CHAPTER 2

Introduction to JMX[1]

The Java Management Extensions (JMX) specification[2] defines a Java optional package for J2SE[3] that provides a management architecture and API set that will allow any Java technology–based or accessible resource to be inherently manageable. By using JMX, you can manage Java technology resources. You can also use Java technology and JMX to manage resources that are already managed by other technologies, such as SNMP[4] and CIM/WBEM.[5]

JMX introduces a JavaBeans model for representing the manageability of resources. The core of JMX is the simple, yet sophisticated and extensible, management agent for your Java Virtual Machine (JVM) that can accommodate communication with private or acquired enterprise management systems. JMX also defines a set of services to help manage your resources. JMX is so easy to use and is so suited for the Java development paradigm that it is possible to make an application manageable in three to five lines of code.

Basically, JMX is to management systems what JDBC (Java Database Connectivity)[6] is to databases. JDBC allows applications to access arbitrary databases; JMX allows applications to be managed by arbitrary management systems. JMX is an isolation layer between the applications and arbitrary management systems. So why do we need this layer anyway?

2.1 Why We Need JMX

2.1.1 Choosing a Management Technology

As we saw in Chapter 1, many different management technologies are being used in different areas of the industry. CMIP[7] dominates the telephony management market. SNMP dominates the device and network management market. Because this book is about developing Java applications and systems, let's narrow our focus to those technologies used by Java-based resources. Most Java-based resources today will be part of applications.

Even though SNMP is supported by some applications and middleware, it is not widely used for application management. One of the most commonly cited reasons for this is that many application vendors and management vendors have felt that the granularity of security in SNMP is not sufficient to use it for configuration updates and sensitive information. Therefore SNMP is often seen as useful only for *read-only* management of more or less public data and events. SNMP also does not have a natural model for operations on managed resources. Operations must be represented as a settable attribute. Sometimes this can be a difficult representation to map. Dependencies and associations can also be difficult to represent in SNMP.

CIM defines a more natural way to represent management data and addresses some of the weaknesses just described. It has extensive models for systems and devices, but the application management models are still emerging. The fact remains that there is no dominant management technology for the management of applications.

This would not be such a big problem if there were a single, dominant management systems vendor. If that were the case, you could use the management technology chosen by that vendor. Unfortunately, life is not that simple. Today the enterprise and application management market is pretty evenly split between Tivoli Systems[8] and Computer Associates,[9] who use their own proprietary technology for their manager-agent infrastructure.

If you have to manage an application or resource that runs on only one operating system, or on one vendor's systems, then choosing a management technology can be guided (or dictated) by the preferences of that vendor. Microsoft's Windows,[10] IBM's AIX,[11] Sun's Solaris,[12] and Hewlett-Packard's HP-UX[13] each has its own management system. However, one of the great things about Java has been the ease in which applications can be ported to and supported on many different vendors' systems. This means that most Java-based applications run on many platforms. If you are developing managed software products, you may be pressured to support multiple management

technologies and systems because every vendor will want you to make your Java resources manageable by its management system.

Just to add to the list of management technologies you need to support, customers may have installed enterprise management systems that they will want to use to manage your application. In fact, they may or may not buy your application, depending on whether or not it can be managed by their existing enterprise management software. Your customers cannot be expected to replace their existing enterprise management systems just to accommodate your application. If you supply your own management system, your customers may still not be happy. They may not want yet another management console to watch and understand just to manage your application. Adding another console mitigates the console consolidation benefits of their enterprise management systems.

If you are a developer or architect working for a vendor of cross-platform applications, you will find yourself between a rock and a hard place. The rock is that your marketplace may be demanding that your software be manageable. To appease that marketplace, the software will need support for multiple management technologies. The hard place is that the cost of developing support for one management technology is expensive. This cost includes the learning curve, design and development, and maintaining currency with those technologies as they continue to advance. The cost of developing support for multiple technologies may very well exceed the potential new sales. This makes the business case for creating manageable applications and systems very hard to maintain and contain.

As a result, the potential return on investment may not motivate you to instrument appropriately for manageability. In fact, you may choose to write your own application-specific management system to solve a particular problem quickly, and not implement it for any external management technology. You can see how this adds to the populations of unique, nonstandard management systems and unmanageable applications.

A single suite of uniform instrumentation for manageability, like JMX, makes it cost effective to develop new applications with management capability. You can use JMX to instrument your Java applications. You can also use JMX to provide access to the manageability capabilities of your non-Java applications via Java Native Interface (JNI) and wrappers. Because JMX is centered on an architecture for pluggable adapters that allow any management technology to manage your resources, you have the best of both worlds: instrumenting your application with one management technology, and being manageable by many different management systems.

2.1.2 Dealing with Diversity

One of the primary challenges in managing applications is their diversity. This diversity is also a challenge for developers! Applications today vary widely in purpose, size, architecture, and criticality. Very little is common across all application types. Application architecture trends are increasing the diversity rather than settling the industry on a few de facto standard approaches.

JMX can be used to enable management of a wide variety of application architectures. JMX allows you as a developer to build your skills on one management technology that you can then apply to lots of application projects, today and in the future. Using JMX to enable this variety of application types benefits management systems vendors as well. They can support JMX well and be able to manage a wide variety of applications. Some of the application types that JMX is suited for are centralized applications, distributed applications, Internet applications, e-business applications, and service-oriented applications.

2.1.2.1 Centralized Applications

Centralized applications, such as payroll and accounting, are backed by a database on a high-end server and usually accessed by a limited set of users, like a financial department. Managing centralized applications entails ensuring high availability and performance throughput because they can be a single point of failure. The clients of these distributed systems are usually other programs, which expect lightning-fast response times.

2.1.2.2 Distributed Applications

Distributed applications, such as mail systems, usually require groups of small and midrange server systems to be running at all times, and they are accessed throughout the enterprise. Managing distributed applications is often a scaling problem: Many, many servers must be managed (i.e., available, connected, and performing well), as well as the networks that connect them, in order to simply ensure the application is available to its users. Generally the clients require proprietary software, so client software distribution and configuration must be managed as well.

2.1.2.3 Internet Applications

The introduction of the intranet/Internet concept precipitated a new class of application that connects the end user to existing, traditional, centralized applications. The new application facilities range from Web-accessible corporate

personnel directories to Web-based order-tracking systems that benefit customers and reduce order management costs. These types of applications make it easier to access corporate information inside traditional applications, and they reduce the number of personal contacts necessary. Managing Internet applications entails keeping several layers of applications available to each other and the network: Web servers, application servers, back ends. Browsers must be appropriately configured.

2.1.2.4 E-Business Applications

The next generation of autonomous, Web-based applications is rapidly being developed and deployed in the new business environment. These applications embody e-commerce in the form of catalogs, shopping, marketplaces, and auctions. The move of the supply chain to the Internet will drive the next set of critical, distributed business-based applications. These applications move the heart of the business—buying supplies and selling products—to the Internet. Managing e-business applications is challenging because applications may span enterprise boundaries and use unreliable protocols like HTTP.

2.1.2.5 Service-Oriented Applications

Service-based architectures are currently emerging where IT resources across the network appear, move, and disappear. Relationships between applications can be just-in-time and fleeting. Managing the dynamic topologies, dependencies, and availability of these applications in this environment will be difficult at best.

Each of these classes of applications has its own management challenges. However, all of these application types execute across multiple hosts, operating systems, and corporations. They incorporate existing traditional and emerging application models. They are no longer just client-server, they are now client-middleware-server. These new application types are business critical, creating the need for them to be uniformly controlled and managed by a business's existing management systems with the same diligence as traditional applications. JMX is flexible and extensible enough to be taken advantage of in all of these diverse types of application architecture.

2.1.3 Being Managed by Multiple Management Applications

As we discussed in Chapter 1, there are many different types of management applications: distribution, inventory, topology, configuration, operations,

event, automation, monitoring, and performance. You will probably want your application to be managed by several of these. Without JMX, you might have to implement explicit support for each type of management application. If you're using JMX combined with JMX adapters, the same instrumentation can be used to support and interact with most or all of these applications.

2.1.4 Supporting Application-Specific Management Systems

Your application will need to be installed, configured, monitored, and maintained. This means that you must implement your own management system—that is, an application-specific management system—that supports these tasks. Otherwise, you will be dependent on a management system being available in your customer's environment. This dependency may limit sales if some of your customers have a different management system, or even no management system. Applications in the market today have their own proprietary instrumentations to communicate with their internal management systems, as well as instrumentation to connect to other management systems. We've seen how JMX can be used to interact with multiple management technologies and management applications. In the same vein, JMX can be used to drive an internal or application-specific management system, as well as an external management system.

2.2 Which Applications Should Be Manageable?

Not all Java applications and resources should be managed. Let's talk about some considerations for deciding whether or not you should enable your application to be managed by an external or third-party management system.

How extensive your application's manager is, and how well it is integrated with an enterprise management system, will depend on a number of different characteristics of your application. These include how critical your application is, how complex your application is, how scalable your application needs to be, what policy your corporation has on application management, and what your application's target market is.

2.2.1 Complex Applications

If your application has a very complex architecture or configuration, you may need a reasonably extensive and comprehensive management application. It doesn't necessarily need to be integrated with enterprise management

systems, unless they are mission critical or if several of them might be executing simultaneously. J2EE application servers[14] are a good example of mission-critical, complex applications that must have their own managers and be integrated with an enterprise management system.

2.2.2 High-Volume Applications

If your application will have a great many simultaneously executing installations in an enterprise, it may require distributed management support. Such applications are also excellent candidates for integration with the enterprise management system because they will benefit from both software distribution and centralized configuration. Because there will be so many images, operations staff will need topologies with status indicators and event management systems to distill the management information into events they can take action on.

2.2.3 Mission-Critical Applications

If your application is mission critical, you are going to need availability and performance management. Your application may be mission critical if it is an integral part of a business-critical system, like a database, router, application server, or name server. Mission-critical applications may be singletons or one of many instances. Your mission-critical application will need to have its own manager and be very well integrated with the enterprise management system. It must have very good availability and performance monitoring, with status reflected in a topology application.

2.2.4 Corporate Applications

If your application is being developed for use in a specific organization, corporate policies may dictate the degree of management support that you need to develop for the application. If the application is being developed for an organization that has invested in an enterprise management system, certain business policies may govern which applications are supported by the enterprise management system, and how extensive that support must be.

2.2.5 Applications with Expectant Customers

If your application is being developed to sell, you must consider the needs and expectations of your target market. Some applications have a target customer base that expects application management. These customers may buy

only manageable applications. For example, owners of mainframes will expect applications that execute on those mainframes to be integrated with the management systems on those mainframes. If you are developing an application intended to execute on a mainframe, you must plan on enabling it to be well managed by the appropriate management application. If a large percentage of your target market uses a particular enterprise management system, then your application should be supported by the same system. If your target market expects that one or more enterprise managers will support your application, you will want to be sure that your application satisfies those expectations.

To summarize, if your application has a large, diverse target market, you may be forced to support multiple management systems on multiple platforms. This diversity can lead to significant, perhaps unsupportable, development costs. Developing and maintaining support for any one of the standard or proprietary management technologies requires you to tackle a steep learning curve and commit to a significant development effort. JMX mitigates your investment in management during application development by allowing you to concentrate on supporting one management technology: JMX. Multiple enterprise management systems can interact with the JMX agent in order to manage your application.

2.3 The Goals of JMX

There has been incredible growth in the number of mission-critical Web-based and Java applications being developed. The Web applications bring new business to the enterprise. Java e-business applications actually conduct business transactions on the Web, including catalog and shopping cart applications. E-business applications that generate income for the enterprise become mission critical in a hurry. When these applications are not available or responding quickly, customers shop elsewhere and income is lost. Therefore, enterprises are demanding end-to-end application manageability for e-business applications. The manageability requirement applies to the device, system, network, middleware, and any other applications that the e-business applications depend on.

In order for Java applications to make this transition from "cool" Web applications to "critical" e-business applications, they must be well behaved, reliable, and manageable. These requirements, along with the need for end-to-end management of these applications, mean that customers and developers will demand that products be manageable out of the box. These products

include application servers, distributed application infrastructures, component toolkits, and the applications themselves.

Ideally, development tools and environments would provide wizards and tools to make it easy to develop manageability along with the product. These tools need a standard, portable, flexible API in order to be able to generate or modify applications so that they are manageable.

JMX defines a set of APIs to address management of and through Java. Three main influencing forces on the success of JMX are (1) having a simple API for the developer, (2) ensuring enough information for management systems to manage the resource "generically," and (3) providing an architecture to support the management of diverse, dynamic applications. Figure 2.1 shows the relationship of the JMX MBeanServer to the resources it manages and the management systems it communicates with. The following paragraphs will explain further.

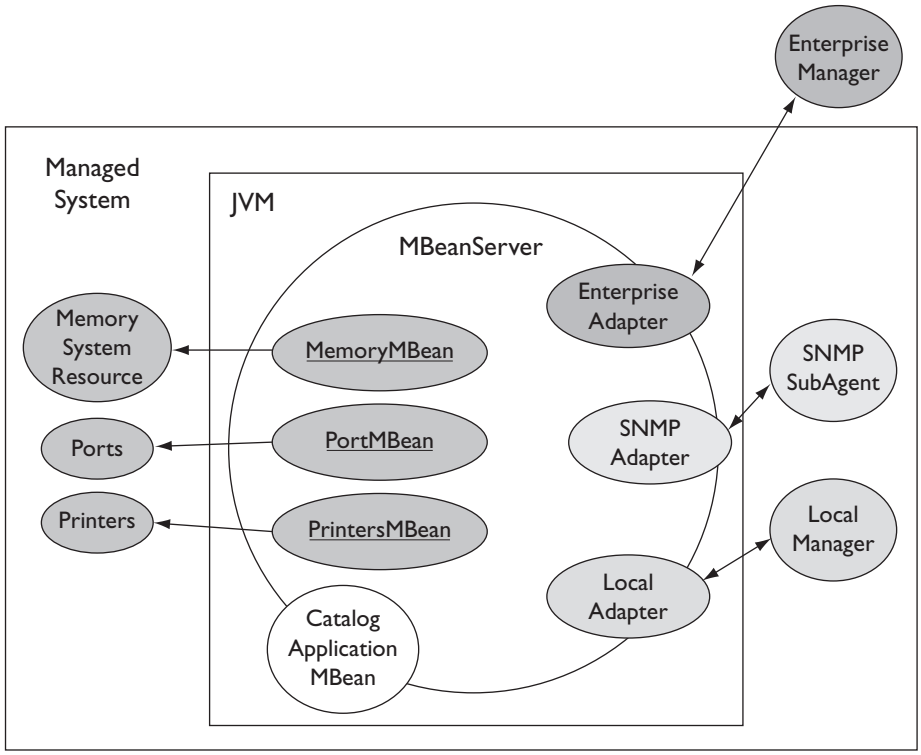

Figure 2.1 JMX Overview

2.3.1 Simple API

JMX provides a simple, straightforward management API for Java application developers. You can use this API to bring attributes, operations, and notifications to management systems from within your application. The API is easy to understand because it is similar to one you already know: JavaBeans. JMX uses Management Beans (MBeans) to represent the management interface for the managed resource (see Section 2.5.2). As you can see in Figure 2.1, the MBeans interact with the resources they manage: memory, ports, and printers. You can use your existing JavaBeans as MBeans, like the catalog MBean in Figure 2.1, or you can easily develop new ones as facades for your existing Java applications.

2.3.2 Dynamic Management

JMX provides a "management container" called an *MBeanServer*. The MBeanServer is a registry and a "traffic cop" in that it is always between the MBeans and their users. You can see this relationship in Figure 2.1. You can register your MBean with the MBeanServer at any time. You can also deregister your MBean at any time. Therefore, the MBeanServer represents an accurate inventory of all the resources to be managed. If resources are permanently removed from the system, you can remove them from JMX without reconfiguring or recycling the MBeanServer. JMX also provides for management services to be registered as MBeans. This means that additional functionality can be added or removed at any time. This flexibility makes JMX a dynamic and easily extensible architecture to use for management.

2.3.3 Isolation

The MBeanServer acts as the isolation or mediation layer between your application and management system vagaries, including protocols, behavior, platforms, and so on. The management system is encapsulated away from the managed resources by JMX adapters. JMX adapters, as seen in Figure 2.1, translate from the MBeanServer's MBeans to the management systems' native protocols and object models, and back again.

In fact, these adapters don't have to be provided by the management system; they could be developed by anyone. If the adapters communicate with a local management agent on the system, like an SNMP agent, then the management system will never "see" JMX. This means that the management system may not be aware if JMX is providing access to the manageability of your application.

2.3.4 Generic Management

Many current management systems require a custom module to be written for every application they are to manage. This module integrates the application's management into the management system. One of the goals of JMX was to provide sufficient information to the management systems so that management support could be generated or provided generically (i.e., without developers having to create custom integration modules for their applications for every management system). Ideally, the management system would provide a tool, along with its adapter, that would generate these integration modules if they were needed.

When information is being provided to management systems, either a management model or metadata must be supplied. JMX is a model-less management architecture; it deliberately does not define an information model or the structure of management information (SMI) like the IETF did for its Network Management Framework.[15] Instead, JMX provides metadata in `MBeanInfo` objects associated with every MBean. Management system adapters and tools can use the information in the `MBeanInfo` object to create integration modules or feed dynamic management architectures.

2.4 History

This section presents a historical perspective on JMX, covering the precursor specifications and products, Sun's JMAPI and JDMK, the JMX JSR and current specifications, compliance levels, and a review of some of the current implementations.

2.4.1 JMAPI

Java Management API (JMAPI) was an effort led by Sun. It was intended to arrest the proliferation of management platforms by developing a standard Java management platform infrastructure. There were two parts to JMAPI:[16] the Admin View Module and the Admin Runtime Module. The Admin View Module was an infrastructure to be used to build management consoles. The Admin Runtime Module was an API and infrastructure to be used to build management systems and applications. Most of the management system vendors were involved, including Tivoli and Computer Associates. An early implementation of this specification was developed and used by a few vendors. In late 1998 this work stalled. An "arms race" developed among the vendors to get their management agents on the most systems. JMAPI was not

focused on helping developers make their resources manageable. But that is precisely where JMX is focused.

2.4.2 JDMK

Java Dynamic Management Kit (JDMK) 2.0[17] was a Sun product that was the precursor to JMX and provided the starting point for the JMX specification. Today it is Sun's product version of JMX. The JMX Reference Implementation is independently licensed and contains quite a bit of additional functionality that is not in the specification. It contains a remote JMX manager, tools to support the creation of MBeans from SNMP MIBs, and Java SNMP manager APIs.

2.4.3 JMX

Sun opened JSR 3[18] in December 1998 using its then brand-new Java Community Process.[19] JMX's initial name was Java Management API (JMAPI) 2.0, although it had no relationship to the first JMAPI's goals, APIs, or implementation (see Section 2.4.1). The first action of the JMX Expert Group was to change the name to Java Management Extensions to eliminate confusion between JMX and JMAPI.

The expert group consisted of Sun, IBM/Tivoli, Computer Associates, Groupe Bull (Evidian),[20] TIBCO,[21] and Powerware.[22] Eventually Borland,[23] Motorola,[24] BEA,[25] IONA,[26] Lutris,[27] and JBoss[28] also joined, as it became obvious that this new technology could be very relevant to J2EE application servers. The interesting thing about this expert group was its cross-industry mix. Not only were enterprise management system vendors represented, but so were telecommunications device and system vendors, as well as application server vendors. This diversity created a very important balance in interests and a willingness to focus on the needs of the Java resource developer rather than the management system.

The initial specification contribution from Sun was based on its Java Dynamic Management Kit (JDMK) 3.0[29] product, which was gaining some following in the telecommunications industry. Sun intended the next release of JDMK (4.0)[30] to be the first JMX-compliant product.

The JMX mailing list is jmx-forum@java.sun.com, and the Web site is http://java.sun.com/products/JavaManagement. The JMX Reference Implementation and Technology Compatibility Kit (TCK) are available from Sun Microsystems through this same site.

Sun was the specification lead and is now the maintenance lead. Here are some of the JSRs that pertain to JMX:

- **JMX 1.5**.[31] JSR 160: "Java Management Extensions (JMX) Remoting 1.2" (http://www.jcp.org/jsr/detail/160.jsp), led by Sun Microsystems. This specification extends the JMX 1.0 specification by adding distributed capabilities to support remote JMX managers, remote MBeans, and JMX agent discovery. At the time of this writing, this expert group is actively working on a new specification that should be available in 2002.
- **JMX and CIM/WBEM**.[32] JSR 146: "WBEM Services: JMX Provider Protocol Adapter" (http://www.jcp.org/jsr/detail/146.jsp), led by Sun Microsystems. This specification defines how JMX instrumentation can be mapped to CIM and provides the definition of a JMX provider protocol adapter for WBEM Services. This JSR provides a bridge from JMX into WBEM though a JMX adapter. Although an expert group has formed, little progress is being made on this specification and there are currently no reliable availability dates.
- **JMX and TMN**.[33] JSR 71: "JMX-TMN Specification" (http://www.jcp.org/jsr/detail/071.jsp), led by Evidian. This specification specifies interoperability between the Telecommunication Management Network (TMN) standards and JMX. This JSR defines bidirectional integration between JMX and TMN. In the end, a JMX-manageable application would be manageable by a TMN manager or agent. Likewise, a JMX manager would be able to manage a TMN environment. This JSR was withdrawn in June 2001.
- **JMX and IIOP**.[34] JSR 70: "IIOP Protocol Adapter for JMX Specification" (http://www.jcp.org/jsr/detail/070.jsp), led by IONA. This specification will establish an IIOP-based[35] adapter for the JMX agent, to allow CORBA[36] clients to access JMX agents. This specification will allow non-Java environments, such as CORBA applications, access to JMX information using IIOP. This expert group has been formed; however, when a specification and reference implementation will be available is unknown.
- **JMX and SNMP**. A JSR for an adapter from JMX to SNMP agents was discussed frequently within the JMX Expert Group, but the JSR was never opened. There is quite a bit of SNMP support available in the JMX Reference Implementation, and from products such as Sun's JDMK and AdventNet.

Figure 2.2 shows how the adapters being defined by these JSRs are related to the JMX MBeanServer and the original management systems. The Mof2MBean and MIBGen tools take other definitions of management objects (for CIM and SNMP, respectively) and generate JMX MBeans to match them. The data going over the network is in the native management system's format.

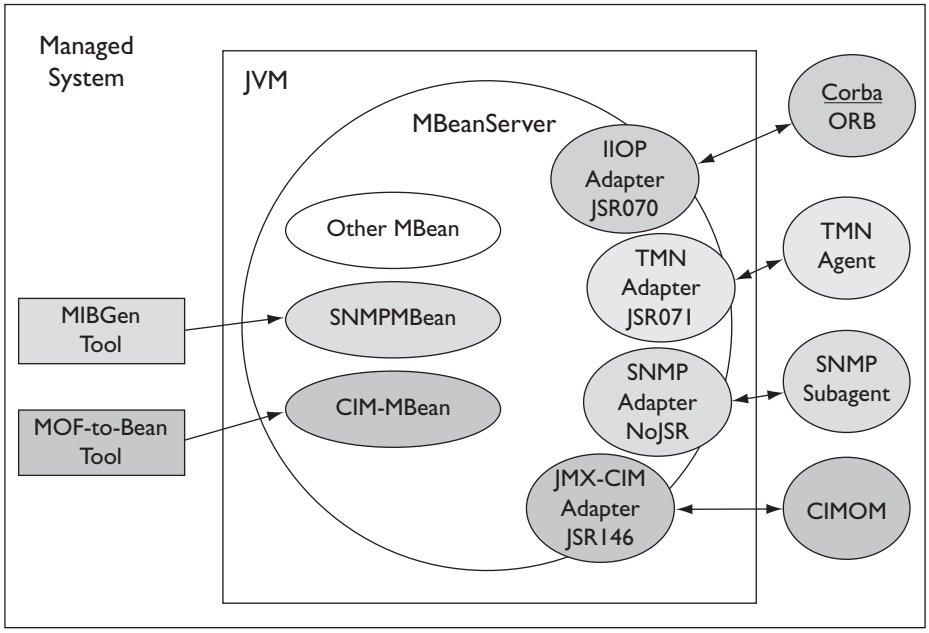

Figure 2.2 JSR Adapters

Other JSRs use or reference JMX as well, including the Java Integrated Networks (JAIN) JSRs and "J2EE Management" (JSR 77).[37]

2.4.4 The Specification and Compliance

The final *Java Management Extensions (JMX) v1.0 Specification* is available from http://jcp.org/aboutJava/communityprocess/final/jsr003. This specification was narrowed to define only the agent and instrumentation layers. The JMX 1.1 maintenance release specification and reference implementation are available at http://java.sun.com/products/JavaManagement. This release fixed specification ambiguities and reference implementation problems. Future versions of the specification (hopefully JMX 1.5) will address the distributed management layers.

The agent layer defines the MBeanServer and service MBeans that cooperate to implement the agent role in the manager-agent architecture. The instrumentation layer consists of the MBeans that are used by resources to expose their manageability.

The manager layer in the original JMX specification included Java APIs for communicating with agents and managers that are based on existing management technologies. These APIs do not communicate with JMX MBeanServers directly. In order for these APIs to communicate with an MBeanServer, an adapter to communicate with the native management technology would need to be loaded. For example, if you had an MBean you wanted to access using the Java SNMP manager APIs, you would communicate with an SNMP agent, which would communicate with an SNMP subagent that is also a JMX adapter, which would communicate with the MBean. This chain of communication is illustrated in Figure 2.2. The JMX Expert Group considered three Java APIs for managers, the Java API for interacting with SNMP agents, the Java API for interacting with a CIMOM using WBEM, and the Java API for interacting using TMN managers.

Some of these "manager APIs" in the original specification were spun off into separate JSRs so that they would have their own specification and reference implementation because they had no direct dependency on the JMX agent or MBeanServer. Figure 2.3 shows the relationship of these manager

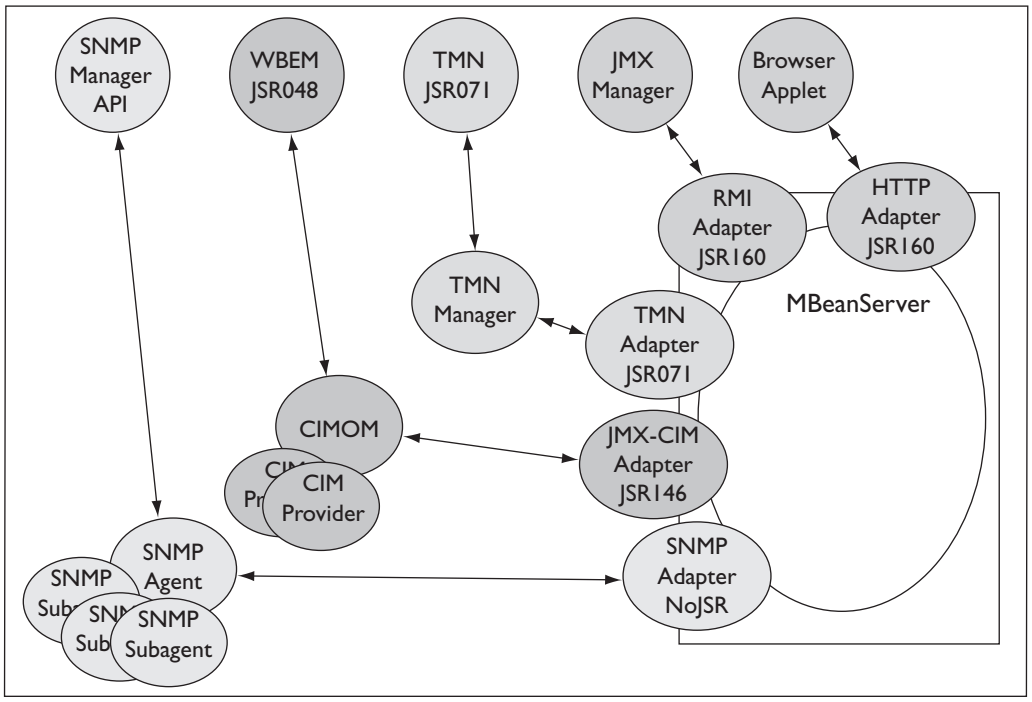

Figure 2.3 Manager-Level API JMX JSRs

JSRs to the MBeanServer. The JMX Java APIs for WBEM specification and libraries became JSR 146.[38] The JMX Java APIs for TMN became JSR 71.[39] The JMX Java APIs for SNMP are not associated with a JSR. However, JDMK does ship the SNMP Manager Java APIs.

JMX *compatibility* is defined to be an implementation of some or all of the JMX 1.0 specification that has not passed Sun's JMX TCK test suite. JMX *compliance* is defined in levels. These levels are defined in terms of what parts of JMX are supported. We will cover these terms and components in depth throughout this book, so if you are new to JMX, this section will make more sense when you revisit it.

2.4.4.1 Compliance at the Instrumentation Level

Resources and applications that want to be managed through JMX and claim JMX compliance must implement an MBean and provide for its registration with an MBeanServer. The application can provide any of the four types of MBeans: standard MBean, dynamic MBean, open MBean, or model MBean.

The JMX Technology Compatibility Kit (TCK) does not test for accurate compliance of MBeans. But the JMX Reference Implementation does come with an MBean verifier.

2.4.4.2 Compliance at the Agent Level

Implementers of JMX agents can be JMX agent-level compliant if they implement the MBeanServer and the four required services: monitoring, timing, relation, and class loading (MLet). There is a formal compliance test Technology Compatibility Kit from Sun Microsystems that an implementation must pass to be declared compliant. If you implement the entire JMX specification and do not pass the compliance tests, you are JMX *compatible*.

2.4.4.3 Support at the Agent Level

There is also the concept of providing JMX support at the agent level. This means that the agent supports managing resources through MBeans, but not necessarily as an MBeanServer. There is no formal qualification or test for JMX support. This level of support was specified to allow existing proprietary product agents to detect and support new management objects, MBeans, without exposing their agents as MBeanServers to adapters.

2.4.4.4 Compliance at the Distributed Services Level

This level of JMX has not yet been defined. The recently opened JSR "Java Management Extensions (JMX) Remoting 1.2" (JSR 160) is considering defining this level. The distributed services level would define how JMX agents work cooperatively across a distributed network, as well as how a manager would discover and interact with groups of agents and cascaded agents.

2.4.4.5 Compliance at the Management Level

JMX was intended to be an umbrella JSR and include a set of manager-side management APIs. The previous three levels describe JMX for instrumentation and management. The management APIs define Java APIs for management of other existing management technologies using Java. Currently Java APIs are being defined for SNMP, WBEM, and TMN. These Java APIs will be defined in other JSRs.

2.4.5 The Reference Implementation

Version 1.0 of the JMX Reference Implementation from Sun Microsystems was released in December 2000. The reference implementation source code[41] is licensed under the Sun Community Source License (SCSL).[41] The binary version of the reference implementation is available for free. The JMX Technology Compatibility Kit (TCK)[42] must be purchased from Sun, and the price can be substantial. Vendors who implement the JMX agent specification must purchase and pass the TCK in order to claim JMX compliance.

JMX 1.1, a maintenance release, is available from the same Web site. For more information and access to the source or binary versions of the reference implementation, see http://java.sun.com/products/JavaManagement.

Sun also provides a Web site called the JMXperience at http://java.sun.com/products/JavaManagement/JMXperience.html, where Sun and other vendors can contribute interesting tools, MBeans, and components for JMX. Sun has contributed a remoting component that provides a remote MBeanServer for use by JMX clients (usually JMX managers) and an Mof2MBean tool that generates MBean skeletons from CIM MOF (Managed Object Format) files.

The appendix of this book lists a short summary of some of the JMX agent implementation vendors, including Sun's JDMK, Tivoli's TMX4J, AdventNet's Agent ToolKit, and MX4J. The appendix also lists JMX manager vendors, including those from Tivoli, Dirig Software, AdventNet, and

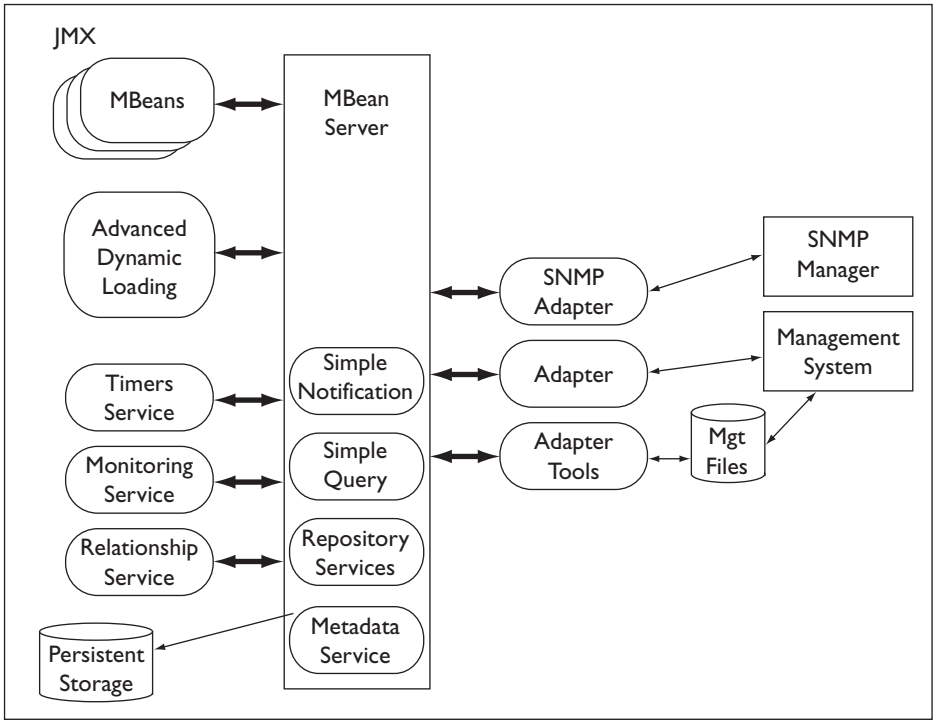

Figure 2.4 JMX Architecture

Jamie. In addition, the appendix lists some of the JMX manageable products, including WebSphere, WebLogic, iPortal, JBoss, SonicXQ, hawkEye, and Pramati Server.

2.5 JMX Overview

In its instrumentation and agent specification, the JMX architecture consists of four components (see Figure 2.4) that map to the management system components discussed earlier:

1. **Managed resources and management beans** (**MBeans**) that expose the management interfaces that make up the instrumentation level.
2. **Agents**, including MBeanServers and the monitoring, timing, relation, and class-loading services, that constitute the agent level.
3. **Adapters** that translate between JMX capabilities and APIs, and between their respective management systems.

4. **Adapter tools** that generate the files required by the management system through JMX APIs. Examples of these file types are Tivoli's AMS file,[43] SNMP's MIB file,[44] and CIM's MOF file.[45]

Let's take a closer look at each of these elements.

2.5.1 JMX-Managed Resources

JMX-managed resources are instrumented with management beans (MBeans). You, the application developer, use MBeans to expose the management interface of the managed resource. The management interface of a resource consists of the attributes, operations, and notifications that are used to manage it.

2.5.2 MBeans

JMX specifies four types of MBeans: standard MBean, dynamic MBean, open MBean, and model MBean. Each MBean has metadata in `MBeanInfo`. `MBeanInfo` defines the attributes, operations, and notifications supported by the MBean.

2.5.2.1 Standard MBeans

A *standard MBean* can be any JavaBean or JavaBean-style program that has been registered with the MBeanServer. You, the application developer, define these MBeans and their management interfaces at development time. The only requirement is that your MBean class must implement a Java interface named `classnameMBean` that you must define for them. For example, an MBean class named `CatalogManager` would implement an interface called `CatalogManagerMBean`. This interface defines the management interface for the MBean. The interface might include all or just a subset of the methods in the actual MBean class.

When your MBean is registered with the MBeanServer, the MBeanServer will create an `MBeanInfo` metadata object for it by introspecting the `classnameMBean` interface. It will create attributes and operations in the MBean by looking for the JavaBean pattern. Each `getAttributeName()` method with a matching `setAttributeName(attributeValue)` method will create an attribute for the MBean. All other public methods will create operations for the MBean. The MBeanServer will use `MBeanInfo` to make sure only the attributes and operations you have exposed are invoked on your MBean.

Standard MBeans can be useful if your application already has management-oriented classes to support its own manager. You can simply register these instances with the MBeanServer after minor modifications to add "implements MBean." Chapter 3 explains standard MBeans thoroughly.

2.5.2.2 Dynamic MBeans

Dynamic MBeans allow your application or domain-specific manager to define or generate the management interface for your resource at runtime. This provides a simple way for you to wrap existing nonbean-style or even non-Java resources.

A dynamic MBean can be any Java class that implements the dynamic MBean interface. The dynamic MBean interface is similar to the CORBA Dynamic Invocation Interface (DII).[46] The most important thing to remember about dynamic MBeans is that they must provide their own management interfaces during runtime by maintaining their own `MBeanInfo` objects. The MBeanServer requests `MBeanInfo` from your dynamic MBean whenever it needs that information. This means that a `classnameMBean` interface is not used by the MBeanServer to create an `MBeanInfo` object. Your dynamic MBean is also responsible for implementing and validating correct invocation of the interfaces it defines in `MBeanInfo`. The MBeanServer delegates invocations of `getAttribute()`, `setAttribute()`, and `invoke()` directly to your dynamic MBean. Your dynamic MBean satisfies the request and returns it to the MBeanServer.

You can develop dynamic MBean implementations for your applications directly, have them generated, or allow a management application developer to create them. You can also develop an MBean service that will instantiate or generate the dynamic MBeans.

The next two types of MBeans we will look at are standardized types of dynamic MBeans: open MBeans and model MBeans. However, the fact that JMX defines standard types of dynamic MBeans does not mean that you cannot write your own dynamic MBeans, nor does it in any way restrict how you implement them. See Chapter 3 for a more detailed discussion of diagnostic MBeans.

2.5.2.3 Open MBeans

Open MBeans are dynamic MBeans that are restricted to accepting and returning a limited number of data types. If you use open MBeans and these basic data types, you eliminate the need for class loading. Removing the need for class loading can make it easier for you to deploy the MBean and support a highly distributed system. However, it does not remove the need for your application or a management application to understand the semantics of the data to be passed or returned. The open Mbean data types that are allowed are

- **Primitive data types**: `int`, `boolean`, `float`, `double`, and so on

- **Class wrappers for primitive data types**: `Integer`, `Boolean`, `Float`, `Double`, `String`, and so on
- **Table**: An array of rows of the same type
- **Composite**: An object that can be decomposed into other open data types

Your open MBeans must return an `OpenMBeanInfo` object from the `getMBeanInfo()` method. `OpenMBeanInfo` extends `MBeanInfo`, adding some additional metadata that you may supply, such as legal values, and default values. The open MBean support is optional in JMX 1.0, and the classes are not available in the current Sun JMX 1.0 Reference Implementation. See Chapter 3 for an explanation of open MBeans.

2.5.2.4 Model MBeans

A *model MBean* is an extension of a dynamic MBean. However, where you *must* write all of a dynamic MBean, you don't have to implement the model MBean. A model MBean is more than a set of interfaces; it is a customizable, standardized, dynamic MBean implementation. An implementation class of a model MBean named `RequiredModelMBean` must come with the JMX agent. This model MBean instance is immediately useful because your application can instantiate and customize a `RequiredModelMBean` instance with its own management interface information. This reuse of an existing implementation drastically reduces the amount of code you need to write to achieve manageability, and it protects your resource from JVM version and JMX agent implementation variances. Using `RequiredModelMBean` instances can allow your managed resources to be installed in a range of JVMs, from embedded environments to enterprise environments, without affecting your instrumentation.

The MBeanServer functions as a factory and delegator for `RequiredModelMBean` instances. Because `RequiredModelMBean` instances are created and maintained by the JMX agent, the `RequiredModelMBean` class implementation can vary, depending on the needs of the environment and the JVM in which the JMX agent is installed. An application that requests the instantiation of a `RequiredModelMBean` object does not have to be aware of the implementation specifics of a `RequiredModelMBean` class. The `RequiredModelMbean` class is responsible for implementing and managing the implementation differences between JMX and JVM environments internally. These differences may include persistence, transactional behavior, caching, performance requirements, location transparency, and remotability.

Because the `RequiredModelMBean` model MBean implementation is provided by the JMX agent, your application does not have to implement `RequiredModelMBean`; it just needs to instantiate it, customize it, and use it.

Your instrumentation code is consistent and minimal. Your application gains the benefit of support for and default policies concerning logging events, data persistence, data caching, and notification handling. Your application initializes its `RequiredModelMBean`'s `ModelMBeanInfo` with its identity, management interface, and policy overrides.

You can add custom attributes to the model MBean during execution. Your application-specific information can be modified without interruption during runtime. The `RequiredModelMBean` instance then sets the behavior interface for the MBean and does any setup necessary for event logging and handling, data persistence and currency, and monitoring for your application's model MBean instance. The model MBean default behavior and simple APIs will satisfy the management needs of most applications, but they will also allow complex application management scenarios. More details on the model MBean are given in Chapter 4.

2.5.3 JMX Agents

Your managed resources will communicate data and events to management systems with their MBeans through the JMX agent. JMX agents consist of an MBeanServer and a set of service MBeans.

2.5.3.1 The MBeanServer

The MBeanServer runs in the JVM local to the managed resources' MBeans. The MBeanServer is a registry for MBeans. It is also a repository of the current set of MBean names and references, but it is not necessarily a repository for your MBeans. The MBeanServer provides a query service for the MBeans. Upon a query, it returns the names of the MBeans, not the references. Because only names are returned, all operations on all MBeans must go through the MBeanServer. The MBeanServer acts as a delegator to the MBeans, returning the results to the requester.

The MBeanServer can be a factory for any MBean, even those you create. You have the option of instantiating MBeans directly and then registering them, or having the MBeanServer return an instance of the MBean to your application. The MBeanServer should always be a factory for `RequiredModelMBean` instances. The MBeanServer also provides access to the metadata about the MBeans in the `MBeanInfo` instance. The metadata includes the attributes, operations, and notifications provided by the MBean. The MBeanServer provides notification registration and forwarding support to MBeans representing adapters, services, and resources. The MBeanServer is discussed thoroughly in Chapter 5.

2.5.3.2 Required Services

The JMX agent includes a set of required services: the monitoring service, the timer service, the relation service, and the MBean class loader. Services are MBeans registered with the MBeanServer that provide some generic functionality that can be used by MBeanServers, MBeans, and adapters. Additional management services can be added dynamically as service MBeans by applications or management systems, making the JMX agent flexible and extensible.

- The **monitoring service** runs monitoring MBeans on a scheduled basis. It must support basic monitoring MBeans, including `Gauge`, `Counter`, `StringMatch`, and `StateChange`. Additional or specialized monitoring MBeans can also be developed and used.
- The **timer service** executes an operation on a timed basis. It is used by the monitoring service.
- The **relation service** supports relationship MBeans. Relationship MBeans contain the names of a set of MBeans that are related in some way. Some kinds of possible relationships include "contains" and "depends on."
- The MLet (management applet) service is an **MBean class-loading service** that loads an MBean across a network when an `MLET` tag in an HTML page is encountered.

These services are covered in more detail in Chapter 7.

2.5.4 JMX Adapters

Adapters communicate between the JMX agent and their corresponding management systems. The adapter is responsible for translating from JMX MBean types to its manager's types and taking care of any remoteness issues. Because the adapter can be implemented to mimic the manager's supported agent technology, the management system may not even be aware that JMX is in the picture. Generally, there is at least one specific adapter for each management protocol or technology required to support different management systems.

Adapters are also MBeans, and they are registered with the MBeanServer. Given that this is the case, it is possible to find out all of the adapters that are currently registered with an MBeanServer. Because adapters are MBeans, they can register for JMX notifications from the MBeanServer or other MBeans. In this case the adapter would have to implement the `NotificationListener` interface, and it might provide a filter to limit the notifications it receives.

Because the MBeanServer returns only names of MBeans and not instances, adapters invoke methods on MBeans only through the MBeanServer. The

type of MBean that represents the resource does not affect how the adapter invokes operations on the MBean. The type of MBean does affect how much data is available to the adapter in `MBeanInfo`. Adapters use the MBeanServer's interface directly. Adapters are responsible for "translating" JMX management information to their native representation of the management information, so JMX provides some hints on how to do that translation using `ProtocolMap` instances for attributes in model MBeans. Common, though nonstandard, adapters are RMI,[47] HTTP,[48] and SNMP.[49] CIM and IIOP adapters are in the process of standardization. Adapters are discussed in depth in Chapter 5.

2.5.5 Adapter Tools

Adapter tools typically accompany a particular adapter for a particular management system. Adapter tools will interact with the MBeanServer to create any files to represent the available management data in the MBeans in a format that the management system can consume. For example, an SNMP adapter tool might create a MIB file from the available `MBeanInfo` instance. This MIB file would be used by an SNMP management system to represent the management data on its console.

2.6 Quick Tour of JMX

The easiest way to understand JMX is with a simple but thorough example. This example uses Tivoli's TMX4J JMX implementation, but the code should be identical regardless of who the JMX vendor is. The remainder of this chapter is dedicated to demonstrating the major features of JMX—MBeans, the MBeanServer, monitors, and notifications—via the design and implementation of a simple manageable server application.[1]

2.6.1 todd, the Time of Day Daemon

todd is a simple daemon that does a simple job. It accepts client connections and sends them the current date and time whenever they ask for it. Figure 2.5 shows todd's static structure.

Three classes form todd's core: `Server`, `Listener`, and `Session`. The `Server` class is responsible for starting a listener and assigning sessions to

1. Please note that whenever you see ". . ." there is code missing for illustrative purposes. Complete code examples are available at http://www.awprofessional.com/titles/0672374083.

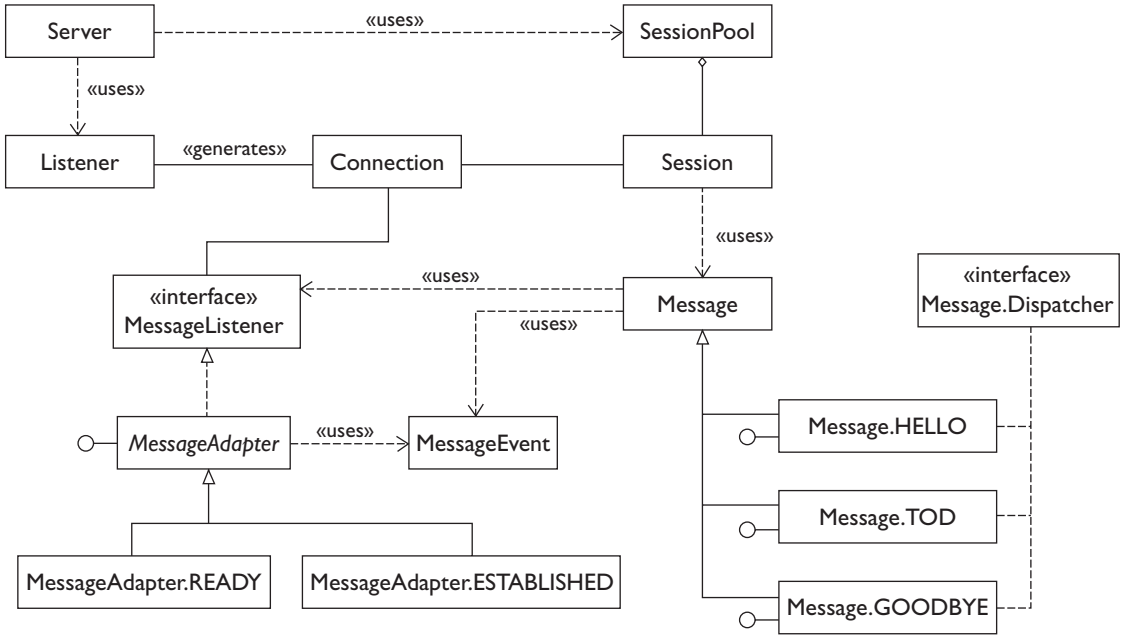

Figure 2.5 Class Structure of the Unmanaged Time of Day Daemon

incoming connections. The `Listener` class accepts incoming connections and queues them for the server. The `Session` class is responsible for responding to client messages received via a connection. Figure 2.6 illustrates the interaction among todd's classes.

2.6.2 todd Management

todd satisfies its basic functional requirements by accepting client connections and supplying the current date and time on demand, but it's not *manageable*. We have no way of knowing the total number of sessions that todd has handled, how many sessions are active, how many connections are queued, how long each session has been running, and so on. We can't stop and start the listener without killing todd, and any active sessions along with it. We can't change the size of the connection queue or session pool without recompiling the code. If todd were to become a popular service, our administrator colleagues would not be happy with us.

JMX lets us build manageability into our Java applications. MBeans capture the *management interface* of the resources that management systems

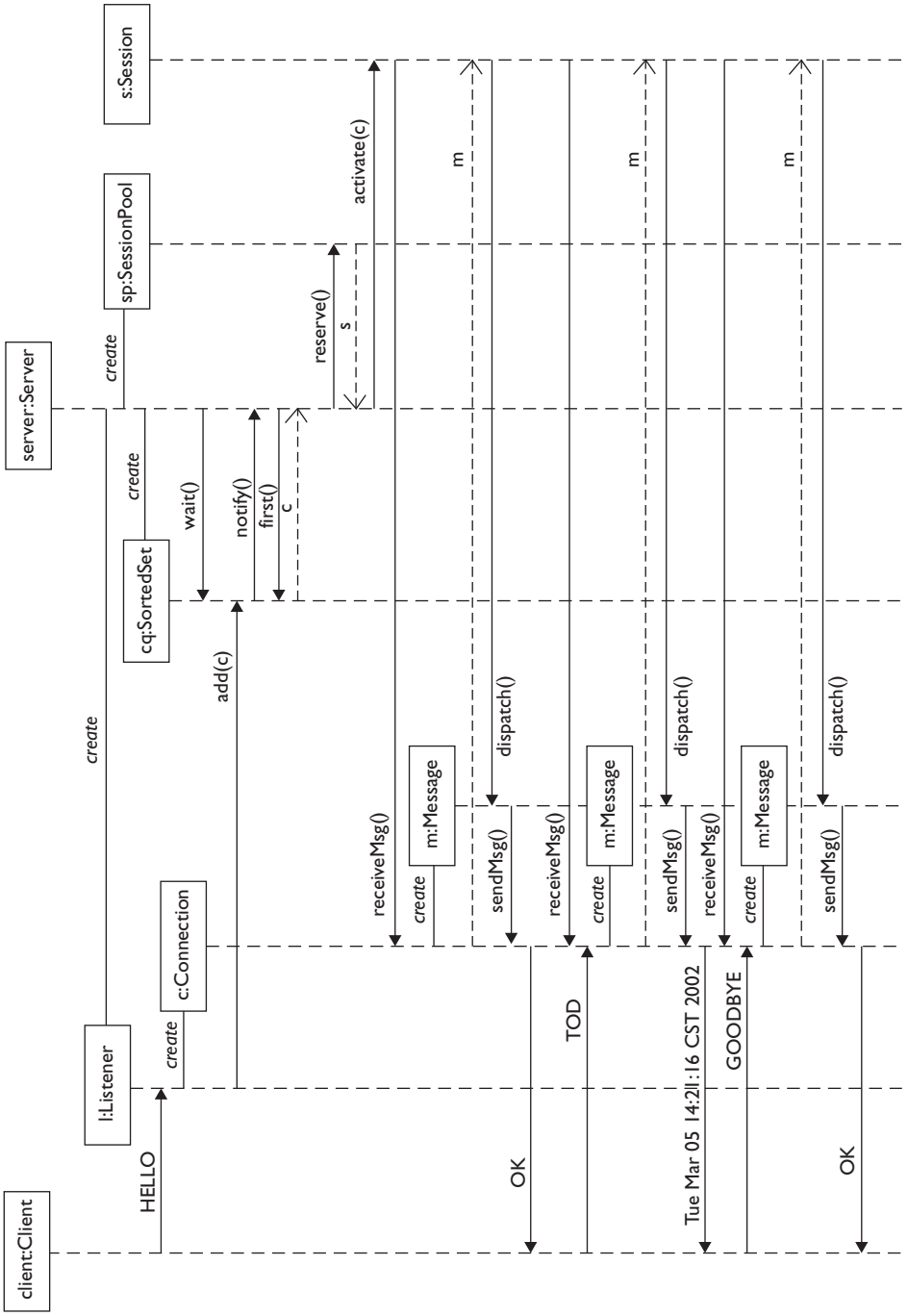

Figure 2.6 todd Component Interactions During a Client Session

need to monitor and control. The MBeanServer provides a common registry and naming model for all of an application's MBeans. MBean services exist to autonomously monitor MBean attribute values and fire notifications when constraints are violated. Notifications provide both a means to alert an administrator of a problem, and an implicit invocation mechanism that developers can use to make an application self-managing.

In the next few sections we will apply JMX to todd and transform it into a manageable application that will delight the system administration staff, or at least keep them from paging us in the middle of the night. Because the goal of this tour is to demonstrate the major features of JMX in an application, we will introduce them without a lot of explanation. Subsequent chapters will provide details of all the features we use in our example.

Before we start, it would be good to have an idea of the sort of manageability requirements the new version of todd has to satisfy. Table 2.1 provides a minimal list. Modifying todd to satisfy these requirements will involve the following activities: designing and implementing the necessary MBeans, incorporating an MBeanServer into todd, and wiring up the monitors and `NotificationListener` instances that are needed to complete the implementation.

Table 2.1 Basic todd Management Requirements

Aspect	Requirement	Description
Server control	Stop/start	Stop and start the server without killing the todd process or any active sessions.
	Shutdown	Shut down the server, killing the todd process and any active sessions.
Server data	Total connections	Show how many connections the server has handled so far.
	Uptime	Show how long the server has been running.
	Active sessions	Show how many sessions are currently active.
Session pool management	Grow pool	Increase the size of the session pool by a specified amount.
	Empty pool management	Stop the server when the pool becomes empty; restart the server when the pool contains at least two sessions.

2.6.3 todd's MBeans

From the requirements in Table 2.1, it's pretty clear that we're concerned with only a couple of todd's resources: the server itself, and the pool of sessions that todd uses to service connections. Because JMX uses MBeans to represent managed resources, it follows that we need to make the `Server` and `SessionPool` classes MBeans. There are several ways to accomplish that goal; the simplest is to make them standard MBeans. In general a standard MBean for a given resource is defined by a Java interface named `MyResourceMBean` and a Java class, `MyResource`, which implements the `MyResourceMBean` interface. `MyResourceMBean` defines the MBean's *management interface*—that is, the attributes and methods that JMX makes available to management applications.

It's clear from the requirements what the server's management interface should look like:

```
public interface ServerMBean {
  void shutdown();
  void start();
  void stop();
  Integer getConnections();
  Integer getSessions();
  Long getUptime();
}
```

A JMX management interface contains attributes and operations. In a standard MBean those attributes and operations are expressed as methods in a Java interface. Methods that have the following form:

```
AttributeType getAttributeName();
void setAttributeName();
```

define an attribute named `AttributeName` that takes values of type `AttributeType`. If the `setAttributeName()` method is missing, the attribute is read-only; if the `getAttributeName()` method is missing, the attribute is write-only. Any method in the MBean interface that doesn't define an attribute defines an operation for the management interface.

The first three methods in the `ServerMBean` interface define operations that a management application can invoke on `ServerMBean` instances. The remaining methods define attributes; specifically they define three read-only attributes: `Connections`, `Sessions`, and `Uptime`.

The implementation of the ServerMBean interface in the Server class is straightforward:

```
public class Server implements ServerMBean, NotificationListener {
  private SessionPool sessions;
  private SortedSet connectionQueue;
  private Listener listener;
  private int connections;  // incremented for each new connection
  private int tzero;   // System.currentTimeMillis at Server start
  …
  // Other Server methods that aren't part of the MBean interface

  /**
   * Shut down the server, killing the process and any active sessions
   */
  public void shutdown() {
    System.exit(0);
  }

  /**
   * Start a listener thread that will queue incoming connections
   */
  public void start() {
    listener = new Listener(connectionQueue);
    listener.start();
  }

  /**
   * Stop the server's listener thread; active sessions continue to
   * handle requests
   */
  public void stop() {
    listener.stopListening();
  }

  /**
   * Connections attribute getter
   * @returns total number of connections handled
   */
  public Integer getConnections() {
    return new Integer(connections);
  }

  /**
   * Sessions attribute getter
```

```
   * @returns number of active sessions
   */
  public Integer getSessions() {
    int as = sessions.getAvailableSessions().intValue();
    int sz = sessions.getSize().intValue();
    return new Integer(sz - as);
  }

  /**
   * Uptime attribute getter
   * @returns number of milliseconds since the server was started
   */
  public Long getUptime() {
    return new Long(System.currentTimeMillis() - tzero);
  }
}
```

The shape of the `SessionPool` management interface requires a little more thought. Clearly it needs a *grow* operation to satisfy the first session pool management requirement. What about the empty pool management requirement? We could specify a `monitorPoolSpace` operation that would provide the necessary behavior, but as we'll see in a moment, that would be reinventing a perfectly good JMX wheel. Instead, let's just satisfy the underlying data requirement by providing access to the number of sessions left in the pool. A glance back at the `ServerMBean` implementation will reveal that we've already assumed that this information, along with the session pool size, is available, so we have this:

```
public interface SessionPoolMBean {
  void grow(int increment);
  Integer getAvailableSessions();
  Integer getSize();
}
```

todd uses `java.util.Set` as the underlying data structure for the `SessionPool` implementation:

```
public class SessionPool implements SessionPoolMBean {
  private static final int DEFAULT_POOLSIZE = 8;
  private Set sessions;
  private int size;
```

```
/**
 * Default constructor creates a SessionPool instance of size
 * DEFAULT_POOLSIZE
 */
public SessionPool() {
  this(DEFAULT_POOLSIZE);
}

/**
  * Creates a SessionPool instance of the specified size and
  * fills it with Session instances
  */
public SessionPool(int size) {
  this.size = size;
  sessions = new HashSet(size);
  fill();
}

/**
 * Increase the number of Session instances in SessionPool by
 * increment
 * @param increment the number of Session instances to add to
 * the pool
 */
public synchronized void grow(int increment) {
  for (int i = 0; i < increment; i++) {
    Session s = new Session(this);
    sessions.add(s);
  }
  size = size + increment;
}

/**
 * AvailableSessions attribute getter
 * @returns number of sessions remaining in the pool
 */
public Integer getAvailableSessions() {
  return new Integer(sessions.size());
}

/**
 * Size attribute getter
 * @returns size of the session pool
 */
```

```
  public Integer getSize() {
    return new Integer(size);
  }
   …
   // Other SessionPool methods that are not part of the MBean
   // interface
}
```

You've probably noticed that all of our attribute getters return Java numeric wrapper types—`Integer`, `Long`, and so on. We do that so that we can use JMX monitors such as `GaugeMonitor`, which we'll use in the implementation of the empty pool management requirement, to observe the values taken by those attributes. Note also that we have kept the operations in the `ServerMBean` and `SessionPoolMBean` interfaces very simple.

2.6.4 Incorporating an MBeanServer

Now that we've got some MBeans, what do we do with them? Because a JMX-based management application can access MBeans only if they are registered with an MBeanServer, we should register them. Unfortunately, we don't have an MBeanServer in todd to register any MBeans with at the moment. That problem can be solved with a single line of code in todd's `main()` method:

```
MBeanServer mbs = MBeanServerFactory.createMBeanServer();
```

In the interest of minimizing the impact of incorporating the MBeanServer, we will instantiate both the `Server` and the `SessionPool` MBeans and then register them, rather than creating and automatically registering them via the MBeanServer. `Server` is instantiated in `main()`, and `SessionPool` is created as part of the `Server` instantiation:

```
public static void main(String[] args) throws Exception {
  MBeanServer mbs = MBeanServerFactory.createMBeanServer();

  Server server = new Server(mbs);
  ObjectName son = new ObjectName("todd:id=Server");
  mbs.registerMBean(server, son);

…
```

```
  while (server.isActive()) {
    Connection k = server.waitForConnection()
    server.activateSession(k);
  }
}

public Server(MBeanServer mbs) throws Exception {
  this.mbs = mbs;

  connectionQueue = new TreeSet();
  connections = 0;

  sessions = new SessionPool();
  ObjectName spon = new ObjectName("todd:id=SessionPool");
  mbs.registerMBean(sessions, spon);

  active = true;
  tzero = System.currentTimeMillis();
}
```

The MBeanServer associates an `ObjectName` instance with each MBean. We've registered `Server` and `SessionPool` under the names `todd:id=Server` and `todd:id=SessionPool`. The portion of the name to the left of the colon is the *domain*, which is an arbitrary string that is opaque to the MBeanServer but may have meaning to one or more management applications. On the right are the *key properties*, a set of name/value pairs that help distinguish one MBean from another. Together they must form a unique name, within a given MBeanServer, for the associated MBean.

2.6.5 Monitoring todd

We still haven't satisfied the `SessionPool` empty pool management requirement. The `SessionPool` MBean tells us how many sessions are left in the pool. What we need is a way to react to these two events: (1) `AvailableSessions` has become zero, and (2) `AvailableSessions` has increased from zero to one or more.

In JMX, events are called *notifications*. Every JMX notification has a class and a type; its class is either `javax.management.Notification` or one of its subclasses; its type is `String` expressed in dot notation—for example, `jmx.mbean.registered`. Notifications are handled by calls to one of the MBeanServer's `addNotificationListener()` methods:

```
public void addNotificationListener(ObjectName objname,
                                    NotificationListener listener,
                                    NotificationFilter filter,
                                    Object handback);

public void addNotificationListener(ObjectName objname,
                                    ObjectName listener,
                                    NotificationFilter filter,
                                    Object handback);
```

The only difference between the two method calls is the second parameter. In the first version the second parameter is a reference to a `NotificationListener` instance—that is, an instance of a class that implements the `NotificationListener` interface. In the second version the second parameter is the `ObjectName` instance of an MBean that implements `NotificationListener`.

Careful readers will have noticed that the `Server` class implements `ServerMBean` *and* `NotificationListener`. Satisfying the empty pool management requirement involves stopping and starting the server that makes the `Server` class. These actions provide the `stop()` and `start()` methods with a natural place to handle the notifications that trigger those actions. The `NotificationListener` interface declares a single method, `handleNotification()`. Here is the `Server` implementation of the method:

```
public void handleNotification(Notification n, Object hb) {
  String type = n.getType();
  if (type.compareTo
       (MonitorNotification.THRESHOLD_LOW_VALUE_EXCEEDED) == 0) {
    stop();
  } else if (type.compareTo
      (MonitorNotification.THRESHOLD_HIGH_VALUE_EXCEEDED) == 0) {
    if (isActive() == false) start();
  }
}
```

Now all that's required is a mechanism for generating notifications at the appropriate times. We need something that will monitor `SessionPool`'s `AvailableSessions` attribute, send a notification when it becomes zero, and then send another notification later when `AvailableSessions` increases to one or more.

The JMX GaugeMonitor class provides just such a mechanism. A `GaugeMonitor` instance is configured to monitor a specific attribute of an MBean registered with the MBeanServer. The `GaugeMonitor` class has two thresholds: high and low. A `MonitorNotification` instance with type `jmx.monitor.threshold.high` is sent when the attribute value increases to or past the high threshold value. Similarly, when the attribute value decreases to or below the low threshold value, a `MonitorNotification` instance with type `jmx.monitor.threshold.low` is sent.

The `Server.configureMonitor()` method sets up a `GaugeMonitor` instance that completes the implementation of the empty pool management requirement:

```
public static void configureMonitor(MBeanServer mbs) throws
  Exception {
  ObjectName spmon = new ObjectName("todd:id=SessionPoolMonitor");
  mbs.createMBean("javax.management.monitor.GaugeMonitor",
spmon);

  AttributeList spmal = new AttributeList();
  spmal.add(new Attribute("ObservedObject", new
    ObjectName("todd:id=SessionPool")));
  spmal.add(new Attribute("ObservedAttribute",
    "AvailableSessions"));
  spmal.add(new Attribute("GranularityPeriod", new Long(10000)));
  spmal.add(new Attribute("NotifyHigh", new Boolean(true)));
  spmal.add(new Attribute("NotifyLow", new Boolean(true)));
  mbs.setAttributes(spmon, spmal);

  mbs.invoke(
    spmon,
    "setThresholds",
    new Object[] { new Integer(1), new Integer(0)},
    new String[] { "java.lang.Number", "java.lang.Number" });

  mbs.addNotificationListener(
    spmon,
    new ObjectName("todd:id=Server"),
    null,
    new Object());

  mbs.invoke(spmon, "start", new Object[] {}, new String[] {});
}
```

The first two lines here create and register a `GaugeMonitor` MBean named `todd:id=SessionPoolMonitor`. The next seven lines set attributes that tell `GaugeMonitor` which attribute of which MBean should be monitored (`ObservedAttribute` or `ObservedObject`), how often (`GranularityPeriod`, in milliseconds), and whether or not to send a notification on high-threshold and low-threshold violations. Then we invoke the `setThresholds()` method, via the MBeanServer, to set the actual high and low threshold values. Finally, we make the server listen for session pool monitor notifications and start the gauge monitor.

2.6.6 Browser Control

The new version of todd satisfies all of the management requirements specified in Table 2.1. There is, however, one more issue to address: We would like to be able to control todd interactively. For example, an administrator should be able to connect to todd from home to check the values of key MBean attributes, stop the server, increase the size of the session pool, and restart the server if necessary.

Although there are as many ways to address this issue as there are communication protocols, most JMX implementations provide an HTTP adapter that allows a user to "surf" an MBeanServer—inspecting MBean attributes, invoking operations, and so on—using a standard Web browser. Here's the code necessary to start the TMX4J HTTP adapter:

```
ObjectName httpon = new ObjectName("adapters:id=Http");
mbs.createMBean("com.tivoli.jmx.http_pa.Listener", httpon);
mbs.invoke(httpon, "startListener",
new Object[] {}, new String[] {});
```

Once the adapter has been started, you can connect to the MBeanServer via a Web browser. In the case of the TMX4J HTTP adapter, you would direct your browser to http://<your-server>:6969. You can change the port the adapter listens on in the `jmx.properties` file that the TMX4J implementation reads on startup. Figure 2.7 shows the TMX4J view of the todd MBeans.

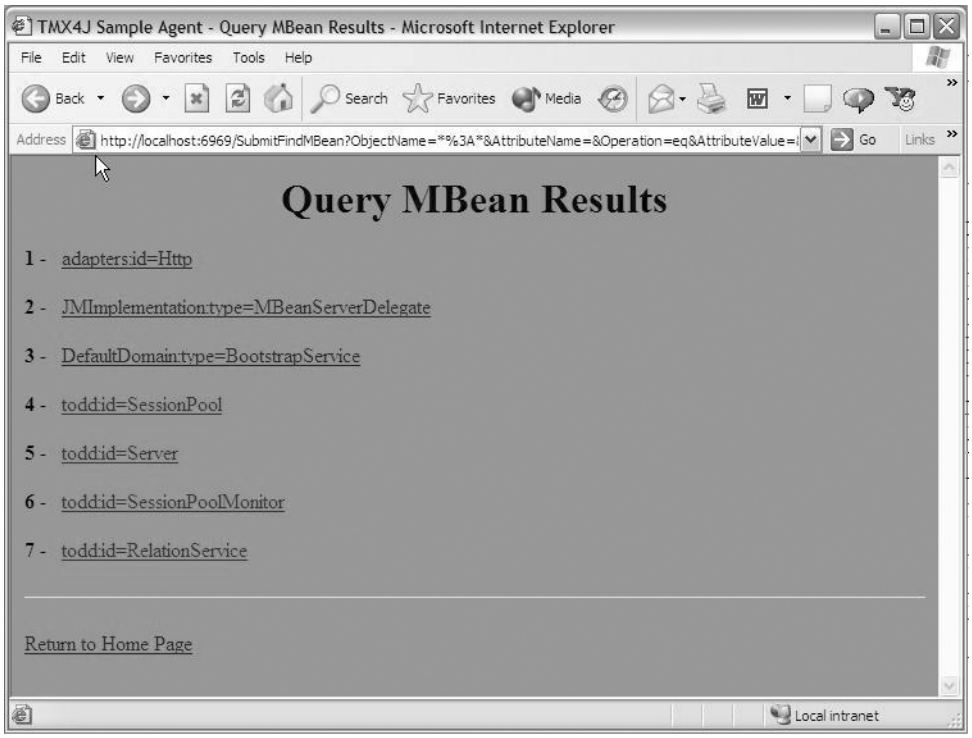

Figure 2.7 The TMX4J HTTP Adapter's MBean View

2.7 Summary

In the course of this tour we have illustrated the use of JMX in the context of a simple Java-based service. Although the service itself may be straightforward, the process we went through to make it manageable is a common approach and does work in practice. Working with a set of management requirements, we developed a set of resources that we must monitor and control to satisfy those requirements. We reflected the management interface for each of our managed resources in an appropriate MBean, and then brought instances of those MBeans together in an MBeanServer. Finally, we used JMX services like monitors and notification to implement the policy mechanism that enabled us to satisfy our management requirements.

2.8 Notes

1. This chapter is generally attributable to the following article: H. Kreger, "Java Management Extensions for Application Management," *IBM Systems Journal* 40(1) (March 2001), http://www.research.ibm.com/journal/sj/401/kreger.pdf.

2. "Java Management Extensions (JMX) Specification," JSR 3, http://www.jcp.org/jsr/detail/3.jsp.

3. J2SE stands for Java 2 Platform, Standard Edition, which is Sun Microsystems' Java platform. More information is available at http://java.sun.com/j2se. Java and all Java-based marks are trademarks of Sun Microsystems, Inc., in the United States and other countries.

4. SNMP stands for Simple Network Management Protocol, which is an IETF (Internet Engineering Task Force) standard. More information on SNMP is available at http://www.ietf.org.

5. CIM/WBEM stands for Common Information Model/Web-Based Enterprise Management. It is defined by the Distributed Management Task Force (DMTF). More information is available at http://www.dmtf.org.

6. JDBC (Java Database Connectivity) is an API that isolates database clients from database vendors. It is a Sun Microsystems technology. JDBC is a trademark of Sun Microsystems, Inc., in the United States and other countries.

7. CMIP stands for Common Management Information Protocol and is usually referred to in conjunction with CMIS (Common Management Information Services). This management standard was defined by OSI (Open Systems Interconnection) as an ISO standard: ISO 9595/2 and 9596/2 (http://www.iso.ch). More information on CMIP/CMIS can be found at http://www.iso.ch.

8. Tivoli Systems, Inc., 9442 Capital of Texas Highway North, Arboretum Plaza One, Austin, TX 78759 (http://www.tivoli.com). Tivoli is a trademark of Tivoli Systems in the United States, other countries, or both.

9. Computer Associates International, Inc., One Computer Associates Plaza, Islandia, NY 11749 (http://www.cai.com).

10. Microsoft Windows is Microsoft's workstation operating system family, including Windows 95, Windows 98, Windows NT, Windows 2000, and Windows XP. More information is available at http://www.microsoft.com. Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

11. IBM AIX is a UNIX-based operating system available from IBM Corporation, Armonk, NY (http://www.ibm.com/servers/aix).

12. Solaris is Sun Microsystems' UNIX-based operating system. More information is available at http://www.sun.com/solaris.

13. HP-UX is Hewlett-Packard's UNIX-based operating system. More information is available at http://www.hp.com/products1/unix/operating.

14. J2EE stands for Java 2 Platform, Enterprise Edition, which is Sun Microsystems' Java platform. J2EE application servers are vendor products that support the J2EE specification. More information is available at http://java.sun.com/j2ee. Java and all Java-based marks are trademarks of Sun Microsystems, Inc., in the United States and other countries.

15. RFC 1514 (http://www.ietf.org/rfc/rfc1514.txt) dictates that, "The Internet-standard Network Management Framework consists of three [four] components. They are: STD 16, RFC 1155 [1] which defines the SMI, the mechanisms used for describing and naming objects for the purpose of management. http://www.ietf.org/rfc/rfc1155.txt; STD 16, RFC 1212 [2] defines a more concise description mechanism, which is wholly consistent with the SMI. http://www.ietf.org/rfc/rfc1212.txt; STD 17, RFC 1213 [3] which defines MIB-II, the core set of managed objects for the Internet suite of protocols.; STD 15, RFC 1157 [4] which defines the SNMP, the protocol used for network access to managed objects."

16. JMAPI .08, Java Management API specification. These specifications are no longer available from Sun Microsystems; however, this book contains an overview of the JMAPI technologies: "Java 1.2 Unleashed," Sams Publishing (http://www.szptt.net.cn/9810dnwl/new/jdk1.2/index.htm). This chapter covers JMAPI (see http://www.szptt.net.cn/9810dnwl/new/jdk1.2/ch36/ch36.htm).

17. JDMK 2.0 is available from Sun Microsystems, Inc., 901 San Antonio Rd., Palo Alto, CA 94303. More information is available at http://java.sun.com/products/jdmk.

18. Source: "Java Management Extensions (JMX) Specification," JSR 3, http://www.jcp.org/jsr/detail/3.jsp, which was led by Sun Microsystems to create a management API for Java resources.

19. Java Community Process (http://www.jcp.org) is Sun Microsystem's process for allowing the Java community to participate in the development of Java language extensions and new Java APIs.

20. Groupe Bull was the original company name for the representative. This company is now referred to as "Evidian: A Groupe Bull Company" (http://www.evidian.com).

21. TIBCO Software is a provider of business integration solutions (http://www.tibco.com).

22. Powerware, an Invensys Company, Powerware Corporation, designs and manufactures innovative, end-to-end power protection and management solutions (http://www.powerware.com).

23. Borland Software Corporation is a provider of technology used to develop, deploy, and integrate software applications (http://www.borland.com).

24. Motorola, Inc., is a provider of integrated communications solutions and embedded electronic solutions (http://www.motorola.com).

25. BEA Systems, Inc., is an application infrastructure software company (http://www.bea.com).

26. IONA iPortal J2EE is an application server by IONA. According to IONA, the iPortal Application Server has been incorporated into the Orbix E2A Application Server Platform as the Orbix E2A J2EE Technology Edition. More information is available at www.iona.com.

27. Produced by Lutris Technologies, Lutris EAS 4 is a J2EE application server that introduces a services architecture, in which J2EE services are pluggable modules, that incorporates JMX manageability into every service, as well as full versioning of service components for complete configuration and product packaging control. More information is available at http://www.lutris.com.

28. JBoss is an open-source J2EE application server. It is inherently JMX based. More information is available at http://www.jboss.org.

29. Java Dynamic Management Kit (JDMK) 3.0 is available from Sun Microsystems, Inc., 901 San Antonio Rd., Palo Alto, CA 94303, http://java.sun.com/products/jdmk.

30. Java Dynamic Management Kit (JDMK) 4.0 is available from Sun Microsystems, Inc., 901 San Antonio Rd., Palo Alto, CA 94303 (http://java.sun.com/products/jdmk).

31. "Java Management Extensions (JMX) Remoting 1.2," JSR 160, http://www.jcp.org/jsr/detail/160.jsp, led by Sun Microsystems.

32. JSR 146: "WBEM Services: JMX Provider Protocol Adapter" (http://www.jcp.org/jsr/detail/146.jsp), led by Sun Microsystems.

33. JSR 71: "JMX-TMN Specification," (http://www.jcp.org/jsr/detail/071.jsp), led by Evidian.

34. JSR 70: "IIOP Protocol Adapter for JMX Specification," (http://www.jcp.org/jsr/detail/070.jsp), led by IONA.

35. IIOP stands for Internet Inter-Operability Protocol. More information is available at http://www.omg.org and http://www.omg.org/technology/documents/corba_spec_catalog.htm.

36. CORBA stands for Common Object Request Broker Architecture. More information is available at http://www.omg.org and http://www.omg.org/gettingstarted/corbafaq.htm.

37. The specification in JSR 77, "J2EE Management" (http://www.jcp.org/jsr/detail/077.jsp), led by Sun, defines the object model and JMX implementation for managing J2EE application servers.

38. JSR 146: "WBEM Services: JMX Provider Protocol Adapter" (http://www.jcp.org/jsr/detail/146.jsp), led by Sun Microsystems.

39. JSR 71: "JMX-TMN Specification," (http://www.jcp.org/jsr/detail/071.jsp), led by Evidian.

40. *Java Management Extensions Instrumentation and Agent Specification v1.0* (Final Release, April 2000), Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303; available at http://java.sun.com/products/JavaManagement.

41. According to the Free Software Foundation (http://www.gnu.org/philosophy/license-list.html#SunCommunitySourceLicense) and numerous editorials at the time of the SCSL release, this license is not well thought of in the free-software and open-source communities.

42. *Java Management Extensions Technology Compatibility Kit 1.0* (April 2000), Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303; available at http://java.sun.com/products/JavaManagement.

43. Tivoli's Application Management Specification is used to define the characteristics of a managed application.

44. The MIB file is the Management Information Base data format used by SNMP to describe its object model. Source: M. Rose and K. McCloghrie,

"Concise MIB Definitions," STD 16, RFC 1212 (March 1991), http://www.ietf.org/rfc/rfc1212.txt?number=1212.

45. MOF stands for Managed Object Format. This format is used to describe CIM information and is defined by the DMTF in the CIM specification. More information is available at http://www.dmtf.org/standards/cim_schema_v23.php.

46. The Dynamic Invocation Interface for CORBA description can be found at http://www.omg.org. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries.

47. RMI stands for Remote Method Invocation, a Java API to support distributed programming with Java technology. More information is available at http://java.sun.com/products/jdk/rmi.

48. HTTP stands for Hypertext Transfer Protocol. See "Hypertext Transfer Protocol -- HTTP/1.1," RFC 2068 (January 1997), http://www.ietf.org/rfc/rfc2068.txt?number=2068.

49. SNMP stands for Simple Network Management Protocol, the protocol used for network access to managed objects. The SNMP is defined in RFC 1157 ("A Simple Network Management Protocol (SNMP)," May 1990, http://www.ietf.org/rfc/rfc1157.txt).