A Java API for Advanced Faults Management

M. H. Guiagoussou
Open Management Software Inc.
5600 Mowry School Rd, No 220
Newark, California 94560
USA
mahamatg@omsi.com

R. Boutaba
Department of Computer Science
University of Waterloo
Waterloo (Ontario)
N2L 3G1, Canada
rboutaba@bbcr.uwaterloo.ca

M. Kadoch
École de Technologie Supérieure
1100 Rue Notre-Dame Ouest,
Montréal (Québec)
H3C 1K3, Canada
kadoch@ele.etsmtl.ca

Abstract

The paper proposes an alternative for modeling managed resources using Java and Telecommunication Network Management standards. It emphasizes functions related to Fault Management, namely: diagnostic testing and performance monitoring. Based on Java Management Extension (JMX^{TM}), specific extensions are proposed to facilitate diagnostic testing and performance measurements implementation. The new API also called Java Fault Management Extension (JFMX) consists of managed objects that model real resources being tested or monitored and support objects defined for the need of diagnostic testing and performance measurements. The paper discusses four Java implementations of a 3-tier client/server scenario focusing on the "SystemUnderTest" package of the new API to instrument a minimalist Managed System scenario. These implementations are respectively built on top of the following Java based communication infrastructures: JMX/JFMX, RMI, CORBA/Java, and Voyager^{TM}. The paper extends the Voyager implementation with JMX/JFMX and uses their dynamic and advanced features to provide a highly efficient Solution. The later implementation also uses Mobile Agent paradigm to overcome well-known limitations of the RPC based implementations.

Keywords

Diagnostic Test, Performance Monitoring, JMX, JFMX, Mobile Agent, CORBA, RMI, Voyager.

1 Introduction

Java is a widely used programming language that matured and became a robust platform for advanced Enterprise Applications development. In addition to the distributed computing capabilities provided by Remote Method Invocation [16] and CORBA implementation [14], several other players enrich the Java Core Platform [17, 3]. Significant contributions to Network and Systems Management are provided by the Java Dynamic Management Kit (JDMK^{TM}) [18] former Java Management API (JMAPI), and the Java Management Extension (JMX^{TM}) [19]. JMX is a Sun Microsystems trademark that provides a standard for the instrumentation of manageable resources and the development of dynamic agents to manage these resources. It consists of a set of specification and development tools that help developers build Java management environment and provide innovative management solutions. JMX also includes a compatibility test suite and a validation tool. It is based on a proven technology, which has already been used in the JDMK.

JMX specifies a management architecture that consists of three layers: Instrumentation, Agent, and Manager layer. The Instrumentation layer allows developers to rapidly provide Java based management solutions for both computing and telecommunication systems. It gives instant manageability to any Java technology-based object. The Agent layer provides a management agents API. In JMX, agents are containers that provide core management services, which can be dynamically extended simply by adding JMX resources. The Manager layer provides management components that can operate as a manager or agent or both for distribution and consolidation of management services. In order to build upon existing management technologies, the JMX specification also provides interfaces to the most widespread management protocols in use today including SNMP, CIM/WBEM, and CMIP [19].

The work presented in this paper is supported by the Java Management Extension. It proposes some Fault Management extensions on top of the JMX. The proposed Java Fault Management Extension aims at facilitating the Development of Diagnostic Tests and Performance Measurements Applications [6]. The JFMX API specification defines the implementation of a set of high level abstract classes to be used by Testing and Monitoring Applications developers. Concrete subclasses are derived from these abstract classes to implement concepts defined in TMN tests and performance management standards, namely X745 [7] and X.737 [8]. JFMX classes may be extended to allow the development of more specific applications and add to existing systems fault diagnostic and performance monitoring features. Thanks to Java and JMX, these MBeans (Manageable Beans) can be instantiated and experimented at runtime. We also used Voyager [13, 5] to add agency and mobility features within JFMX. JFMX API is organized into three packages mapped to the basic JMX layered architecture. These packages are: *jfmx.DiagnosticTestsServer* (Manager layer), *jfmx.TestAgentServer* (Agent layer), and *jfmx.SystemUnderTest* (Instrumentation layer). Core packages are related to utilities and enumerated classes such as Managed Object States/Status, User information, and Object Reference. Future versions of the JFMX will include additional packages related to Alarm Management, Fault Localization and Fault Resolution.

After a review of the main JFMX packages, the paper discusses four Java implementations of a simplified Managed System Instrumentation Scenario using the "SystemUnderTest" package. The objective is to show how Java infrastructure can be used to provide an integration platform of the main Java distributed programming technologies (JDMK/JMX, RMI, CORBA/Java, and Voyager) in the context of diagnostic testing and performance monitoring. The paper also investigates the use of Mobile Agents [2, 4, 9] as a complementary alternative to RPC based implementations. This allows developers to implement dynamic and highly efficient Faults Management Applications using Java [10, 12].

2 The Java Fault Management API

This section provides an overview of the proposed Fault Management API. It describes the Java Fault Management Extension (Figure 1) as built on top of the current JMX Reference Implementation [19] and Voyager^{TM} [13]. The Java Fault Management Extension adds new packages for the purpose of Networks and Systems Fault Management [6]. The JMX compatibility test suites and validation tools are used to verify that the defined MBeans are JMX Compliant. At this early specification of JFMX, we focused on Intelligent Diagnostic Testing and Performance Monitoring. In order to provide a complete Fault management API, complementary packages for alarm management and fault localization will be added in the future. JFMX is built on top of two additional Java environments (Java Dynamic Management Kit, and Voyager) to provide advanced dynamic, agency and mobility features.

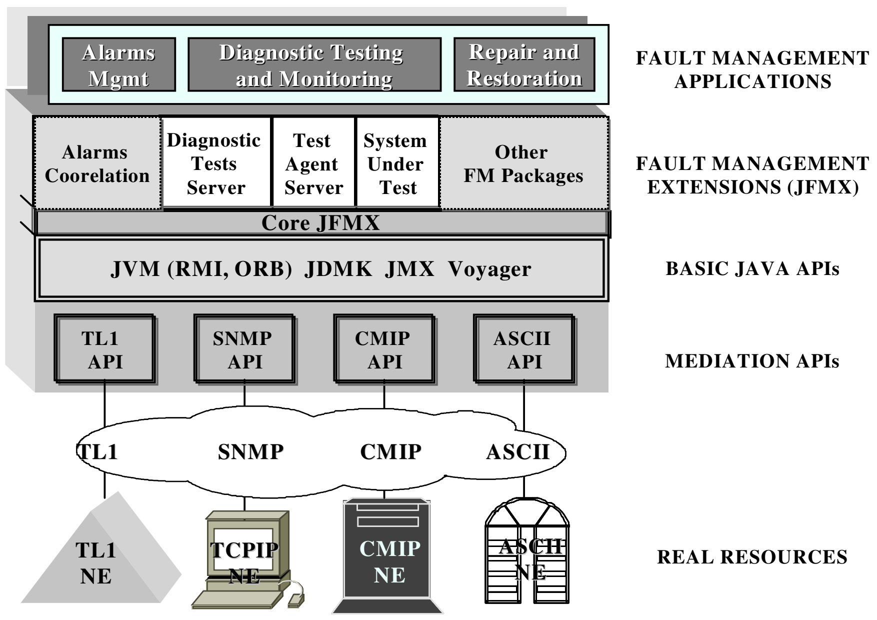

Figure 1: Java Fault Management Extension

JFMX API is organized into three main packages: *jfmx.DiagnosticTestsServer*, *jfmx.TestAgentServer*, and *jfmx.SystemUnderTest.* In the following, these packages are briefly described.

Diagnostic Tests Server: This package is composed of Manager MBeans defined to implement the coordination and management of a set of performers and lower level agents triggered by a user session request. These MBeans implement concepts such as: Session Request Receiver, Test Conductor, Diagnostic OS, Performance OS, Diagnostic Session, Monitoring Session, Fault and Alarm. The package covers the implementation of tests and performance management application MBeans that manages multiple users sessions (registration, access control, sessions creation, conductors and agents coordination, test results correlation and fault diagnostic). The package also supports a dynamic Class Loader that allows any Fault Management Application to load at runtime new Agent, Test Session, or Test Object class.

Test Agent Server: This package is composed of MBeans defined to facilitate the implementation of advanced dynamic and mobile test/performance agents. It allows dynamic Tests/Monitoring MBeans creation, registration, and roaming, test lifecycle management (start, stop, suspend and resume), test results collection, and events notification. Test Object MBean is defined to model individual test activity. Sessions aggregate one or several Test/Monitoring Objects. The Performer Agent implements set of dynamic functionality to control test and monitoring activities. Fault Detection Agent is defined to detect faults and perform lower level alarms filtering and correlation. Detection and Performer agents are built with dynamic and migration capability [4, 12] to allow them to travel the nearest possible to the managed resources and perform their duties. This package utilizes event-driven approach to model communication between agents and managers. It also defines Diagnostic Test Results, scheduling Conflicts and test-related events as JMX notifications. Tests/Monitoring scheduling is achieved using JMX Timer. A Test Action Request Receiver (TARR) MBean is defined to implement test and monitoring service interface as specified in the X.745 [7].

System under Test: This package is composed of MBeans modeling real resources under test and/or monitoring. It includes tested/monitored resources (Managed Object, Managed Object under Test, Manageable Component and Manageable Relation), tests access views and logical test units (Associated Object) representing test resources connected or embedded into real resources, and vendor specific test/monitoring resources (Remote Testing Resource). The package includes an abstract Support Object class from which all classes defined for the need of testing and monitoring (Session, Test Object, Test Schedule and Monitoring Schedule) are derived. The package also defines additional utility classes such as Users Contact Information, Object Reference, Managed Object States/Status, and XML configurations Parser.

In the JMX terminology, a JMX manageable resource (MBean) is a resource that has been instrumented in accordance to the JMX Instrumentation Level Specification and tested against the Instrumentation Level Compatibility Test Suite [19]. A manageable resource is a software application, a hardware device, or a software implementation of a service. In order to be instrumented, a resource can be fully written in Java (one or several MBeans) or just offer a Java wrapper. While Java provides portability and ease of prototyping, JMX ensure flexibility, inter-operability, and dynamic management capabilities required by current service-driven networks. JMX agent and manager levels provide flexible, distributed, and dynamic management infrastructure to be implemented in Java.

Depending on the managed environment, several protocols (SNMP, CMIP, TL1, and ASCII) can be used to synchronize real resources with Managed Beans and instrument them (see lower part of Figure 1). JMX/JDMK provided an SNMP API to allow the instrumentation of SNMP manageable Resources [19]. Outback provided *jSNMP* [15] a cross-platform and higher level SNMP API. AdventNet implemented on top of JDMK a dynamic SNMP Access solution [1, 19]. Future releases of JMX are expected to support CMIP/Q3 manageable telecommunication resources. Alternatives to access Telecommunication NEs are available via Bellcore TL1 protocol. A Java based TL1 Translator developed by Advanced Network Solutions is proposed in [20]. ASCII based protocols such as MML (Man Machine Language) may also be used in a Java Environment to access NEs such as Wireless Switches.

3 Modeling Managed System using JFMX

The “SystemUnderTest” package is defined to allow developers instrument managed systems which functionality is to be tested and monitored. This section describes the classes required to build managed system views on top of which basic management applications may be developed to detect fault conditions, perform diagnostic testing and conduct monitoring of performance degradations. This package allows developers to focus on advanced fault management features such as fault resolution and troubleshooting. It provides a refined instrumentation layer and allows implementation of Network Elements interacting with their testing resources via static agent functions, vendor specific testing units, or Java based mobile agents [4, 10, 12].

3.1 SystemUnderTest Package Description

As illustrated by the OMT diagrams in Figures 2 and 3, the SystemUnderTest package is composed of several types of MBeans: Managed Objects, Support Objects, and Utilities Objects. Each MBean implements a management interface that is compliant with JMX specifications.

Managed Objects

Managed Objects model physical or logical manageable resources. A basic Managed Object MBean is defined with limited testing capability. Managed Object under Test is a subclass of the basic Managed Object class extended with advanced test and monitoring capabilities. Associated Objects are defined as Managed Objects that represent test instruments or measurements devices. For Diagnostic and Fault localization purpose, Managed Objects under Test are defined as aggregation of Manageable Components at which level the localization of faults stops. Users are free to define the level of granularity of this decomposition. The lower level of Figure 2 contains a sample decomposition that consists of software and hardware manageable components, manageable relations and their possible associations.

Detailed descriptions of the main Managed Object MBeans are presented below.

Managed Object MBean: Any class that defines a manageable resource may extend this basic Managed Object class (MO). This class represents objects that provide JMX compliant software management view of physical or logical managed resources [8, 19]. It implements limited testing and monitoring capabilities (e.g., simple ping test).

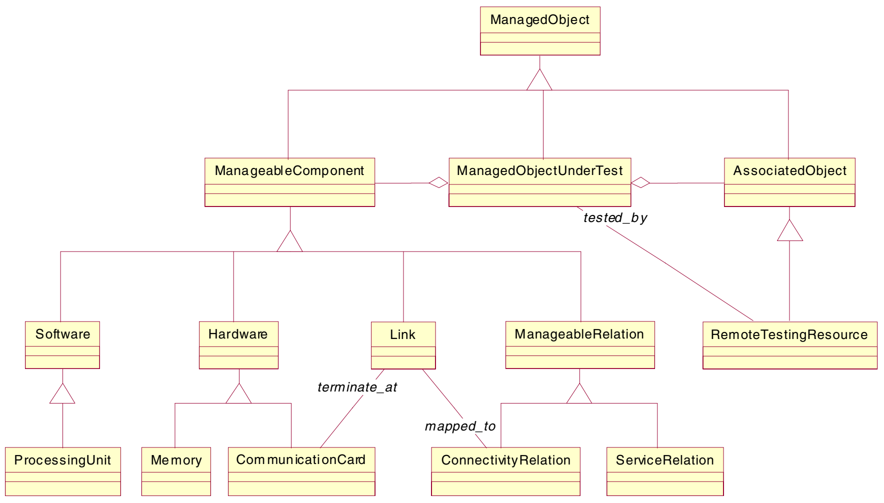

Figure 2: Managed Objects OMT Model

Managed Object under Test MBean: This is an advanced Managed Object that represents managed resource which functionality is to be tested [7, 8, 19]. It is derived from the basic Managed Object MBean and extends its diagnostic testing and performance monitoring capabilities. Managed Object under Test MBean (MOT) models either a simple resource or an aggregated resource. It has self-monitoring capabilities and several interfaces to allow managers and agents to perform advanced testing, collect detailed performance measurements, detect and diagnose faults occurring in the target managed resource down to its Manageable Components.

Manageable Component MBean: This class also derived from the basic Managed Object class defines a lower level of granularity for the decomposition of managed resources. Manageable Components (MC) represent the leaf of any diagnostic tree of their aggregated parent object (i.e., MOT). They may have dependency relations with other Manageable Components or Managed Objects (e.g., terminate_at, mapped_to, and served_by). In order to model complex managed resources or change the representation of existing Managed Objects in a management information model, Voyager Facet mechanism [5, 13] is used to dynamically add both Manageable Components and Relations at run time

Associated Object MBean: This MBean class is a MO that models a view of testing resource or test access points used for testing and monitoring of MOTs. Examples of Associated Objects (AO) include Mediation Device, Test Unit, SNMP/CMIP Agent, Monitoring Agent, and Embedded Probes.

Remote Testing Resource MBean: Remote Testing Resources (RTR) are derived from AO. They are designed with remote test/monitoring management capability and may in turn be specialized to vendor specific resources with advanced test/monitoring features and technologies.

Support Objects

The support objects classes defined in Figure 3 (a) represent objects needed for the purpose of test and performance management. They do not represent real resources.

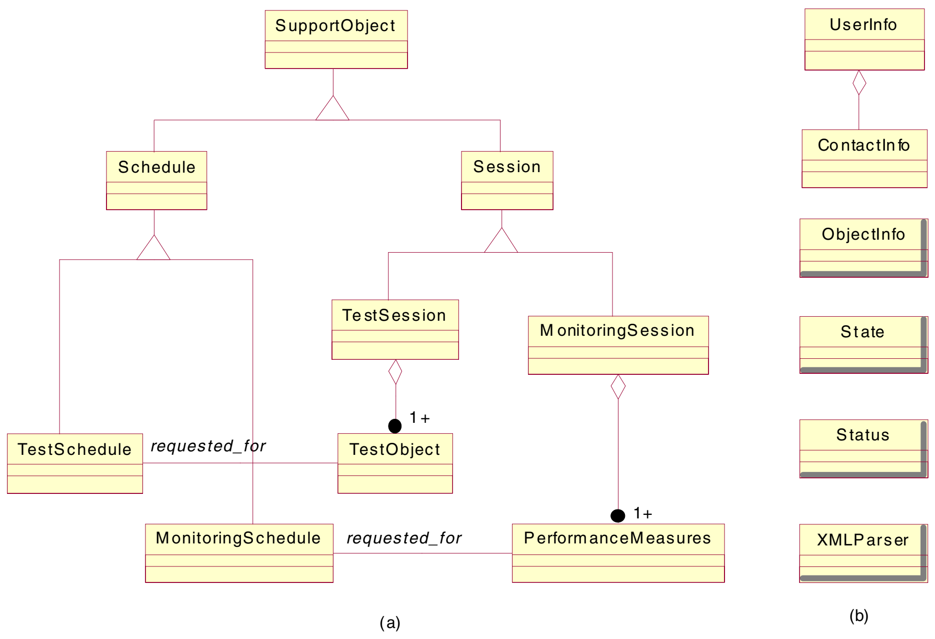

Figure 3: Support Objects Model and Utilities Classes

Support Object MBean: This class defines an abstract support object containing common information needed for test and performance management purpose.

Monitoring Schedule MBean: This is a Support Object that contains monitoring schedule data. It is used to keep information such as Operation State, Availability Status of the resources on which monitoring activities are scheduled. It is used to notify available performance measurements, monitoring test results to pre-registered users. Indeed, a user may be granted passive monitoring of ongoing tests scheduled by other users. These grants are configured in the monitoring schedule.

Testing Schedule MBean: This is a Support Object that contains tests scheduling information. It contains data such as start time, stop time, scheduling status, used AOs, and tested MOTs. At a given time two conflicting schedules may be requested for the same MOT or AO. It is the role of the manager and agents to resolve the conflicts by either rejecting the test request or proposing an alternative available time interval. It is also the test managers and agents responsibility to reject further schedules when testing resources reach their maximum capacity.

The remaining Support Objects illustrated in Figure 3 (a) are not covered here. They are part of other JFMX packages (Manager and Agent layer). These support objects are Session, Test Session, and Monitoring Session. The later are respectively defined as aggregation of Test Objects and Performance Measurements. Each Test Object (respectively Performance Measurement) is associated with a predefined Test Schedule MBean (respectively Monitoring Schedule MBean) indicating the time at which it should start, the expected termination time and other relevant information.

Utilities Objects

Several utility classes are defined in JFMX and used by the "SystemUnderTest" package. The utility classes described below are depicted in Figure 3 (b).

UserInfo MBean: This class defines information on tests and monitoring services subscribers. It contains authentication data such as login and password and carry detailed information related to user accounts (name, group, and category), and number of ongoing tests/monitoring activities initiated by the user. Complete user contact information is defined as an aggregated class (ContactInfo). UserInfo MBean is used to page or send e-mail to the user at the occurrence of a fault, when critical test results are available, or when diagnostic agents require help to complete a task. This MBean is able to prompt the user on a web interface, paint an applet or a dialog window, and ask for helps and information updates.

ObjectInfo MBean: This class defines a reference to a managed object or to a support object. It is used to store information on objects defined in the JFMX package. It is generally used to reference objects in a relation or in a list.

Other utility classes are defined and used in JFMX. These definitions cover Managed Object and Support Objects State/Status, and also XML configuration file parsers. The JFMX core package defines Sate/Status attributes that are used to detect real resources manageable behaviors relevant for fault management. The state characteristics define current behavioral information on the resource carried by attributes such as Operational, Administrative, Usage or User defined State. Status characterizes continual behaviors like Connectivity (bandwidth sharing), Service (Quality Of Service), Synchronization (accurate view of the real resource), or a User defined Status. Associated Object, Managed Object Under Test and Manageable Components inherit the basic State/Status properties of their parent Managed Object. Additional status characteristics are defined to track diagnostic test and monitoring activities in MORTs (Testing/Monitoring Status), AOs (Testing/monitoring Status), and Support Objects (Schedule, Session, and Test Object Availability Status). Manageable Components are defined with two useful properties that allow localizing and measuring the impact of a faulty component in the MORT: "Location in MORT" and "Health Status".

The remaining of the paper focuses only on the use of JFMX, specifically that of a reduced set of MBeans defined in the "SystemUnderTest" package to model and instrument real manageable systems.

4 JFMX Implementations of a System Under Test

In this section we consider a minimalist Client/Server test application scenario and implement it using the "SystemUnderTest" package. Four implementation alternatives based on JMX, RMI, CORBA and Voyager are also presented here.

4.1 Generic Scenario Description

The scenario considered for implementation is illustrated in Figure 4. It is a test and monitoring application composed of multiple User Interfaces in the client tier, a Server/Manager and an Agent in the middle tier, and Managed Resources Accesses in the last tier.

The generic test server provides a limited test access to a number of MBeans implementing different views of managed resources. The server also provides its clients with access to MBeans that represent implementation of management services such as Configuration, Diagnostic Testing and Performance Monitoring.

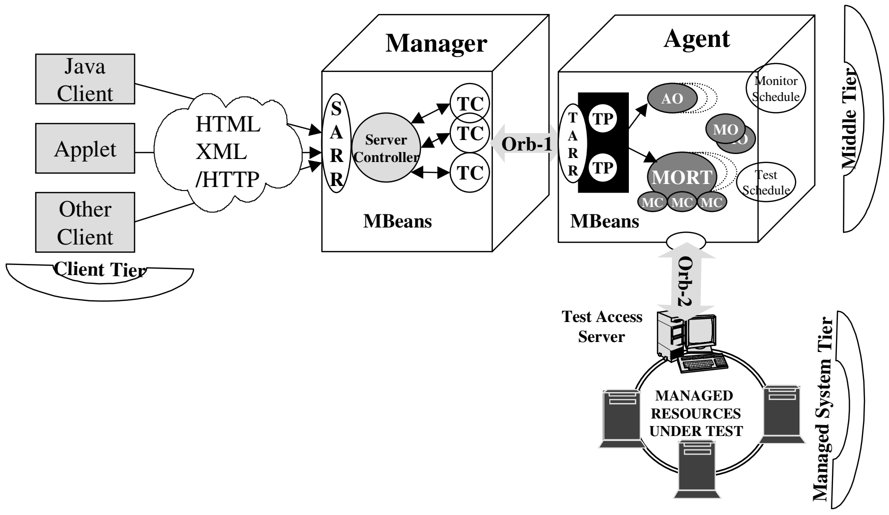

Figure 4: Generic Test Management System

The test and monitoring server is implemented as a Java Server with the following typical behavior:

(1) set required Orbs (HTML/XML/HTTP, Orb-1, Orb-2) parameters,
(2) create and export managed objects (MOs, MOTs, and MCs),
(3) create services support objects (SARR/TARR, TC/TP, AOs and RTRs),
(4) listen to multiple clients, perform test actions and responds to requests,
(5) return to client available results, state/status values, and performance data.

4.2 JFMX Implementation

The scenario illustrated in figure 4 is implemented here using the JFMX API. Figure 5 depicts the implementation and shows the interactions between the involved JMX/JFMX and non-JMX components.

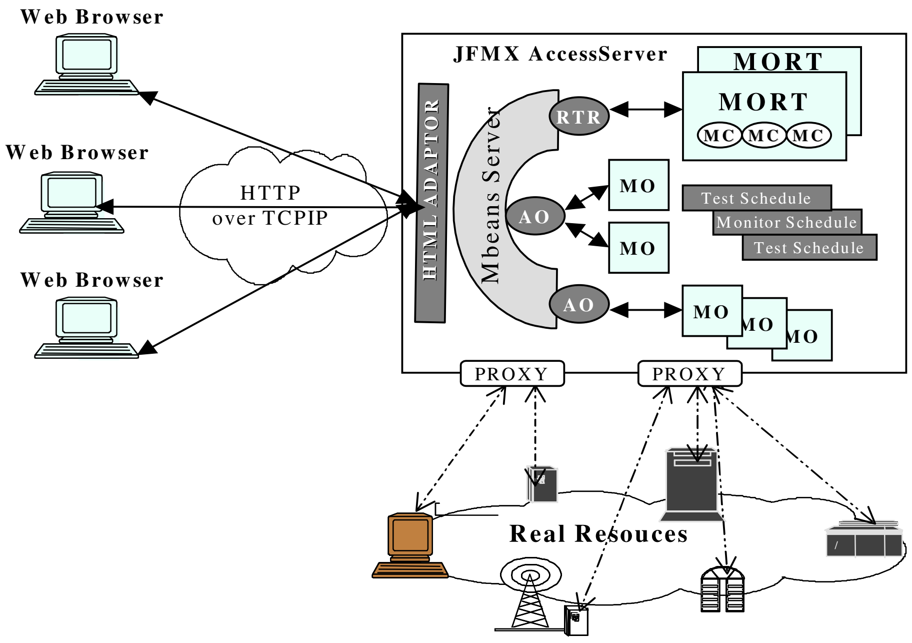

Figure 5: Pure JMX Implementation of the Test Server

The implementation of the Test Access Server using JFMX is as follows:

Step 1: *Definition of MBean interfaces (implementation).* Real resource MBeans include: Managed Object MBean (Managed Object), Manageable Component MBean (Manageable Component), Associated Object MBean (Associated Object), and Managed Object Under Test MBean (Managed Object Under Test). Services MBeans include Monitoring Schedule MBean (Monitoring Schedule) and Testing Schedule MBean (Testing Schedule).

Step 2: *Creation of a JMX MBeans Server:* create and run the Test and Monitoring Access Server by creating one MBeans Server to provide access, configuration, monitoring and testing of MBeans created in step 1. JMX provides a sample MBean server called BaseAgent. This agent can be used as it is, or a customized MBeans Server can be defined as Test/Monitoring Access Server.

Step 3: *Running a Web Browser as a client:* A Web browser can then be launched and connected to the BaseAgent. This allows to dynamically manipulate the previously created MBeans (create, configure, perform test/monitoring actions, and unregister).

Step 4: *Creation of System Under Test scenario:* the last step proceeds with administrative tasks by using the interface provided by the HTML Adapter (or any similar service in a customized MBeans Server). An HTML adapter is created when the BaseAgent is started in step 3. This can be used as a web interface to create, register and remove MBeans instances. When the MBeans needed to run the Test Server scenario are successfully created, the user can then browse the MBeans by defining a selection criteria (filtering scope), view each MBean attribute, invoke MBeans test and monitoring operations, and experiment with the MBeans. The experimentation consist to get and set MBeans attributes, perform operations on these MBeans (e.g., testing and collection of monitoring data), create new instances of MBeans, register as event listener and receive notifications generated by the MBeans.

4.3 RMI/CORBA Implementations

RMI-Implementation and CORBA-Implementation are quite similar and hence described together. In the case of CORBA there is a need for IDL definition and additional compilation of IDL files by an idl2java compiler. The RMI implementation does not require IDL files, but still interface definitions are required to follow certain rules. For example each interface must extend java.rmi.Remote and all of their methods must throw RemoteException. In addition, the implementation of these interfaces should extend java.rmi.UnicastRemoteObject and implements a default constructor that throws RemoteException. Finally, the rmic tool must be used on the interface implementations to generate the appropriate stub and skeleton classes. This should be done any time a change is made on the interface implementations. The MBeans defined by JFMX can be used directly by the RMI and CORBA implementations.

RMI/CORBA implementations of the Test Access Server is build as follow:

Step 1: *Definition and Implementation of MBeans Interfaces (IDL definition and Implementation):* same as in the JMX implementation, except that the previous set of rules related to RMI should be respected both by the interfaces and their implementations (idl2java compilation required in the case of CORBA).

Step 2: *Implementation of the Server:* an RMI security manager is installed (ORB and BOA are initialized for CORBA), then instances of "SystemUnderTest" MBeans classes are created and bound in the RMI registry (exported to the ORB), and the server then loops on listening to possible clients.

Step 3: *Implementation of Client:* client program will simply installs an RMI security manager (initialize an ORB), obtains a proxy to the MBeans objects exported by the test access server, bound to them in the RMI registry (bind to the CORBA Server Objects) and use them remotely.

4.4 Voyager Implementation

Voyager implementation is similar to the RMI/CORBA implementation with the following difference: it is much more easier as it does not require to follow rules like those used in RMI or CORBA. Like RMI it requires a Java interface to either extend java.rmi.Remote or Voyager interface com.objectspace.voyager.IRemote. Unlike RMI and CORBA, Voyager does not require that methods throw RemoteException, idl2java/rmic compilation. There is no need for stub and skeleton generation. This is done automatically by Voyager. The steps for the implementation of the Access Server scenario based on Voyager are as follow.

Step 1: *Definition and Implementation of MBeans Interfaces: s*ame as in the case of the JMX implementation with the appropriate extension needed for each interface.

Step 2: *Implementation of a Voyager Server:* a server program is started on a given port in the local or remote host, its built-in HTTP server is then enabled, instances of SystemUnderTest MBeans classes are created and bound to their provided names in Voyager naming service. The server then listens to eventual request from clients and process them.

Step 3: *Voyager Client Implementation:* a client program is started on a given port in the "localhost" or any other host, then a proxy to the MBean objects is bound to the server defined in step 2 (*VoyagerAccessServer)*. The remote Voyager Objects can then be experimented by performing the desired tests and monitoring actions.

We presented in this section four alternative implementations of a sample scenario based on the "SystemUnderTest" package of JFMX: a web based JMX instrumentation, a full Java RMI, a CORBA-based, and Voyager implementation. Due to the simplicity and ease of use of the provided APIs, the total development cost was significantly low [5].

5 Advanced Communication and Mobility Features

In this section, we analyze the benefits provided by advanced JMX and Voyager capabilities such as dynamic classes loading, dynamic aggregation [5, 13], agency and mobility [10, 12, 13]. We also discuss the way Voyager can be used as a Universal ORB to access non-Voyager implementations.

5.1 Dynamic Aggregation, Agency and Mobility

Both JMX [18, 19] and Voyager [5, 13] provided us with several attracting features: dynamic class loading, dynamic aggregation, object serialization and mobility, and agent autonomy and mobility. Dynamic aggregation is a Voyager feature that allows a developer to add at runtime secondary objects as facets to an existing object. This is a step forward in object modeling that complements traditional inheritance and polymorphism. For example, Manageable Components and Manageable Relations MBeans can be dynamically added to Managed Objects under Test to reflect addition of software features or extension of hardware in a real managed resource. JMX provides another interesting dynamic mechanism based on MLET (Management Applet) concept. MLET are MBeans obtained from remote URLs [18, 19]. Adding to these two features, object serialization and class loading, we obtain a workable MBeans migration infrastructure. Thanks to Voyager, more advanced object/agent mobility (Figure 6) can be achieved [13].

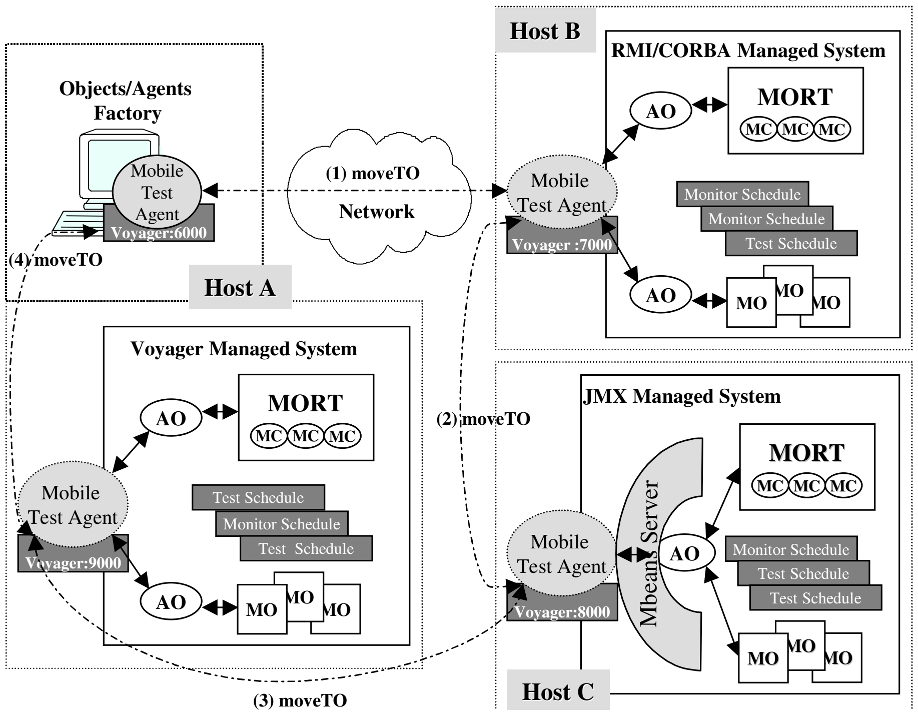

Figure 6: Mobile Agent Implementation of the Test Access Server

We used the previous dynamic features to enhance the traditional RPC based implementations presented in the previous sections. In particular, we implemented the Test and Monitoring Access Server with Mobile Agents [4, 9, 10] using Voyager [5, 13] and JFMX [6]. In Voyager any serializable object can be easily moved between programs at runtime as illustrated in the Figure 6. A proxy mechanism is used to forward a message sent from an object's old location. The proxy is automatically updated with the new location and messages are resent. Mobile autonomous agents that move themselves between programs and continue to execute upon arrival are easily created using Voyager facet mechanism.

The steps involved in the full Mobile Agent based implementation of the generic scenario presented in section 4.1 are as follow:

Step 1: *Definition and Implementation of Test/Monitoring Access Server:* one of the RPC implementations (i.e., JFMX, RMI, CORBA, or Voyager) can be used. The Test Access Server should be started on Host A.

Step 2: *Implementation of a mobile Object/Agent:* an object that will be downloaded the nearest possible to the Access Server implemented in step 1 should be implemented with test/monitoring performer capability. With the current JFMX, one of the following test agents may be directly used: Associated Object, Remote Test Resource, Test Performer, or Diagnostic/Monitoring Agent.

Step 3: *mobile Object/Agent factory:* create locally an instance of the mobile objects/agents factory in the same Voyager daemon (running on Host A). The agent factory is then used to create an instance of the object/agent to be migrated. It then converts it to a mobile agent (by adding the Mobility facet to the selected agent MBean). And finally moves it to another Voyager enabled host (by invoking its *moveTo* method).

The agent may performs a round trip for example interacting with a Voyager Access Server on the same Host (A), an RMI or CORBA Access Server on remote Host B, and a JFMX Access Server on Host C. At each step the agent may autonomously decides its itinerary moving to one of these destinations based on scheduled activities or other embedded knowledge skills. The agent will then perform test/monitoring tasks and send back results or returns to its original location (Host A). Meanwhile, any agent, service MBean, or remote client may call an exported method of the roaming object. As stated earlier, Voyager keeps an updated reference to the current location of the roaming agent MBean

The main motivation [2, 4, 9] of mobility in this implementation alternative is to reduce management information traffic in the network during period of stress and to faster the response time to network events and faults. Coupling mobility feature with other JFMX/JMX dynamic features, we were able to achieve a highly efficient web based solution for Diagnostic Testing and Performance Monitoring.

5.2 Communication and Interoperability

A Voyager program is both a universal client and a universal server [5, 13]. It communicates with other Voyager programs using its own protocol called VRMP (Voyager Remote Method Protocol), and supports simultaneous bi-directional communication with other programs (JMX/JFMX, RMI/CORBA, and DCOM). A Java component does not have to be modified to send or receive messages using these standards. No stub generators or helper classes are required. Since Voyager is a *Universal Client*, the original Voyager client program is automatically a JFMX/JMX client, an RMI/CORBA client, or a COM client and can bind to any type of Test Access Server. This means that all the Access Server implementations do not need to any modification to be managed by a fixe or a mobile Manager/Agent developed with JFMX and Voyager. Voyager has another interesting feature with respect to interoperability issue. It is a universal gateway, which means that Voyager automatically translates messages between non-Voyager implementations such as RMI, CORBA, or COM programs. For example, if an RMI or CORBA server binds an object into a Voyager naming service and a COM client looks up the object, messages sent from the COM client to the RMI /CORBA server are automatically routed through the Voyager naming service and translated to VRMP and vice-versa. Note that Sun implementation of IIOP (Internet Inter Orb Protocol) provides similar gateway to make RMI programs communicate with CORBA programs.

6 Conclusion

Based on the Java Management Extension (JMX), we have proposed a Fault management API, the Java Fault Management Extension (JFMX). The API is composed of three main packages mapped to the JMX architecture: Instrumentation layer (*jfmx.SystemUnderTest*), Agent layer (*jfmx.TestAgentServer*), and Manager layer (*jfmx.DiagnosticTestsServer*). The paper presented a simple test and monitoring Client/Server scenario and discussed four implementations of this scenario based on JMX/JFMX, RMI, CORBA and Voyager. The objective was to propose extensions for Fault Management (JFMX), show the practical usefulness of some advanced and dynamic features provided by JDMK/JMX, then combine these features with mobility and agency using Voyager. The performance of the integrated alternative revealed to be significantly better than that of basic RPC implementation with Voyager, RMI, CORBA, JMX and JFMX.

Through these implementations, we showed how a manageable system can be easily and rapidly instrumented using JFMX with a focus on testing and monitoring. We also showed how to use Mobile Agents to test and monitor Managed Resources instrumented in JFMX. Agency and mobility features are added at run-time. Neither additional efforts, nor development delay were added, and the scenario was successfully implemented with Mobile Agent mechanism without any change to the RPC based implementations.

The new Java Fault Management Extension (JFMX) will facilitate the automation of diagnostic tests, performance measurements and other fault management functions implementation using a fully distributed environment. Using Voyager objects/agents mobility and other dynamic features of JMX, the proposed API enhances significantly existing development environments by providing flexibility, scalability, efficiency, robustness, and lower operation cost.

7 References

[1] AdventNet, *AdventNet SNMP Release 3.0*, http://www.adventnet.com/

[2] M. Baldi, S. Gai and G.P. Pico, *Exploiting Code Mobility in Decentralized and Flexible Network Management*, Proceedings of the First Intl. Workshop on Mobile Agents, Berlin, Germany, April 1997.

[3] NicholasKassem, *Designing Enterprise Applications with Java2 Platform, Enterprise Edition.* Addition Wesley, June 2000.

[4] M. El-Darieby, A. Bieszczad, *Intelligent Mobile Agent: Toward Network Fault Management Automation*, in Integrated Network Management VI, *Distributed Management for the Networked Millenium*, Boston, USA, May 1999.

[5] Graham Glass, *Reducing Development Effort using ObjectSpace Orb*, CTO ObjectSpace, 1999.

[6] Guiagoussou Mahamat, Michel Kadoch, *Java Fault Management Extension (JFMX),* Technical Report, June 2000.

[7] ITU-T Recommendation X.745, *Diagnostic and Confidence Tests Categories*, Geneva 1992.

[8] ITU-T Recommendation X.737, *Test Management Function Management function*, Geneva 1992.

[9] Susilo, G., Bieszczad, A. and Pagurek, B. (1998), *Infrastructure for Advanced Network Management based on Mobile Code*, in Proceedings of the IEEE/IFIP Network Operations and Management Symposium NOMS'98, New Orleans, Louisiana, February 1998.

[10] David Wong, Noemi Paciorek, and Dana Moore, *Java-based Mobile Agents*, Communications of the ACM, Mars 1999.

[12] Jeff Nelson, *Programming Mobile Object with Java*^{TM}, Wiley Computer Publishing, 1999.

[13] Object Space Inc, *Voyager*, http://www.objectspace.com/home.htm

[14] Robert Orfali, Dan Harkey, *Client/Server programming with JAVA and CORBA*, Second Edition, Wiley Computer Publishing, 1998.

[15] OUTBACK Resouce Group, Inc. JSNMP Entreprise^{TM}, Java-Based SNMP Package User's Guide http://www.outbackinc.com/, 2000

[16] Sun Microsystems Corporation, *Java RMI Specification*, ftp://ftp.javasoft.com/docs/jdk1.1/rmi-spec.pdf.

[17] Sun Microsystems Corporation, *Java Entreprise Bean*: http://www.javasoft.com/products/ejb/index.html

[18] Sun Microsystems Corporation, *Java Dynamic Management Kit*, http://www.sun.com/software/java-dynamic/

[19] Sun Microsystems Corporation, *Java Management Extension*: http://java.sun.com/products/JavaManagementExtension/.

[20] Advanced Network Solutions, *Java*^{TM} *TL1 Translator 1.0*: http://www.ans.it/jtt/