourceForge⁸. To run the FastFix server, the following steps have to be accomplished:

1. Download the latest build of the FastFix server and unzip into a folder on the *server machine*.
2. Configure the FastFix server (see Section 3.1.1).
3. Start the server (usually by double-clicking).

After the last step, the FastFix server is ready and waits to receive information from FastFix clients.

⁶For details on the sensor lifecycle, we refer the reader to [11]

⁷For more details on current FastFix sensors, we refer the reader to [12]

⁸<https://sourceforge.net/projects/fastfixrsm/files/server/>

3.1.4 FastFix Sensors Setup

FastFix is an open source project and as such provides a generic, extensible maintenance platform. The extensibility of the context observation system is a main enabler for the applicability of FastFix in different scenarios. To this end, the FastFix platform is designed to work with arbitrary sensors, as long as these sensors implement the FastFix sensor protocol called “sensor lifecycle” [12]. The setup procedures for sensors depend on the particular sensor at hand. In general, sensors are first installed, and then register at the FastFix client. From there they can be controlled via the user interface as described in 3.1.2.

During the FastFix project, several sensors were developed. The general configuration applicable to all sensors is described above in Section 3.1.1.1. The setup of sensors developed during FastFix is described in detail in this section.

3.1.4.1 Eclipse RCP Sensor Setup

Technology and Framework Dependencies The FastFix RCP sensor⁹ monitors user actions and exceptions of RCP applications. Hence, it can be used for all applications based on Eclipse and RCP. In the following, we use MOSkitt as an exemplary Eclipse RCP application.

Requirements The following is required to deploy the FastFix RCP sensor on a machine:

- Java 1.5 installed
- MOSKitt installed

Sensor Installation The following steps are necessary to install the RCP sensor. Overall, the RCP sensor is installed using the RCP/ Eclipse update mechanism.

1. Download the update site of the latest build of the RCP sensor.
2. Unzip the downloaded file into a directory “moskittsensors”.
3. Start MOSKitt.
4. Navigate to “Help -> Install new Software” in the main menu. Make the the downloaded update site known by clicking on “Add...”, then clicking on “Local...”, and selecting the “moskittsensors” folder. Specify “FastFix MOSKitt Sensor” as name of the update site and acknowledge with “Ok”.
5. Select the new entry “FastFix Moskitt Sensor” in the list of potential software updates and click Next and Finish, accepting the license agreement.
6. Close MOSKitt.
7. Configure the application bridge and the general sensor settings (see Section 3.1.1.1).

The RCP sensor is now installed. If sensors are configured to start automatically upon application startup, it will start when MOSKitt is started. Otherwise the sensor has to be started manually via the FastFix menu added to the MOSKitt main menu.

⁹Formerly called “MOSKitt sensor”

3.1.4.2 Struts Sensor Setup

Technology and Framework Dependencies The FastFix Struts sensor monitors HTTP requests and corresponding actions defined by Struts web application framework. Hence, it can be used for all applications hosted in Tomcat that are using Struts. In the following, we use Espigon as an exemplary web application based on Struts and hosted in Tomcat.

Requirements The following is required to deploy the FastFix Tomcat sensor on a machine:

- Local Tomcat server instance with Espigon web application installed
- PostgreSQL Server 8.1 or higher installed
- Java 1.5 installed

Sensor Installation The following steps are necessary to install the FastFix Struts sensor. The Struts sensor is installed by copying downloaded libraries into certain /lib folders and modifying configuration files of a Tomcat web application.

1. Download the latest build of the Struts sensor.
2. Unzip the downloaded file into a directory “espigonsensors”
3. Stop Tomcat
4. Copy the following jars from “espigonsensors” in the “webapps/EspigonValencia/WEB-INF/lib” folder of the Espigon web application:
eu.fastfix.targetapplication.sensor.tomcat.struts*.jar
eu.fastfix.common.applicationbrige*.jar
eu.fastfix.common.configuration*.jar
eu.fastfix.common.logging*.jar
flexjson-2.1.jar
5. Modify the ‘webapps/EspigonValencia/WEB-INF/struts-config-valencia.xml’ file:
In a clean Espigon install it contains the value:
`<controller processorClass="org.apache.struts.tiles.TilesRequestProcessor"/>`
That value has to be changed to:
`<!-- FastFix Struts sensor hook --> <controller processorClass="eu.fastfix. targetapplication.sensor.tomcat.struts.processor.FastFixTilesProcessor"/>`
6. Modify the ‘webapps/EspigonValencia/WEB-INF/web.xml’ file:
Add the following tag within the `<web-app></web-app>` root XML element:
`<!-- FastFix StartStop Listener Definition --> <listener> <listener-class>eu.fastfix.targetapplication.sensor.tomcat.struts.processor.ApplicationStartStopListener </listener-class> </listener>`
7. Configure the application bridge and the general sensor settings (see Section 3.1.1.1).

The Struts sensor is now installed. It will be activated the next time Tomcat server is started and if sensors are configure to start automatically upon application startup. Please note that there is no way to start the Struts sensor manually.



3.1.4.3 Log Sensor

Technology and Framework Dependencies The FastFix Log sensor monitors log files and new log entries appended to them. Hence, it can be used for all applications writing log files. It can be directly used for applications writing log entries using the same pattern as PostgreSQL or MySQL error messages and has to be adapted for other log entry patterns. In the following, we use Espigon as an exemplary application writing information about errors to a log file.

Requirements The following is required to deploy the FastFix Log sensor on a machine:

- Java 1.5 (Runtime Environment) installed

Sensor Installation The following steps are necessary to install the FastFix Log sensor. The the FastFix Log sensor is installed by copying the binary and running it from console.

1. Download the latest build of the Log sensor.
2. Move the downloaded jar into the “espigon/tomcat/logs” folder of the Espigon application hosted within a Tomcat server.
3. Configure the application bridge and the general sensor settings (see Section 3.1.1.1).

The Log sensor is now installed. Perform the following steps to run it:

1. Open the folder “espigon/tomcat/logs” in a console window.
2. Run the Log sensor by entering the command “java -jar logsensor.jar -f catalina.out -p PostgreSQL” from this console.
3. Do not close the console window.

The Log sensor is now running.

3.1.4.4 WCF Sensor Setup

The WCF sensor¹⁰ monitors the content of WCF messages. Hence, it can be used for all applications developed in .NET that are using WCF. In the following, we use TXTExecute as an example of a .NET application whose WCF messages are monitored.

Requirements The following is required to deploy the WCF sensor on a machine:

- Windows operating system

¹⁰Formerly called “TXT Execute sensor”

Sensor Installation The following steps are necessary to install the WCF sensor.

1. Download the latest microsoft installation file (.msi extension) of the WCF sensor.
2. Run the installer by double clicking on the file and choose the desired location for the installation of the sensor.
3. Open the IIS Manager and select the “ExecuteService” site. Then, on the right side, double click “Modules”.
4. Click on “Add Managed Module” on the upper right, and insert the following into the resulting dialog box:
 - a) Name: **FastFixHttpModule**
 - b) Type: **FastFix.FastFixHttpModule, FastFixHttpModule, Version=1.0.0.0, Culture=neutral, PublicKeyToken=6fead1b590d37761**

Sensor Configuration In order for the sensor to run with the start of the TXT client application. The Web.config file from the TXT service needs too be modified. To do so, the following steps are necessary:

1. Open the execute_service folder (where the ExecuteService is deployed), and add the following under the <configuration> tag of the Web.config file:

```
<appSettings>
<add key="sensorBridgeURL" value="http://localhost:9999" />
<add key="callTimeoutMS" value="3000" />
<add key="filterURL" value="/service" />
<add key="logEnabled" value="true" />
<add key="logfile" value="fastfixsensor.txt" />
</appSettings>
```

2. Save and close the file the Web.config file.
3. Restart the ExecuteService Web Site in the IIS Manager.

3.2 Platform Usage

The following sections describe typical use cases of the FastFix platform features in detail, specifying current requirements and configuration possibilities.

3.2.1 Context Observation

The main use case of FastFix sensors is context monitoring, i.e. collecting data that is used by other components such as event correlation, fault replication and self-healing. However, manual interaction by the maintenance engineer is required only for installing and configuring sensors. How to setup FastFix sensors is described in Section 3.1.4.

3.2.2 Event Correlation

The pattern of an error is the key piece of the FastFix event correlation system. So, the use cases for maintenance engineers are related with its management: create and edit patterns of error. Each member of the maintenance team can access both features through the FastFix Server menu.



Figure 3.15: Event Correlation Menu

3.2.2.1 Create a pattern of error

FastFix includes several error patterns by default, but the maintenance team can add more, based on their expertise in the target application. The steps to create a pattern of error are the following:

1. Access to the creation pattern interface, selecting the menu option *Pattern Management / Event Correlation / Add new error pattern*. This interface consists of three tabs: *General Information*, *Symptoms* and *Cause*.
2. Fill general information fields. All of them are mandatory (Figure 3.16).
 - Pattern Name: Represents the name of the pattern. It will be used to identify the pattern.
 - Subject: This field represents the subject of the report associated with this new pattern. This report will be inserted in issue tracker (Jira, Eventum or Trac) when current pattern of error occurs.
 - Summary: In the same way that *Subject*, this field must content the summary of the report associated with this new pattern.
3. Add symptoms. This step is the most important, since the symptoms contains the conditions to be matched to detect a pattern. For each symptom, some fields must be inserted. The following are the most important but all of them are explained in detail in D4.6. (Figure 3.17)
 - Event Type: This field represents the context event type associated with a symptom. To choose one, a selection dialog is provided. Mandatory.
 - Symptom Name: It represents the name of the symptom. It will be used to identify the symptom, hence, it must be unique. Mandatory.

- Operator: This field is a combo-box which represents the action to be performed over the next field *Value*, to detect if an event, with a concrete event type, is a symptom of a pattern. Possible values are: *matches*, *<*, *>*, *=*, *other*. Optional, but mandatory, if *Value* field has been inserted.
 - Value: This field represents the metadata associated to the symptom, that will be evaluated by using the last field *Operator*. Optional, but mandatory if *Operator* field has been inserted.
 - Criterion: This field is a combo-box which represents if it is necessary or not that all the declared symptoms must occur to detect a pattern. Possible values are: *all* and *any*. Mandatory.
4. Identify the cause. In this tab maintenance team can associate a cause to the current pattern. Associate a cause to a pattern it is not necessary, but all the fields are mandatory in such case. (Figure 3.18)
 - Cause Type: It represents the type of this cause. To choose one, a selection dialog is provided.
 - Cause Name: This field represents the name of the cause. It will be used to identify the cause so it must be unique.
 - Info: The information inserted in this field must explain in a general way, what this cause means.
 5. Save the new pattern. If any mandatory field is empty, a message is shown warning the engineer that must review the tabs and complete all the required information.

Once a pattern is created, it's added automatically to the FastFix correlation system, i.e. it is not necessary to restart the FastFix server to detect this new error.

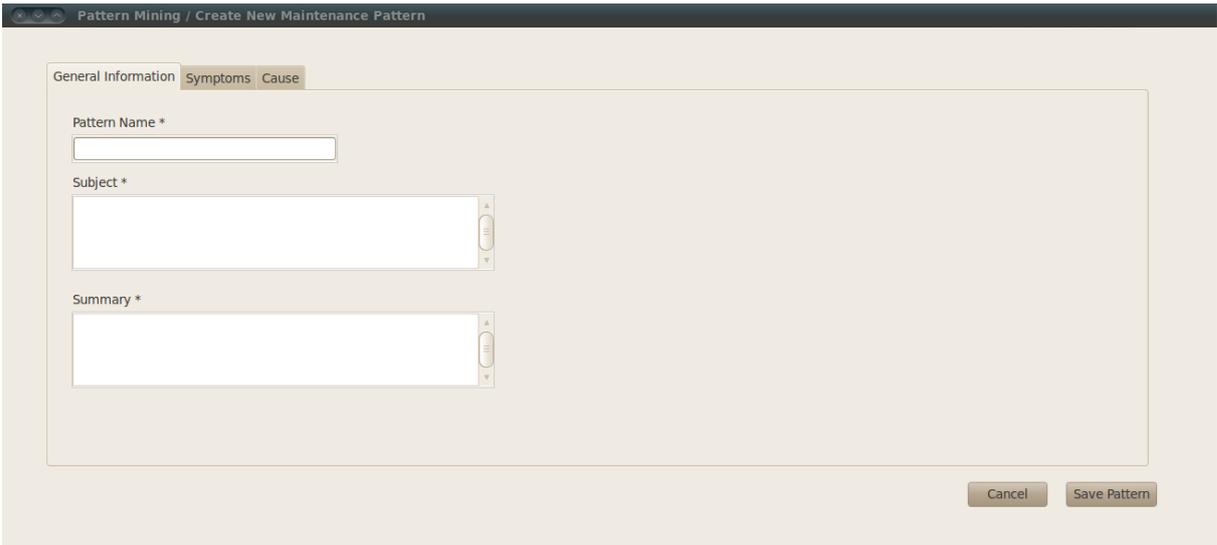


Figure 3.16: General Information Tab

| Event Type * | Symptom * | Operator | Value | Info | Reaction | Delay | Reuse |
|--------------|--------------------|----------|-------|------|----------|-------|-------|
| PressButton | symptomPressButton | matches | save | | | | |

If you choose an operator, you have to fill the field "Value", since the operator is used to evaluate it.
 In the same way, if you fill the field "Value", you have to choose an operator.
 If you fill the field "Reaction", you have to fill the field "Delay", since reaction event should occur at most as many seconds as specified in "Delay" field.
 In the same way, if you fill the field "Delay", you have to fill "Reaction" field.

Criterion *

Figure 3.17: Symptoms Tab

[Choose an existing cause](#) or create a new one

Create new Cause

Cause Type *

Cause Name *

Info *

Figure 3.18: Cause Tab

3.2.2.2 View and edit pattern of error

Using this feature, each member of the maintenance team can view all the existing patterns and edit them. The steps to view and edit a pattern are the following:

1. Access to the edition pattern interface, selecting the menu option *Pattern Management / Event Correlation / View error pattern* (Figure 3.15).
2. Select one error pattern from the list, and press "Edit" (Figure 3.19). The resulting interface is the same as the last use case, with the information of the selected pattern.
3. Edit general information (the name of the pattern is not editable).

4. Edit symptoms (add and delete operations are also supported)
5. Edit cause information.
6. Save changes. If any mandatory field is empty, a message is shown noticing the engineer that must review the tabs and complete all the required information.

Once a pattern is edited, changes are automatically applied in the FastFix correlation system, i.e. restarting the FastFix server is not needed.

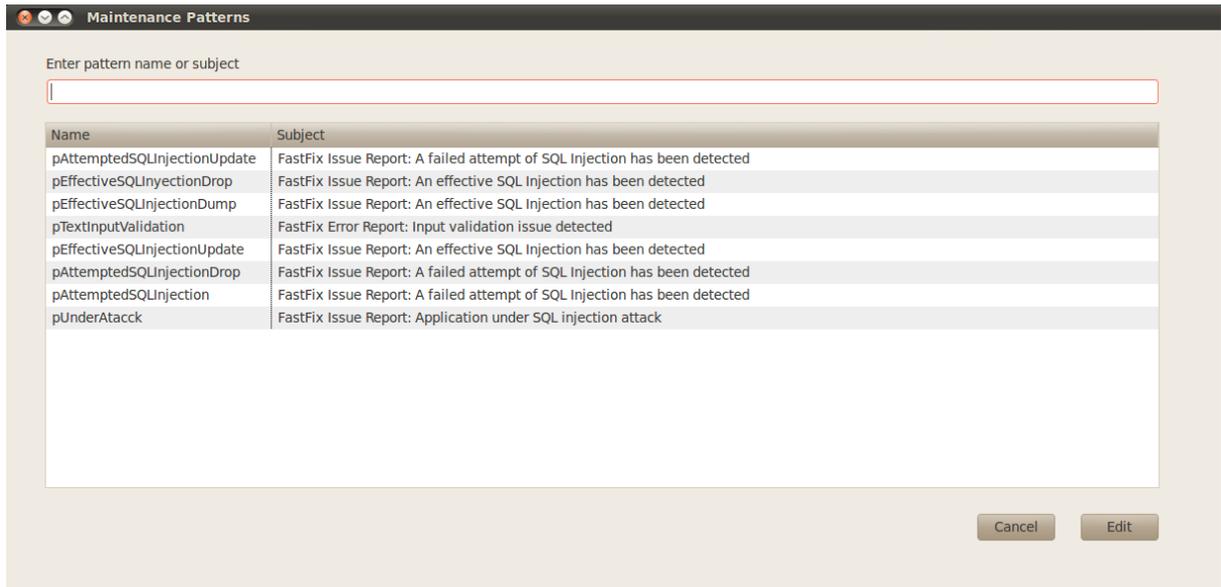


Figure 3.19: List of existing patterns of error

3.2.3 Pattern Mining

Apart from the predefined error patterns and ones added by the maintenance team, FastFix correlation system provides a pattern mining module to suggest unknown patterns. The process to mine them consists of sub-processes: the learning and mining procedures and can be launched in two modes: on demand or automatically, i.e. using a cron expression. To use this feature automatically, see section 3.1.1.2

First two use cases are related with these sub-processes, in case on demand launching. The third one is validate the discovered pattern, i.e. once a pattern is discovered, a member of the maintenance team must validate it in order to add it to the correlation module. By the other hand, if engineers consider the new pattern as a false positive, they can discard it. As others FastFix correlation system use cases, maintenance team can access them through the FastFix Server menu.



Figure 3.20: Pattern Mining menu

Learn application behavior patterns

Not many steps are required for launching the learning process, just select the menu option *Pattern Management / Pattern Mining / Launch learning process*. The purpose of this process is to detect sequences of events that represents the normal behavior of the target application. This sequences or patterns are stored in a file, which is specified in configuration.xml file (see section 3.1.1.2). If no normal behavior patterns are detected after launching the process with default configuration values, the maintenance team can tune some parameter values, like the time window of the sequence and the support. The smaller is the time window of a sequence (or *support* parameter value), the big the number of sequences that will be detected. However, the number of false positives could be also higher.

Mine new error patterns

Launch mining process is easy, just select the menu option *Pattern Management / Pattern Mining / Launch mining process*. This procedure will analyze a stream of events and will compare them against the normal behavior patterns obtained in the last process. The maintenance team can view the mined patterns using the menu option *Pattern Management / Pattern Mining / New mined error patterns*. A dialog will present the list of new patterns (Figure 3.22), and engineers can look at the details of the pattern. An example is illustrated in Figure 3.21



| Context Event Type | Date | Value | Info |
|----------------------|------------------------|--------------------------------|------------------|
| TextInput | Sat Jul 31 00:32:03 CE | 3 | Possible anomaly |
| PressButton | Fri Dec 15 03:05:23 CE | Listado de transeuntes | |
| HttpRequestOnServers | Fri Dec 15 03:05:23 CE | /espigonvalencia.war/perfil.do | |
| DbStatementEvent | Fri Dec 15 03:05:23 CE | Select * from cod_pue where s | |
| ApplicationEvent | Sat Mar 14 19:58:43 C | /espigonvalencia.war/perfil.do | |

Figure 3.21: Detail of a mined pattern

Discard error patterns

Maintenance team users can validate or discard mined patterns. Both features are accessible in the interface where the mined patterns are listed via buttons. In case that the maintenance team member considers that the pattern is a false positive or a duplicate, they can discard the pattern using the button *Discard Pattern*. Once a pattern is discarded, it is automatically deleted in the FastFix correlation system, so it is not necessary to restart the FastFix server.



| Pattern | Date | Status |
|-------------|------------------------------|--------|
| p3636473612 | Mon Dec 10 12:49:09 CET 2012 | New |

Figure 3.22: Dialog showing a new mined pattern

Validate error patterns

If the engineer consider the mined pattern is not a false positive, he can validate it. The process to validate a pattern is the same than creating a new one (see section 3.2.2.1).

The difference is that some information about the symptoms are pre-inserted. Once a mined pattern is validated it is automatically inserted in the FastFix correlation system, so it is not necessary to restart the FastFix server.

| Event Type * | Symptom * | Operator | Value | Info | Reaction | Delay | Reuse Symptom |
|-------------------------|-----------|----------|---------------------------|------------------|----------|-------|---------------|
| ApplicationEvent | s02 | | /espigonvalencia.war/perf | | | | |
| DbStatementEvent | s28 | | Select * from cod_pue wh | | | | |
| HttpRequestOnServerSide | s20 | | /espigonvalencia.war/perf | | | | |
| PressButton | s36 | | Listado de transeuntes | | | | |
| TextInput | s45 | | 3 | Possible anomaly | | | |

If you choose an operator, you have to fill the field "Value", since the operator is used to evaluate it.
 In the same way, if you fill the field "Value", you have to choose an operator.
 If you fill the field "Reaction", you have to fill the field "Delay", since reaction event should occur at most as many seconds as specified in "Delay" field.
 In the same way, if you fill the field "Delay", you have to fill "Reaction" field.

Criterion *

Delete Symptom Add Symptom

Cancel Save Pattern

Figure 3.23: Validating error patterns

3.2.4 Error Reporting

Error reporting in FastFix is triggered by FastFix sensors that trigger the error reporting mechanism. Use of FastFix error reporting is mainly an issue of connecting the correct sensors to the FastFix-enabled application as described above in section 3.1.4. If the installed sensors detect a fault, they will request an error report which is sent to the FastFix server and inserted into an issue tracker.

Once the error report is inserted in the issue tracker, it can be viewed by the maintenance engineers. The mode of presentation of a particular error report depends on the specific issue tracker being used. More commonly, the TRAC issue tracker has been used in FastFix. An example of the display of an error can be seen in Figure 3.24. If fault replication is being used, the error report number can be used for fault replay (see next section).

The screenshot shows a web browser window with the URL `https://trac.fastfixproject.eu/ticket/1254`. The page is titled "Ticket #1254 (new defect)". The main content is a "FastFix Issue Report" with the following details:

- Reported by:** automatic-reporting
- Owned by:** szamarripa
- Priority:** minor
- Component:** component1
- Keywords:**
- Description:** A `NullPointerException?` has been detected while trying to select an element of a dialog. This error occurs when the same dialog is opened several times and the element is selected from the item list of the first opened dialog.
- Symptoms:**
 - PRESS BUTTON: open**
User "jdoe" on client IP Address: "172.18.232.49" clicked on a button, with label: "open"
 - PRESS BUTTON: open**
User "jdoe" on client IP Address: "172.18.232.49"
 - WIDGET SELECTED**
User "jdoe" selected a widget
 - PRESS BUTTON: ok**
User "jdoe" on client IP Address: "172.18.232.49"
 - EXCEPTION:** `NullPointerException?`
- Possible cause:** Loss of object's reference. The reference to an object has been lost, due to multiple creation of other instances.

Below the description, there is an "Attachments" section with an "Attach file" button, and an "Add a comment" section with a rich text editor and a "View" link.

Figure 3.24: Viewing an error report

3.2.5 Fault Replication

Fault replication, the ability to see the replay of a client fault at a maintenance server, is available for Java console, AWT and SWT applications. Configuring fault replication requires preparing the application (instrumentation in most cases) and configuring the FastFix client and server.

3.2.5.1 Preparing Applications: Instrumentation

3.2.5.1.1 SWT Applications Setting up SWT applications for FastFix error record/replay requires no previous instrumentation. This is due to the fact that the FastFix SWT fault replication sensor records all the necessary event for SWT replay. The only requirement is that the runtime configuration of the FastFixed application includes the following FastFix OSGi bundles: `eu.fastfix.client.faultReplication.guiRecorder`, `eu.fastfix.client.faultReplication.sensing`, `eu.fastfix.client.faultReplication` and `eu.fastfix.common.faultReplication.gui`.

3.2.5.1.2 Console Applications In the case of console applications, the compiled application needs to be instrumented in order to support FastFix record/replay. Assuming

that the main class of the console application is called `MainClass`. We must run: `java -cp .:<path-to-FastFix_server>/plugins/eu.fastfix.targetapplication.sensor.reap edu.hkust.leap.transformer.LEAPTransform -Xmx2g MainClass` This creates a version of the application for running at the client in a subfolder called `instrumentResult`. Both the original and the instrumented Java class files of the application need to be copied, by the deployment process, to the client using them. In order to deploy the application, it must be connected to the sensor that will monitor potential application crashes (unhandled exceptions). The shortcut to start the application should point to the sensor's main class `eu.fastfix.targetapplication.sensor.reap.StartSensor`.

3.2.5.1.3 AWT Applications In the case of AWT, the machine where applications are instrumented must have the `GUIAnon.jar` which comes from generating an executable jar of `eu.fastfix.targetapplication.sensor.javaapplication` and contains all utilities for GUI anonymization. If you run: `$java -jar GUIAnon.jar`, you'll be provided with all available options of GUIAnon (see Figure 3.2.5.1.3 below).

```
Usage Help : java -jar GUIAnon.jar -h
Launch app : java -jar GUIAnon.jar -launch MainClass
Instrument : java -jar GUIAnon.jar -instrument MainClass
Rip Graphical Interface : java -jar GUIAnon.jar -rip MainClass
Record Events : java -jar GUIAnon.jar -recevents MainClass
Record Listeners : java -jar GUIAnon.jar -reclisters MainClass
Convert Tracefile : java -jar GUIAnon.jar -convertl2e guistructure tracefile
Generate anonymized : java -jar GUIAnon.jar -anonymize guistructure tracefile
MainClass
Generate anonymized : java -jar GUIAnon.jar -smartanonymize guistructure
tracefile MainClass
```

Figure 3.25: GUIAnon usage instructions

To instrument the application (assuming a main in `MainClass`), we must run: `$java -jar GUIAnon.jar -instrument MainClass`, which generates an instrumented version of the application in a folder placed in the same parent folder as the main class and which is called `instrument_result`. Then, we must extract the graphical model of the AWT application by running: `$java -jar GUIAnon.jar -rip MainClass` Select all options available in the application's GUI (click every button, open every menu, etc...). Then close the application, which generates a file called `model.txt`. Just as in the case of console applications, in order to deploy the application, it must be connected to the sensor that will monitor potential application crashes (unhandled exceptions). The shortcut to start the application should point to the sensor's main class `eu.fastfix.targetapplication.sensor.javaapplication.StartJavaApplicationSensor`.

3.2.6 Patch Generation and Self-Healing

In its current state, the FastFix Self-Healing component applies to applications written in Java and for which the source code is available. A typical use case for self-healing and patch generation follows the steps below:



1. Instrumenting source code and generate application models.
2. Collecting a set of traces representing desired behaviors. This can be done by executing passing tests again on the instrumented application. The *Testing Model for Trace Collection* option in Figure 3.13 should be enabled for this.
3. Extracting patterns. This is achieved using the traces collected in Point 2. This results in more application models that will be loaded at runtime with the ones extracted in Point 1. Point 1,2 and 3 are performed just before the application is deployed.
4. Performing patch generation. This is done once the application is deployed and new files are received on the server side.
5. Patch merging and deployment. Once a new patch has been generated, it can be merged with the currently applied one and the outcome can be deployed.

These different steps are also illustrated in menu of the FastFix Self-Healing component (Figure 3.26).

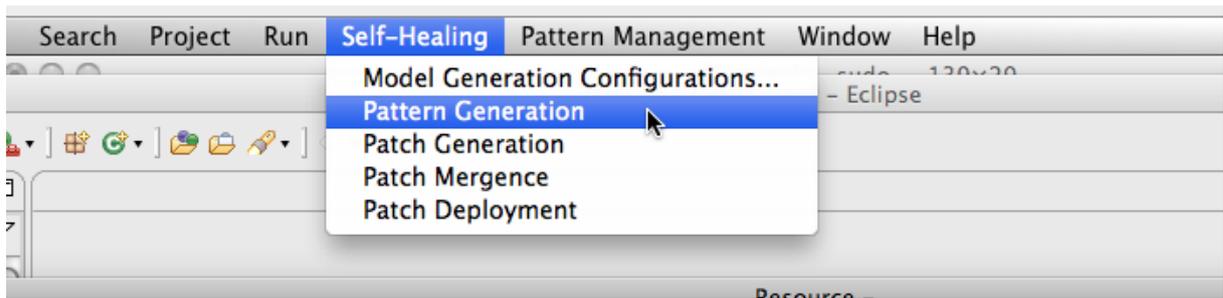


Figure 3.26: FastFix Self-Healing plugin menu

This menu matches the different steps describe above, except for Point 2, that is performed on the running application and is therefore controlled by parameters in the FastFix client (*Testing Mode for Trace Collection* in this case). It is worthwhile noting Point 2 and 3 are actually optional. When Point 2 is performed, it allows for more relevant patch validation. When Point 3 is also performed, it allows for an alternative patching strategy.

4 Developer Manual

In this section we describe, how developers can configure, extend, and change the FastFix functionality, in order to be able to provide remote maintenance services tailored to their customers' requirements.

4.1 Configuration Extensions

The FastFix platform consists of several components running on two distributed systems. These components provide specialized services to other components, which can be configured in order to set the components up for a specific runtime environment. For instance, the FastFix client listens for sensor calls on a specific port. However, since the default port might be reserved on some machines, FastFix allows users to change such parameters using a mechanism called *unified configuration* (see also Section 3.1.1). All components can be configured in a single place, namely a configuration file in XML format. The FastFix project wiki describes in detail how this configuration file is structured¹.

To guarantee the extensibility of FastFix, the configuration parameters for all components can be extended. In addition, new elements can be created for newly developed FastFix components. Details on the steps required to extend the configuration mechanism are described in the FastFix project wiki².

4.2 Feature Extensions

The following sections show how to extend the existing FastFix functionality to work with additional applications and technologies, for instance to satisfy additional monitoring needs and detect additional errors.

4.2.1 Context Observation

4.2.1.1 Implementation and Integration of New Sensors

Every application can be instrumented and integrated into the FastFix framework. The instrumentation can be done on source code or byte code level. On source code level, additional statements are added in the programming language used that log the occurrence of certain events as specified by the developer of the application. In a scenario where a virtual machine is used to execute a software application monitoring instructions can be added on byte code level without having access to source code of the application. Instrumentation can be done manually by developers or semi-automatically. The instrumentation approach implemented for the FastFix self-healing feature is an example for a

¹http://fastfixproject.eu/wiki/Configuration_file_structure

²http://fastfixproject.eu/wiki/Unified_configuration

semi-automatic instrumentation because the methods to instrument are specified in an expert file and the actual instrumentation is done automatically. It can be reused and extended for other instrumentation approaches.

In order to integrate a self-implemented sensor into the FastFix framework, two things have to be done. First, the sensor implementation has to use the lifecycle described in Deliverable D3.6 [12]. The sensor lifecycle allows the FastFix client to control sensors, e.g. switch them on or off. Second, the sensors have to communicate to the FastFix client via RMI or HTTP as described in Deliverable D2.4 [11]. This allows self-implemented sensors to send monitored events to the FastFix platform and use FastFix functionality like event storage or event correlation. As HTTP is available on almost all devices and it usually can transmit through firewalls, this constitutes a general communication mechanism that is available in most environments. Also, existing instrumentation solutions can be integrated into the FastFix framework in this way.

4.2.1.2 Extension of Existing Sensors by Adding Monitoring Code

Existing FastFix sensors can be extended to monitor additional types of events or additional properties. This extension has to be done by adding additional monitoring code, recompiling the sensor, and redeploying it. For example, the RCP sensor can be extended to monitor SWT events currently not being sensed by registering it as a listener to these SWT events and creating events representing the occurrence of those events by calling the event creation routines. Similarly, the expert file within the FastFix self-healing feature can be extended by additional methods to be sensed and then re-instrumenting the application code with the extended expert file.

4.2.1.3 Extension of Existing Sensors by Configuration

The TXT sensor is implemented so that the addition of new sensed fields in the WCF message can be added easily. In order to add a new field to sense from the WCF message the following needs to be added in the `appSettings` tag of the Web.config file of the ExecuteService:

1. A key-value pair that with a key named `XPath_WCF_Rule` and a value that denotes the position in the WCF message where the new field is located in the form of an XPath rule.
2. A key-value pair with a key named `WCF_child_list` and a value that denotes the name of the field to sense.
3. Save and close the file the Web.config file.
4. Restart the ExecuteService Web Site in the IIS Manager.

The new sensed fields are then added as data to the corresponding event. An example of steps 1 and 2 can be seen below:

```
<appSettings>
    .
    .
    .
    <add key="XPath_WCF/_Rule" value="/Envelope/Body/EntitiesRequest/
request/*/Envelope/Body/LoginRequest/request/*" />
```

```
<add key="WCF_child_list" value="SessionSequence,ClientLastView,  
WCFException,ClientProgRequest,UserCode" />
```

```
</appSettings>
```

In this example there are two XPath rules that denote the location of the new fields that we want the sensor to monitor. They are under the XPaths: `/Envelope/Body/EntitiesRequest/request/*` and `/Envelope/Body/LoginRequest/request/*` the names of the WCF fields that we want to monitor are: `SessionSequence`, `ClientLastView`, `WCFException`, `ClientProgRequest` and `UserCode`.

4.2.2 Event Correlation

Event correlation component is a key point for extension since it is the FastFix component where complex faults are detected. Each fault or error is represented by a pattern. Patterns of error are stored in the maintenance ontology. Hence, this ontology must be modified to add or update error patterns. Hence, it will extend the error type coverage of FastFix when the system is applied to monitor new applications.

Developers can edit ontologies by using ontology edition tools, such as Protege. However, because this is a very common use case of FastFix, we developed a user interface to facilitate the corresponding task. Using this approach, the extension of error patterns can be performed by engineers without any knowledge about ontologies, such as the maintenance team. The corresponding use case is explained in detail in Section 3.2.2.1. The interface consist of three tabs: *General Information* (Figure 3.16), *Symptoms* (Figure 3.17) and *Cause* (Figure 3.18). Via these tabs, users can enter the required information to create a pattern of error. The symptom tab is the most important one, since the symptoms contains the conditions to be matched by context events to infer the error emergence. The more different types of events are detected by sensors, the greater the coverage of errors, since the system can use a bigger variety of symptoms to conform a greater variety of patterns. Thus, to extend the error type coverage of FastFix when the system is applied to monitor new applications, developers must take into account also how to implement and extend sensors, which it is explained in section 4.2.1.

4.2.3 Pattern Mining

From a developer's perspective, pattern mining is an extension mechanism, since it discovers new patterns of unexpected behavior by analyzing and processing the executed event sequences associated to the application environment.

Because of its connection with the event correlation component, it might be considered as the main method to extend FastFix to support error detection for additional applications, technologies and especially, error types.

Nevertheless, there are some considerations that should be taken into account, in order to conveniently mine new patterns, depending on application specific properties, i.e. when it comes to average time delays between events, event generation rates (also associated with the sensors and which events are monitored), as well as minimum support, in order to consider certain event types as frequent.

Hence, there is a group of parameters that can be tuned or modified in order to achieve a better effectiveness while mining new patterns. This group of parameters are located

in the unified configuration file (`configuration.xml`), which are represented on figure 3.6. Among the main parameters that can be tuned, the main one is the pattern mining algorithm that will be executed in order to look for normal behavior patterns, in our case, we have selected the `PrefixSpan` algorithm, but any developer can develop and select the algorithm that would better fit for mining normal behavior (and, then in the mining phase, it will detect any event sequence that is different from the normal sequences).

Additionally, other parameters can be tuned, depending on the temporal properties and differences that any application can be characterized by. Thus, the developer can configure the following parameters, as explained in section 3.1.1.2:

- **support** (long): The value of this field represents the minimum number of occurrences (in percent) of a sequence of context events to be considered as frequent.
- **minimumSequenceSize** (int): It refers to the minimum number of context events needed to become a sequence.
- **sequenceTimeWindow** (long): It refers to the value, in milliseconds, of the time window of a sequence.
- **itemSetTimeWindow** (long): It refers to the value, in milliseconds, of the time window of a `itemSet`. A sequence is composed by one or more `itemSets`, so the value of **sequenceTimeWindow** must be higher than **itemSetTimeWindow**.

As a summary, depending on the application temporal properties, the developer can specify what should be considered as frequent (`support`), what number of events will be considered as a sequence, what is the time window of each sequence, as well as the period of time to consider events as part of a different item set.

4.2.4 Error Reporting

Extending the development of FastFix in terms of error reporting can proceed along two main lines: adding additional sensors that trigger the generation of error reports and supporting additional issue trackers.

Different type of applications and error types may require the design of new sensors. Sensor design is described in Section 4.2.1. The connection of sensors to error reporting is done by the fact that an error reporting sensor sends events of type `ERROR_REPORT_EVENT`, e.g. `eu.fastfix.targetapplication.sensor.javaapplication.internal.JavaExceptionSensor` (the sensor class for unhandled exceptions in Java AWT applications). Sensors who issue the same type of event will trigger the emission of an error report. These events are processed by the FastFix's client fault replication component. See the `handleContext` of the `eu.fastfix.client.faultReplication.internal.ContextListenerImpl` class for which of the event's fields are used in the creation of an error report. The only mandatory fields is the summary. If additional auxiliary files need to be sent to the server, their absolute paths (at the client) should be included as event fields and the processing of the fields at `eu.fastfix.client.faultReplication.internal.ContextListenerImpl` correspondingly extended.

Extending the number of supported issue trackers can be done by implementing the services described in the `eu.fastfix.server.error.reporting.abstractions.service` classes as done in the `eu.fastfix.server.error.reporting.trac` and `eu.fastfix.server.error.reporting.eventum`

examples. The only restriction is that the insertion of a ticket in the issue tracker should return a unique integer identifier because this identifier is used to name the folder on the server where any auxiliary logs are stored. As long as this is ensured, FastFix will behave correctly with any issue tracker.

4.2.5 Fault Replication

There are basically two approaches to additional development of FastFix fault replication: extending current record-replay components or creating new ones.

The extension of current record-replay depends on the component being extended. In the case of the current console and AWT sensors, applications are being monitored by using bytecode instrumentation that resorts to the SOOT tool (<http://www.sable.mcgill.ca/soot/>). For SWT applications are monitored using system graphical events. The logs generated by the record procedure are transferred with the error reports to the FastFix server. If any changes are made to the recorder, the replayer must be updated to reflect the changes in recording instrumentation and to inject into the replayed application the additional information being recorded.

Creating new record-replay components involves basically two steps: developing a sensor (see Section 4.2.1) that detects the events that are relevant to the record and replay you want to perform (and which sends the FastFix client an *ERROR_REPORT_EVENT* event) and update the replay mechanism in the ticket browser (see *makeActions* method in the *eu.fastfix.server.maintenance.ticketbrowser.views.FastFixTicketBrowser* class) so that the result of the record procedure can be replayed. The ticket browser uses tags embedded in the error report summary to decide on the replay method.

4.2.6 Patch Generation and Self-Healing

The FastFix patch generation and self-healing component tackles any Java application. In order to be applied, the source code of the application must be available and imported into Eclipse. The main features of the self-healing component are indeed accessed through an Eclipse plugin (see Figure 3.26).

A fully automated approach can be taken where no expertise on the application is used. This corresponds to the selection of the *Automatic Entry Point* option in Figure 3.11. Alternatively, some expertise on the application can be taken into account through the use of the expert file (Section 3.1.1.4). This allows for improved selection of the methods to be instrumented. Therefore this improves on the runtime overhead and the relevance of the extracted models and collected traces. Models and collected traces play themselves an important role in the relevance of the generated patches.

With the use of an expert file, for a new application to be self-healed using the FastFix platform, users must import the source code in Eclipse and define an expert file such as the one in Figure 3.12. This file contains a description of sets of methods of the application that are relevant for monitoring and control (i.e. prevention at runtime). Methods whose calls is triggered by user interactions or any other application entry points correspond to such method. For instance, typical sets of methods are the ones that handle graphical events, e.g. *actionPerformed* methods of the SWING library, etc.

5 Summary

This document gives a conceptual overview of the FastFix platform, describes how to deploy, set up, and use the FastFix platform for remote maintenance, and explains how to extend it for further applicability. The document is a supplement to the FastFix source code, which can be accessed in the FastFix source code repository on SourceForge¹ and complements the detailed Javadoc documentation of the FastFix source code as well as the project Wiki².

¹<https://svn.code.sf.net/p/fastfixrsm/code/trunk>

²http://fastfixproject.eu/wiki/Main_Page



Bibliography

- [1] M. del Carmen Calle and S. Zamarripa López. FastFix Deliverable D4.4: First iteration prototype of the Event Processor. Technical report, 2012.
- [2] J. Garcia, P. Romano, L. Rodrigues, N. Coracao, N. Machado, and J. Matos. FastFix Deliverable D3.5: 1st Prototype of the Error Reporting. Technical report, 2011.
- [3] J. Garcia, P. Romano, L. Rodrigues, N. Coracao, N. Machado, and J. a. Matos. Fast-Fix Deliverable D5.3: 1st prototype of the execution recorder/replayer tool. Technical report, 2011.
- [4] J. a. Garcia, P. Romano, L. Rodrigues, J. a. Matos, J. a. Nuno Silva, J. a. Barreto, and T. Röhm. FastFix Deliverable D5.4: 2nd prototype of the execution recorder/replayer tool. Technical report, 2012.
- [5] B. Gaudin, Z. Cui, P. Monjallon, and M. Hinchey. 3rd Prototype of the Self-Healing and Patch Generation Component. Technical report, 2012.
- [6] B. Gaudin, M. Hinchey, P. Nixon, and N. Al Haider. FastFix Deliverable D6.4: 2nd Prototype of the Self-Healing and Patch Generation Component: FastFixSH. Technical report, 2012.
- [7] B. Gaudin, M. Hinchey, P. Nixon, R. Ali, and N. Al Haider. FastFix Deliverable D6.3: 1st Prototype of the Self-Healing and Patch Generation Component: FastFixSH. Technical report, 2011.
- [8] B. Gaudin, R. Yates, and M. Hinchey. FastFix Deliverable D6.6: 4th Prototype of the Self-Healing and Patch Generation Component. Technical report, 2013.
- [9] E. Guzmán, A. Mahmuzic, J. Garcia, and W. Maalej. FastFix Deliverable D3.3: 1st Prototype of the Context Observer. Technical report, 2011.
- [10] D. Pagano, E. Guzmán, J. Cano, N. Narayan, A. Mahmuzic, A. Waldmann, S. Zamarripa López, D. De los Reyes, J. Garcia, and W. Maalej. FastFix Deliverable D2.2e: Integration Plan and Technical Project Guidelines - extended version. Technical report, 2012.
- [11] D. Pagano, T. Roehm, E. Guzmán, S. Zamarripa López, J. Garcia, B. Gaudin, J. Cano, and W. Maalej. FastFix Deliverable D2.4: Architecture Changes and Change Rationales. Technical report, 2012.
- [12] T. Roehm, J. Garcia, and D. Pagano. FastFix Deliverable D3.6: Refined and Integrated Version of Context Observer, User Profiler and Error Reporting. Technical report, 2012.

- [13] T. Roehm, D. Pagano, S. Zamarripa López, and M. del Carmen Calle. FastFix Deliverable D3.4: 1st Prototype of the User Profiler. Technical report, 2012.
- [14] S. Zamarripa López and M. del Carmen Calle. FastFix Deliverable D4.6: Second refined prototype of the event correlation component. Technical report, 2012.
- [15] S. Zamarripa López, M. del Carmen Calle, E. Guzmán, T. Roehm, D. Pagano, and W. Maalej. FastFix Deliverable D4.5: 1st iteration prototype of the pattern mining module. Technical report, 2012.