Experiences from Implementation and Evaluation of Event Processing and Distribution of Notifications in an Object Monitoring Service

ALEKSANDER LAURENTOWSKI AND KRZYSZTOF ZIELIŃSKI

Institute of Computer Science, AGH University of Science and Technology, Al. Mickiewicza 30, Kraków, Poland
Email: al@ics.agh.edu.pl, kz@ics.agh.edu.pl

Organization of events processing and distribution of notifications is a key issue for performance of software monitoring and management systems, particularly for reducing the execution-time overheads they incur. This paper provides practical guidelines for design and implementation of event report generation and dissemination frameworks, based on experiences from implementation, configuration and evaluation of Melita On-line Object Monitoring Service (MOOMS): an on-line monitoring service for distributed object-oriented applications. As MOOMS is in many aspects similar to the Java Management Extensions (JMX) framework, the presented evaluation may shed some light on how this class of object services may be developed and used with efficiency in mind.

Received 6 August 2002; revised 15 July 2003

1. INTRODUCTION

Processing and distribution of events play a fundamental role in monitoring and management systems. The architectures of this class of systems have often been a subject of research and refinement, with a notably mature and universal approach recently proposed by Sun Microsystems as the Java Management Extensions (JMX) [1] specification and Java API. Nevertheless, a high level of abstraction and generality of this proposal have left much space for configuration and implementation details that may have a tremendous influence on the performance of applications. In particular, we believe that the design and implementation of the two lowest architectural levels from the monitoring systems' reference model [2, 3]—the monitoring information generation and dissemination layers—are the key for the ultimate performance of the target system. Middleware service programmers and users also demand practical guidelines on how to implement and use the monitoring service frameworks in an efficient way. This paper aims at providing such guidelines, based on experiences from implementation, configuration and evaluation of Melita On-line Object Monitoring Service (MOOMS) [4, 5], which has been conceived independently, but at the same time when the JMX specification efforts took place.

MOOMS is dedicated to monitoring the lifecycle of software objects, having in mind that each component of a real IT system may itself be such an object or may be mapped into a representing or wrapping object. MOOMS is language- and middleware-independent, and has already been implemented for two different middleware systems. Coincidentally, similar initial assumptions of MOOMS and JMX resulted in a situation that MOOMS—due to their similar architecture and functionality—may be easily considered as a system developed under the JMX framework. But, in contrast to JMX, MOOMS's design and implementation were strongly performance-driven in the sense of reducing execution-time overheads. Thus, its performance evaluation may shed some light on how JMX implementations and their applications may be efficiently designed and configured, at least in the sense of performance implications associated with the use of this class of services—despite the fact that JMX is basically limited to Java and provides (as a general-purpose framework) a richer functionality than MOOMS.

The paper is structured as follows. Section 2 is an attempt to gather the requirements for events reports' generation and dissemination layers in monitoring services. Section 3 describes the architecture and functionality of the MOOMS service, as a case study of the targeted class of systems. In Section 4, the architectures of JMX and MOOMS have been compared with stress on the organization of monitoring events' generation and distribution mechanisms. An overview of possible implementational solutions that could be easily applied to this category of systems, illustrated by example of the MOOMS service, is given in Section 5. Section 6 brings in a performance study of influence of these solutions, and of Quality of Service (QoS) mechanisms applied to tune event reports' dissemination, on the incurred execution-time overhead. The paper ends with conclusions and service usage recommendations based on the obtained results.

2. REQUIREMENTS FOR EVENTS' GENERATION AND DISSEMINATION

This section identifies the basic issues concerning requirements for events' generation and dissemination layers of on-line remote software monitoring services for distributed OO applications. It focuses on the following key questions:

(i) What should be the target of monitoring and what events should be intercepted?
(ii) How is the monitoring information gathered and disseminated?
(iii) What are the configuration and management requirements for the service?

2.1. Monitored entities and events

The actual target monitored entities to which the service should be applied are usually application-level software objects of distributed applications, as they model or represent the real-world resources under scrutiny. These objects, called monitored objects, can be of different granularity: both language-level (e.g. C++ or Java objects) and larger-grain networked objects (e.g. CORBA objects) and components (e.g. JavaBeans), but they may also wrap or represent even larger application components, legacy code and hardware devices/interfaces.

The state of monitored objects, described by the objects' attributes and their values, should be under monitoring. In the monitoring service model, the run-time activity of monitored objects (and, in effect, of the server in which they reside) can be thus depicted by a history of changes of their attributes. Therefore, tracking changes of monitored attributes' values and subsequently sending notifications (event reports) on that should be the key functionality of the monitoring service.[1]

Logical relations (associations) between monitored objects should also be under monitoring. Real-world objects, structures and organisms always enter in logical relations with other objects/organisms. Object-oriented systems have been invented to model the real-world beings, so their nature is alike. Inter-object relations may be of different type and lifecycle profile (e.g. static or dynamic) and a mature monitoring service must provide facilities to express, define and discover graphs of relations and their shape (topology), as well as to track changes to these graphs if applicable [6, 7].

2.2. Generation and dissemination of monitoring information

The key elements of each monitoring system's infrastructure are sensors (probes) residing in the monitored application and observing the behaviour of a small part of its state space in order to intercept events and report on them, i.e. generate event reports, also called notifications.

The possible dissemination schemes (also called monitoring modes) include sampling (pull-mode), tracing (push-mode) and periodic notification on events [8, 9, 10].

Sampling is querying sensors on demand about the current state of monitored entities. Tracing is immediate notification subsequent to the event, initiated by a sensor residing upon the monitored resource.

Sampling by definition is rather infrequent, as it is steered manually by an operator and thus adapted to the—limited—human perception. Tracing implies a potentially more intensive operation of the monitoring service infrastructure, as the initiative is here on the system's side and one can expect much higher event rates. This in turn may cause degradation of the underlying monitored system's performance by an increase in the execution-time of the application, called overhead.

Therefore, a proper design and implementation of the monitoring service infrastructure responsible for generation and dissemination of notifications in the push-mode (tracing) is particularly important in performance-sensitive applications.

2.3. Configuration and management requirements

For a better usability and user acceptance, a monitoring service should be flexible, dynamically configurable and manageable at run-time, as the users' requirements may change in time.

Multiple concurrent users, in the configuration and management context, mean that the service must allow dynamic creation and update of a set of monitoring applications simultaneously watching a particular server/application under observation, and not only a single monitoring client.

The second issue is a dynamic definition of a scope of monitoring, i.e. the view that a particular observer can build for his/her own purposes by choosing only the targets of interest from the monitored system. Free choice should concern the types of events and notifications emitted by monitored entities, as well as the delivery scheme (monitoring mode). Monitoring applications should be able to operate at different levels of abstraction of monitored entities, ranging from an attribute level, through object level to a class of objects or even applications. In other words, a monitoring service must provide a fully dynamically configurable, distributed architecture, allowing an arbitrary number of observers to connect/disconnect to/from monitored servers, choose a different set of monitored entities, types of events and activate different functionalities (e.g. choose a proper monitoring mode) per each monitoring application at run-time, in order to keep up with various application scenarios and varying views and functional expectations of different users.

A monitoring service should also provide some kind of mechanisms, configuration facilities and parameters for management of performance factors like the incurred

[1] An alternative or supplement to this approach is tracking invocations of object methods/operations, but note that not all these inter-object communication events must result in changes of state of objects (e.g. attribute read operations).

overhead, notification delay or event rate at the client side. For example, the service could provide relevant QoS mechanisms and parameters with regard to dissemination of notifications.

3. CASE STUDY: MOOMS ARCHITECTURE

The goal of this section is to describe the architectural design of MOOMS—a distributed on-line monitoring service for software objects that satisfies the architectural requirements defined in Section 2 and was the prototype for our performance evaluations.

The primary intention of MOOMS's design and implementation was that the service should be lightweight and efficient, i.e. it should minimize the (anyway inevitable) execution-time overheads introduced by the monitoring software infrastructure to the business computations of the underlying (monitored) application. Moreover, MOOMS is language- and middleware-independent, i.e. its modules, IDL interfaces and classes may be implemented in most programming languages (e.g. C++ or Java) and over any middleware platforms (e.g. CORBA, RMI or proprietary systems).

3.1. MOOMS infrastructure inside a monitored component

Figure 1 depicts the architecture of MOOMS and identifies the key elements of its monitoring infrastructure. In this viewpoint a group of objects residing in a target component (e.g. a CORBA server) are selected and monitoring-enabled at a software instrumentation phase and in effect are visible to the service and its users as monitored objects (MOs). MOOMS enables its user to bind MOs into a tree, in order to express containment relations between these objects (also relations determined solely for the needs of monitoring and management). This tree, called Monitoring Information Base (MIB), creates the heart of the monitoring infrastructure.

The functionality of the service is monitoring the lifecycle and containment relations of monitored objects (i.e. addition, removal and physical deletion of the nodes of the MIB tree) and their run-time activity expressed by changes of values of their attributes (also selected for monitoring at the instrumentation phase). The activity of intercepting these events and formulating notifications (event reports) is performed by two kinds of sensors: MO-sensors (which are actual nodes of MIB, aggregated with monitored objects), and attribute sensors (which are wrappers replacing original monitored object's attributes). These sensors, along with external service interfaces and their implementation, form the key part of the monitoring infrastructure, generating monitoring information and disseminating it among the monitoring client programs being direct users of the service. In the current MOOMS implementation, the invocations of appropriate sensors' operations are to be hand-coded into the application source code, while the implementation of sensors (a class hierarchy) is provided as a linkable library.

3.2. MOOMS interfaces and monitoring modes

Communication between monitoring clients and the server's monitoring infrastructure is maintained by invocations of two types of interfaces (see Figure 1):

(i) Server-side interfaces: Introspection, Pull Monitoring, Push Monitoring, Subscription End-Point (SEP) (containing also a QoS-setting operation) and Management.
(ii) Client-side interface called Observer that must be implemented by a monitoring client working in the push-mode to enable transmission of notifications from the target server. This idea is based on the well-known Observer design pattern [11]. The stream of event notifications forms a communication channel which can be tuned with QoS mechanisms.

The server-side interfaces support the following functionality of MOOMS:

(i) Run-time instance-level introspection of monitored entities (monitored objects with their attributes in the context of containment relations described by the MIB)—via the Introspection interface;
(ii) pull-mode monitoring—via the Pull Monitoring interface;
(iii) monitoring client registration (attachment) to the push-mode monitoring (tracing)—via the Push Monitoring interface;
(iv) subscription to the chosen monitored entities in the push-mode—via the SEP interface;
(v) setting QoS policies and parameters for streams of event notifications to monitoring clients—via the QoS-setting operation of the Subscription End-Point interface; and
(vi) management of monitored entities—via the Management interface.

The Introspection interface enables monitoring clients to browse the MIB at run-time in order to discover the monitored objects and attributes of interest and obtain their names, identifiers, types, properties and contained objects, i.e. descendants ('children') in MIB.

Once a monitoring client program knows a name or identifier of a monitored attribute discovered using Introspection, it may get (sample) its current value with usage of the Pull Monitoring interface.

Alternatively, the monitoring client may initiate the push-mode monitoring by connecting to the server and subscribing for one or more types of events concerning a discovered entity with usage of the Push Monitoring and SEP interfaces. Basically, the types of events can be: addition or removal of subordinate objects in MIB or a physical deletion while the entity is an MO, or change of value in case the entity is an attribute. Should such events happen, the monitoring infrastructure in the target server will intercept them and generate and transmit relevant notifications to the monitoring client(s) by invocations of the Observer interface. Here

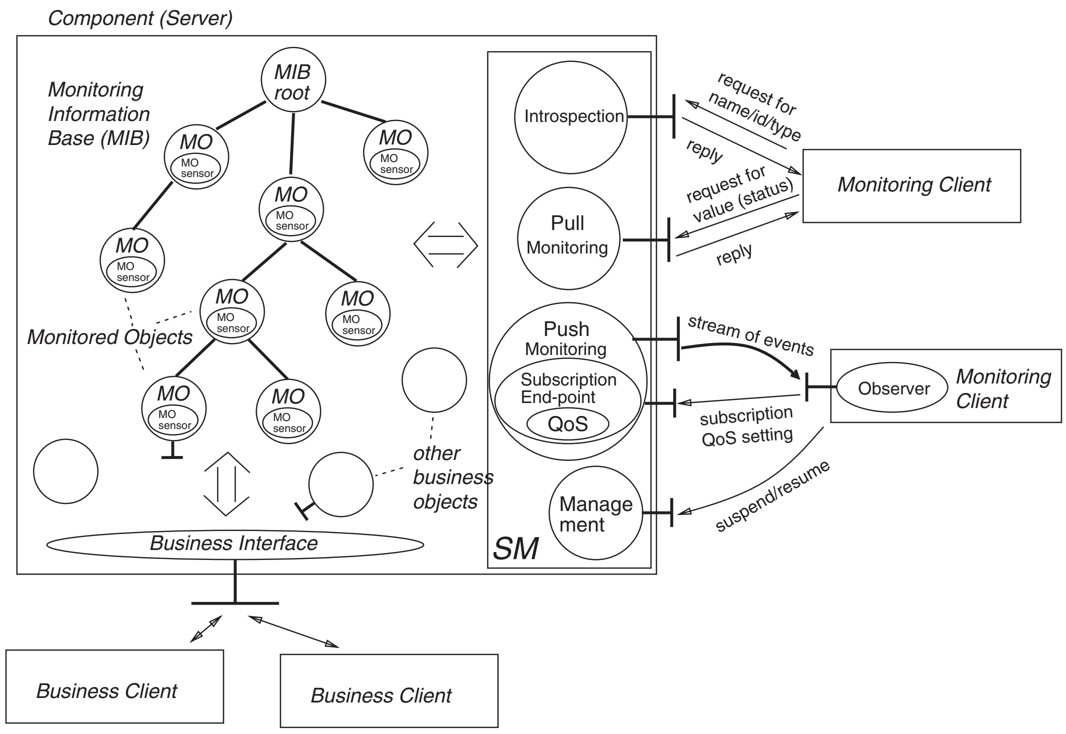


FIGURE 1. Architecture and interfaces of the MOOMS monitoring service.

the active role is played by the monitored server, and thus push monitoring is an implementation of tracing. When a monitoring client is no longer interested in receiving notifications from a particular event source (MO or attribute sensor), it may un-subscribe from it or even disconnect from the server at any time.

3.3. Event notification with QoS elements

One object implementing the SEP interface will be created in the monitored server for each observer (monitoring client), thus forming a subscription and QoS controlling end-point for the one-to-one communication channel to that client. To manage a potentially high-throughput of notifications incoming to its Observer interface over that channel, the monitoring client may invoke a relevant operation from the SEP interface at the server-side to set appropriate QoS parameters for this event stream. Actually, the QoS policies available in MOOMS are targeted mostly on the server side, to help to minimize the run-time overheads thereby reducing the CPU costs of remote invocations of notification operations. These policies are as follows.

Batching event reports, i.e. collecting them into a sequence and delaying their dispatching until the size of that sequence meets some threshold value, given by the monitoring client as a QoS parameter. The advantage of batching is that delivery of notifications—a potentially heavyweight remote invocation—takes place only every *n* events (where *n* is the sequence size), so the communication overhead is shared between event reports. In case of low event loads, batching should be supplemented by an automatic flush mechanism after an appropriate timeout set for the event reports collection period. This would prevent unpredictable delays of event reports delivery.

Periodic notification, with the period to be set by the monitoring client as a QoS parameter. That means sending a sequence of notifications on all events that happened during the last period. Both batching and periodic notification are per-observer policies, and a server servicing many monitoring clients in parallel may employ a different policy for each one.

Priorities of delivery for events originating from chosen sources. Some sensors may produce alarm events that are of special importance to some monitoring clients. Alarm event notifications should be delivered to those clients immediately, i.e. in practise as soon as possible or at least with a higher priority than regular events (i.e. the rest of events), depending on a possible implementation. MOOMS provides a high-priority mode of notifications' delivery, called urgent mode, along with a standard, normal-priority one, called regular mode. This QoS policy is set on a per-source basis (i.e. per-attribute-sensor or per-MO-sensor), and therefore it must be set in the event generation layer, i.e. in a sensor, which must dispatch event reports for a delivery with a required priority. The other policies could be implemented in the dissemination layer only.

The Management interface, in turn, allows for run-time management of event sources: temporarily suspending (and later resuming) generation of notifications by chosen sensors working in the push monitoring model. This approach of just avoiding excessive generation of data is effective and much simpler and cheaper than data processing in filters, which can be quite expensive. A wise usage of Management can bring substantial decrease is execution-time overheads generated by the monitoring infrastructure, particularly in scenarios with a number of sensors active concurrently.

The MOOMS service maintains a fine-grained (i.e. per-sensor) subscription for notifications. This design assumption allows in the implementation to keep a registry of observers subscribed to a particular sensor. It helps to assure that no notifications are generated when there are no subscribers even when a sensor is active ('switched on'), and thus to reduce the overhead to an indispensable minimum in such cases.

3.4. Other work

The assumptions on types of events and modes of monitoring place MOOMS in the stream of management and monitoring systems employing polling a dedicated interface and/or triggering generation of event reports on events' interception. This approach has been adopted e.g. by SNMP and OSI/ISO [12] management standards, and recently JMX or the DMTF model (so called Triggers [13] and, to some extent, by the solutions described by Debusman and Kroeger in [14].

A different approach is tracing invocations or transactions in distributed applications, to build e.g. an execution log for visualization, debugging or other purposes. This model is represented e.g. by CORBA Portable Interceptors, experimental systems like MODIMOS [7] or in some degree, by the Application Response Measurement (ARM) API [15].

It should be also underlined that the MOOMS architecture is open to plugging-in compatible standard mechanisms and services, e.g. the CORBA Event or Notification service to be used for events delivery instead of the simplistic (but in many cases more efficient) invocation of the Observer interface.

4. JMX VERSUS MOOMS: A COMPARISON OF MONITORING FRAMEWORKS

This section brings a short comparison of two existing monitoring frameworks: JMX and MOOMS, satisfying most of the requirements proposed in Section 2. These frameworks have been designed in parallel, but have some common fundamental features that will be revealed and elaborated below. A framework, by definition, is rich and flexible, so that it can be used like a tool box: you have all the tools (here: different functionalities, modes of operation etc.) available down there, but you pick up only what you need at a moment. The spectrum of 'must-provide' functionalities has been established throughout the recent years by network management standards (SNMP, CMIS/CMIP [12]), CORBA Event and Notification object services [16, 10] and many commercial and research monitoring tools [6, 17, 18].

4.1. A comparison of architectures and interfaces

The JMX specification defines a generic notification model based on the Java event model. The JMX architecture is divided into three levels: instrumentation level, agent level and distributed services level. The current version of the JMX specification [1] addresses the first two levels and provides only a brief overview of the latter. JMX provides a rich framework functionality out of which users may choose features, interfaces and services adequate to their current needs. The MOOMS project addresses all three respective levels, but in a much simpler approach, while putting more attention to dissemination of events with QoS elements.

Both investigated frameworks deliver the three basic monitoring modes: sampling (pull-mode), tracing (push-mode) and periodic notification on events, out of which the user is able to choose and use one or more.

The key architectural components of the instrumentation level in both systems are instrumented objects called either Managed Beans (MBeans) in JMX or MO in MOOMS. They implement a relevant monitoring/management instrumentation interface and interoperate with agent-level components such as MBean Server in JMX or service infrastructure level interfaces like SEP, Introspection and Pull Monitoring in MOOMS. An MBean is a Java object that encapsulates attributes and operations through their public methods and follows specific design patterns for exposing them to management applications via agent-level access and manipulation operations. A MO in MOOMS possesses its own relation-sensitive MO-sensor and may contain a collection of attribute sensors, which are objects substituting chosen original attributes of that MO. The former type of sensor notifies on changes of a graph of containment relations formed by MOs, while the latter one intercepts changes of attributes' values and generates event reports on such events.

The primary mechanism to achieve the desired flexibility and dynamic configurability of both investigated monitoring services is introspection—a run-time mechanism for discovering what is available there for monitoring in the underlying monitored application/server: objects, their attributes, interfaces, properties, associations, types of events etc. In JMX introspection can be implemented with so called standard MBeans by applying naming convention design patterns to attributes, their getter/setter methods and operations. A JMX agent uses introspection to look at the methods and superclasses of a class to determine if it represents an MBean that follows the design patterns, and to recognize the names of both attributes and operations. A more intuitive method of using a special DynamicMBean inspection interface can be used for the more advanced types of MBeans, i.e. dynamic and open MBeans. MOOMS follows an approach similar to this latter one, by offering a single, uniform Introspection interface for discovering MOs, their containment relations with other MOs, attributes, types, names, identifiers and properties.

The core component of the JMX infrastructure is the MBean server, which is a registry for MBeans. Any object registered with the MBean server becomes visible to management applications and all management/monitoring operations applied to MBeans need to go through this agent-level entity. The MBean server provides methods necessary for the creation, registration and deletion of MBeans, as well as the access methods for registered MBeans and their attributes (for getting and setting their values). It also contains a method for discovering attributes, operations, notifications (event reports) and constructors exposed by an MBean for management/monitoring. In MOOMS, this functionality is implemented by the MIB (comprised of sensors) and is distributed across the different service interfaces, including SEP, Pull Monitoring and Introspection.

Incidentally, the same sensor switch on/off functionality provided in MOOMS via the Management interface is available in JMX on the sensor level via so called monitor MBeans (see below).

4.2. A comparison of notification models

In the JMX's notification model, event reports can be emitted by MBean instances and by the MBean server, generally on specific changes of the MBean's attributes which are the fields or properties of the MBean's management interface. The mechanism for detecting changes in attributes and triggering the notification of events is not a part of the JMX specification, at least in its current release. The attribute change notification behaviour is therefore generally dependent upon the implementation of each MBean's class. An exception to this are monitor MBeans, which are in fact predefined sensors performing periodic sampling of an attribute in the MBean they observe. The three types of monitor MBeans that must be provided in every JMX implementation are CounterMonitor, GaugeMonitor and StringMonitor. If switched on, each of them automatically sends a relevant notification when a specific set of conditions concerning the value of the observed attribute is satisfied. In contrast to this, MOOMS implementations deliver ready-to-use classes of sensors for all simple data types, implementing exactly the same user interface as these predefined types respectively. In C++ this can be implemented in a particularly elegant way with usage of operators overleading.

JMX also provides a RelationService class supporting definition, management and querying of arbitrary relation types and relations between MBeans. This approach is more general than the simple containment relation supported by the MIB in MOOMS.

The monitoring client application in JMX may register once as a notification listener with a notification broadcaster MBean and further receive notifications on all events that may occur in the broadcaster, i.e. the listener's handleNotification() method will be invoked when any notification is issued by the MBean (an explicit implementation of the Observer design pattern [11]). All the monitor MBeans and the RelationService are broadcasters, but the NotificationBroadcaster interface may also be implemented by customary MBeans. Subscription for notifications may be performed also via the MBean server. A broadcaster MBean and its listeners exchange instances of subclasses of the Notification class. For example, the AttributeChangeNotification class allows management service and applications to be notified whenever the value of a given MBean's attribute is modified. The broadcaster has got a notification filter and an immutable hand-back object associated per listener. Filters help to constrain the potential stream of notifications, e.g. the AttributeChangeNotificationFilter class implements the following operations: enableAttribute(), disableAttribute(), disableAllAttributes() and getEnabledTypes()—the first three being self-describing while the last one returns a list of the attribute names currently enabled for emitting notifications.

The model specified by the current JMX specification only covers the transmission of events between MBeans within the same JMX agent (i.e. a local notification inside a single JVM). However, a JMX implementation may provide services that allow remote distribution of notifications via so called connectors, thus allowing a management application to listen to MBeans' events remotely. In the JMX architecture, an MBean has the full responsibility for sending notifications, but it may be supported by a relevant connector server by its side, as shown in Figure 2.

Similar to JMX, MOOMS also exploits the Observer design pattern in the approach to tracing. In MOOMS, the SEP objects form notification subscription and QoS management end-points for their monitoring client applications. One SEP object is created dynamically for each monitoring client in a one-to-one proxy relationship. Prior to the push-mode operation, each client registers its SEP at run-time as an observer (in the Observer design pattern sense) within each relation-sensitive MO-sensor or value-sensitive attribute sensor in the scope of interest of the given client. Therefore, similar to JMX, this process of subscription for notifications is performed directly with a target sensor. Quite importantly, however, in MOOMS the client settles the expected types of events and priority of delivery during subscription. This precise selection is intended to constrain the sensor as the source of events, i.e. to prevent it from generating unwanted monitoring information, as MOOMS does not provide a separate filtering functionality. A sensor only produces an event report and forwards a pointer to it to subscribed SEPs. These SEP objects are further responsible for dissemination of event reports to—potentially remote—monitoring client applications (as shown in Figure 3), as well as for implementing dissemination optimization and QoS policies over communication channels with observers. Such an approach enables a lightweight implementation of a sensor as a fine-grain object in comparison to an MBean. The advantages of this approach will be shown further in this paper. Pull-mode monitoring operations like getting the value of an attribute are invoked directly on the relevant sensor, but being human-initiated, they are definitely infrequent and do not affect performance.

Both JMX and MOOMS provide some mechanisms dedicated to improve their efficiency or reduce the execution-time overhead that they incur. An interesting feature of the MBean server is the cached values behaviour of attributes' values—if the requested data is up-to-date (i.e. a timeout since the last update has not yet expired), then the monitored/managed resource is not interrupted with a data retrieval request. MOOMS in turn employs the QoS policies of notifications' dissemination and several performance-supporting solutions in the service implementation, that will be elaborated in the following sections.

And last, but not least, it must be stated that JMX is a framework and API for the Java programming language only,

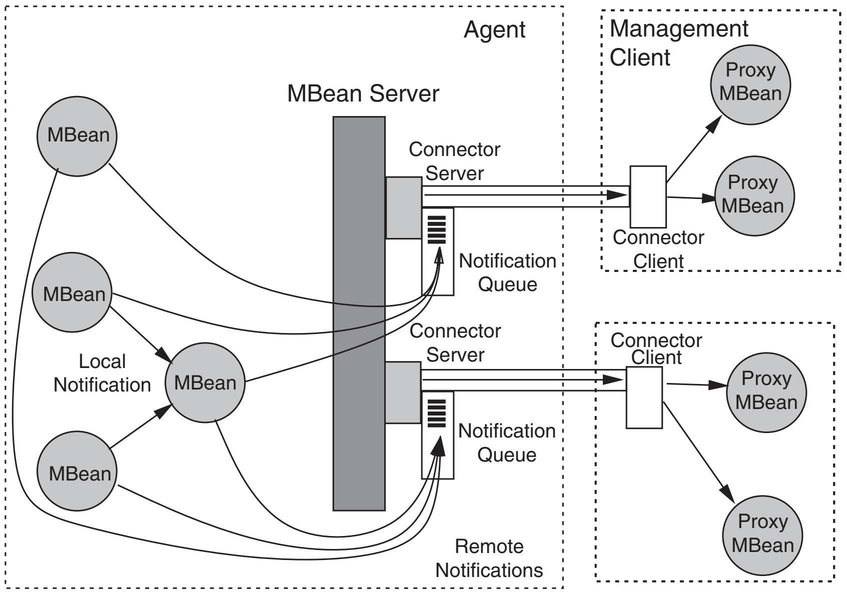


FIGURE 2. Architecture of generation and dissemination of event notifications in JMX.

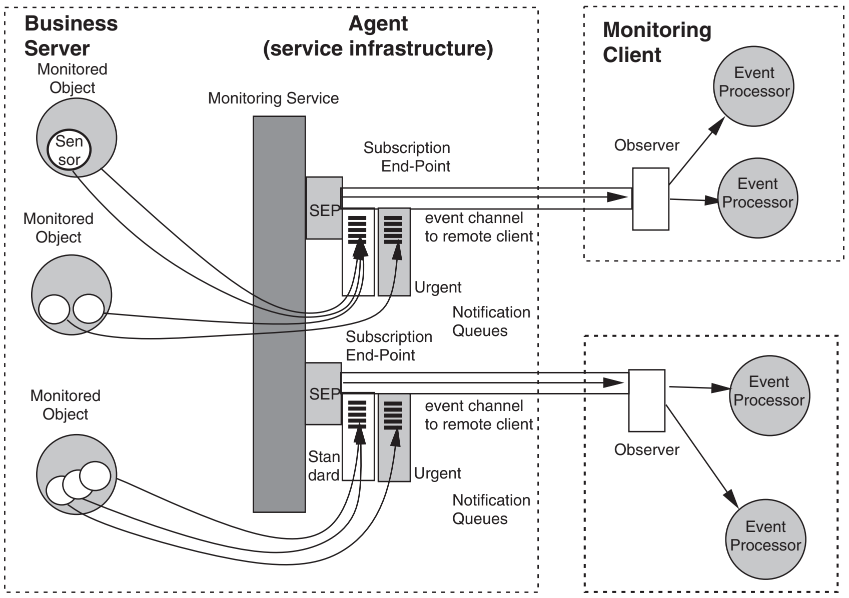


FIGURE 3. Architecture of generation and dissemination of event notifications in the MOOMS monitoring service (push-mode).

while MOOMS is language- and middleware-independent. The two existing implementations of MOOMS are in C++, but over different middleware platforms (CORBA and EPOCHA—a Melita Inc. proprietary solution).

5. IMPLEMENTATION ISSUES OF EVENT GENERATION AND DISSEMINATION LAYERS

This section focuses on technical solutions that should be applied in the implementation of a monitoring service to support an efficient working of its event reports dissemination layer. The advocated solutions are illustrated by example of the MOOMS implementation.

5.1. Efficient internal sorting and distribution of event reports

Intercepting events and generating event reports is the first phase of work of a monitoring service. Generation of an event report technically means that memory for an appropriate programming-language-level structure (record) is allocated and event information and context data are written into its fields. Then there is a subset of monitoring clients subscribed to obtain a particular event report among the potentially many monitoring clients attached to the monitoring-enabled server. Therefore, the second phase is internal event reports' sorting and distribution within the monitoring service infrastructure, where event reports are being scheduled for delivery to

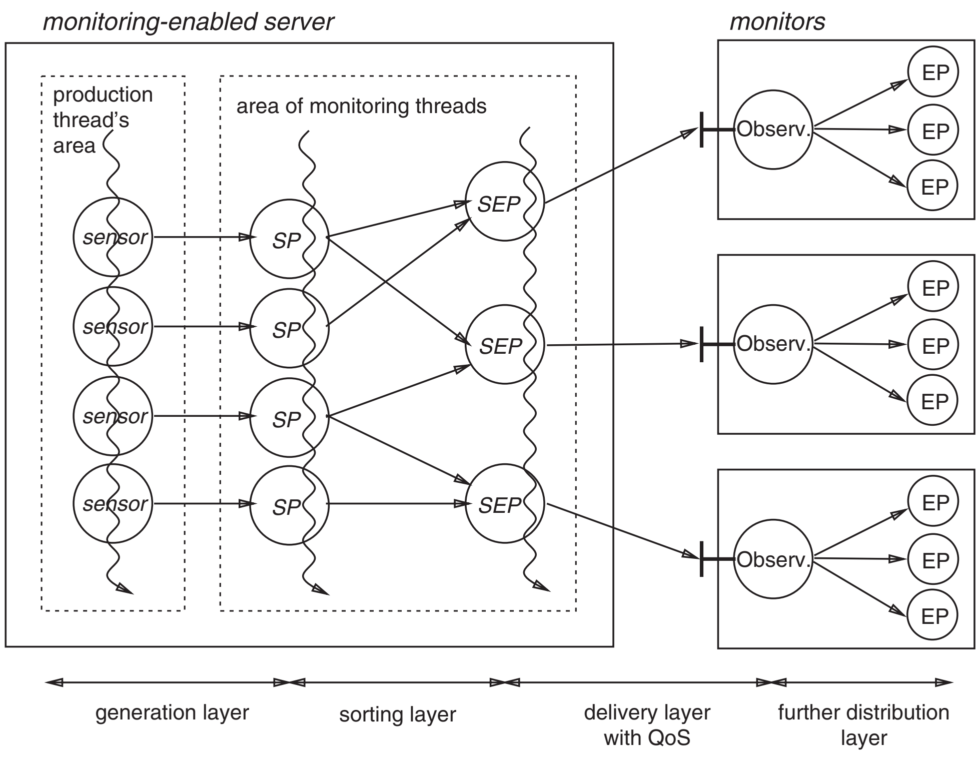


FIGURE 4. Stages of work of a monitoring service plotted onto its simplified architecture.

appropriate monitoring clients. In MOOMS there are the SEPs that represent monitoring clients at the server side, so at the second stage the work of this service is to sort out and pass particular event report structures to appropriate SEP objects in an efficient way, i.e. with a minimal execution-time overhead and avoiding the costly repetitive copying of event report data.

To identify the first problem, consider a single-threaded implementation of the service, with no separate threads to serve the monitoring infrastructure, but with the production thread(s) doing all the job in an instrumented application. In such an implementation, the production thread executing the internal code of a sensor should iterate over a list of registered SEP objects, and invoke a proper delivery operation at each of them, passing a reference to the newly created event report as a parameter to the invocation. Note that in case of a remote observer this requires network communication, and thus can be heavyweight if a guaranteed semantics of delivery is desired (e.g. not CORBA oneway). At the end of this iteration, the production thread could release the memory allocated for the event report structure. This approach is simple, but may potentially impose a heavy overhead (equivalent to a sequence of RPCs) on the production thread.

A slightly more effective solution is depicted in Figure 4 within the borders of the sorting layer. It introduces the notion of a sensor proxy (SP), which is a per-sensor buffer to which the production thread could insert the original event report and immediately return to the business computation. For efficiency, it could just pass a pointer to the report structure instead of copying it to the SP, provided that a release of the allocated memory is further guaranteed in a safe manner. The sensor proxies could be further served by a separate thread or threads, either one per all SPs in a server, or even in a thread-per-proxy model (the latter one, assuming creation of tens or hundreds of threads is unacceptable in a general case). This solution is along the right lines but it comes at some costs. With one proxy thread, e.g. a list of all sensor proxies in the server would have to exist, over which the thread could iterate. This list would cost both in terms of memory and CPU time, as it would have to be updated every time a new MO or sensor is registered in the MIB, and mechanisms for maintaining consistency between this list and the MIB would have to be employed. Moreover, in case of existence of additional specialized threads (e.g. implementing differentiated delivery modes as it is in MOOMS), even this thread may be one too many to assure an effective work of the service on uniprocessor machines with the inevitable presence of context switching, as we will show in the following section.

The second problem is a risk of unnecessary copying of event report data which may occur in case of multiple SEP objects registered in a given sensor, particularly if the burden of distributing the event report is to be handed over to a number of additional threads (consider the responsibility for memory de/allocation). This costly physical replication can be easily avoided if the so called reference-counting smart pointers [19] are used to achieve a safe and automated memory management.[2] This approach is effective, as copying 4 or 6 bytes of a pointer is a very short operation

[2] see smart pointers' descriptions on the web, e.g. http://www.boost.org/libs/smart_ptr/smart_ptr.htm.

for contemporary processors and optimizing compilers, so such work can still be assigned to the production thread. The job of a smart pointer is: (1) to allocate memory for a given object only once, along with setting a counter of references to this object to 1; (2) to convert further attempts to copy the pointed object to incrementation of the reference counter only; (3) to convert attempts to delete the object to decrementation of the reference counter; and (4) finally to delete the object and release the memory only when the reference counter equals zero (which means that nobody is using the pointer any more). This solution helps to avoid repetitive memory allocation (the production thread creates only one copy of each event report structure) and excessive copying of invocation parameters while passing a single event report structure to the queues of potentially many SEP objects (only pointers are copied).

5.2. Processing event report queues and delivering event reports

The key implementational assumption of MOOMS to satisfy the main goal of reduced overhead was to exempt the production thread(s) from the most heavyweight duty in the monitoring service infrastructure—the actual delivery of event reports—by using extra threads for that purpose. Thus, once an event report is created and its smart pointer inserted to the event report queues at appropriate SEP objects, the production thread returns to its business computation. The process of notifying subscribed monitoring clients (according to chosen QoS policies) is performed by one or more separate thread(s), called monitoring thread(s). These threads iterate the FIFO buffers (queues) of event reports, associated with SEP objects. There are two queues at each SEP: one for urgent events and one for regular delivery events, to sort them initially and avoid costly event tagging and searching (see Figure 3).

Several different thread models can be applied to implement processing the event report queues, out of which we have considered the following:

Production thread(s) only, i.e. a model in which no separate monitoring threads are created and the production thread(s) must take on all the monitoring duties, including generation and delivery of event reports.

One monitoring thread in a server: a model in which only one monitoring thread is created to serve all queues in all existing SEP objects. The advantage of this model is that it should work better than others on multiprocessor machines with two CPUs only. It should also work well on uniprocessors in many application scenarios. This model however, has, two disadvantages:

(i) On larger multiprocessor architectures it will not take advantage of the remaining CPUs.
(ii) Fairness and efficiency of the scheduling algorithms to be applied for this thread are controversial. Basically, two variants can be considered, but none of them is truly acceptable. In the first one, the thread iterates over all SEPs and at each of them clears the urgent queue first (by dispatching the event reports to the associated monitoring client) and then turns to do the same for the regular queue; in the second one the thread iterates over the urgent queues at all SEPs first, then switches to make a loop over the regular queues of all SEPs, and goes back to the urgent queues and so on. The first variant does not guarantee the actual priority of the urgent events, while the second one is inefficient (too much polling of empty queues).

Pool of monitoring threads: a fixed number of monitoring threads created to serve the set of SEPs, e.g. each thread to serve a subset of them in one of the two scheduling variants described above. Their drawbacks remain, but scalability is improved and performance is better on a multiprocessor.

Monitoring thread per SEP, i.e. one new monitoring thread is being created when a new SEP is being created (for a newly attaching monitoring client), to serve exclusively both its queues. A sole problem with this solution (to be noted, is shared also with the previously mentioned variants), is implementation of the periodic notification QoS policy while guaranteeing immediate delivery of urgent event notifications at the same time (e.g. the most straightforward solution of putting the monitoring thread to sleep for arbitrary periods of time will result in delaying the delivery of urgent event reports potentially already waiting there in their queues).

Monitoring-thread-per-queue, serving that queue only, i.e. in result a pair of threads is being created for each new SEP (see Figure 3). This approach seems to be the best one, as it enables a straightforward and efficient queue processing procedure: each thread observes one queue only and initially performs a wait (suspend) on a conditional variable associated with the queue's mutex, releasing the CPU; once the production thread puts an event report's smart pointer into the queue and the QoS conditions are met, it sends a signal to wake up the monitoring thread and thus trigger processing the event reports residing in the queue. This solution also scales automatically on multiprocessors, which is an important advantage. Problems may arise only on uniprocessors executing applications with extreme frequencies of events and many observers (and thus many threads), as the performance might be spoiled with excessive context switching. Results of a performance study of this model, done on uni- and multi-processor hardware architectures are given in Section 6.

The advantage of the two latter thread models is that they help to implement the feature of setting a QoS policy on a per-observer basis, i.e. potentially a different policy and conditions per each attached monitoring client.

One more issue concerning threads of the service infrastructure is their priority: should they have equal or lower priority than the production thread(s)? Intuitively, the latter solution could offer better preservation of the production thread's performance. However, much of a true answer depends on the properties of the given operating systems and its thread scheduler, as well as on the frequency of events in a given application. Several tests with high event rates, performed with a MOOMS implementation on Windows NT, have shown clearly that the mechanism of priorities should be used with care. The lower priority for the monitoring thread may lead to its prolonged pre-emption or even starvation, which may in effect cause unacceptable delays in delivery of event reports to the monitoring client.

5.3. CORBA-based implementation of MOOMS

The performance evaluation described in the following section has been accomplished with a CORBA-based C++ implementation of the MOOMS monitoring framework (OmniORB 2.8 [20, 21]), employing the monitoring-thread-per-queue thread model and tested both on a PC under RedHat Linux and Sun SPARC machines under Solaris 7 and 8. The thread interface used was OmniThreads [21]—a C++ class library wrapping the operating systems thread library (on Solaris: POSIX pthreads [22]).

This implementation comprises five IDL interfaces, 29 C++ classes, totalling only about 4500 lines of source code. Hence, in terms of size the implementation is really lightweight.

6. PERFORMANCE STUDY

This section presents selected results of tests performed to evaluate the CORBA-based implementation of the MOOMS monitoring service and to verify some of the solutions concerning its implementation and configuration, postulated in the previous section. The results concern the timing of a series of remote invocation tests, measured at the client side, in a client-server application. The monitored server, the application client and the monitoring client run at a separate computer each, all machines connected to a switched Ethernet LAN.

6.1. Definition of tests

Two basic types of tests were performed:

(i) Monitored attribute update tests.
(ii) Monitored object lifecycle (i.e. creation + destruction) tests.

They were based on invocations of two respective operations from the following IDL interface:

```
interface TestInterface
{
  long UpdateAttribute(in long aParam);
  long CreateAndDestroyMO(in long aParam);
};
```

This interface was implemented by a CORBA object residing in a CORBA server on a dedicated machine, either on a single-processor Sun SPARC Ultra 1 workstation or on a 2- or 3-processors Sun Ultra Enterprise 3000 server, each running a Solaris 8 operating system.

The UpdateAttribute() operation merely increments an integer attribute, triggering the associated sensor to generate one attribute change event, and returns the input parameter. The CreateAndDestroyMO() operation creates a small monitored business object (of class FOO) together with its MO-sensor, adds this sensor to the MIB, and then destroys both objects. The source code for that is shown in the following listing.

```
class FOO
{
public:
#ifdef MONITORED
        MonitoredObject *MO;
#endif
        long item1, item2;
};

long CreateAndDestroyMO(long aParam)
{
  FOO *foo = NULL;
  foo = new FOO;
#ifdef MONITORED
  foo→MO = new MonitoredObject;
  primary.MO→addObject("childFOO1",foo→MO);
  delete foo→MO;
#endif
  delete foo;
  return aParam;
}
```

Note that each newly created MO-sensor object (of the MOOMS library class MonitoredObject) is registered in the MIB at the place determined by its parent monitored object (here pointed by the variable primary) already existing in the MIB. The foo monitored object is given a MIB name 'childFOO1'. Execution of the addObject() method causes generation of an additional event. Then, the foo's MO-sensor and the foo business object are deleted, which causes de-registration of the MO from the MIB and, in consequence, generation of a removal event (generation of deletion events was suppressed in the test).

A single test case involved execution of a test application client making two subsequent remote invocations: one of the UpdateAttribute() and one of the CreateAndDestroyMO() operation. Execution time of each operation was measured separately with a μs accuracy (with the gettimeofday() Unix system call) at the client side and written to a file. For simplicity of presentation, however, the tests' results presented in the following subsections are the total time of a test case, i.e. the time of the UpdateAttribute() plus the time of the CreateAndDestroyMO() operation. They are average values, as each test case was repeated 100 times in a loop. Note that each single run of a test case generated three events in the monitored server: one change of value of an attribute and two MIB structure change events (one MO addition and one MO removal). The well-known effect of the first invocation of a CORBA object being much longer due to managing Object References, initialization of a TCP connection and the ORB itself etc. was bypassed by providing an extra one at each sequence of invocations, and discarding its timing.

Non-instrumented versions of the test application, lacking the elements of the MOOMS infrastructure (the sensors, SEPs etc.), have also been developed (note the preprocessor directives in the source code). They were tested alongside for performance comparison, to estimate the overheads introduced by the monitoring service. Eventually, four different versions of the test application were timing-measured to assess also the effects of other factors, such as multi-threading, number of simultaneous monitoring clients, and QoS policies and parameters, as follows:

(i) An original (non-instrumented) test application.
(ii) A single-threaded implementation (the 'production thread only' model), tested to produce a 'level of reference' for performance comparison with the multi-threaded one.
(iii) A test application instrumented with the principal, multi-threaded version of the service (the thread-per-queue model), with one up to five monitoring clients attached to the test server, observing all events taking place inside (each monitoring client running on a separate dedicated machine). This test was repeated with different QoS policies and different values of QoS parameters in the regular mode of event reports' delivery.
(iv) An instrumented test application without any monitoring client attached (and thus no notifications emitted)—in order to discover the overheads introduced by an 'idle' monitoring infrastructure.

6.2. Results of tests on a uniprocessor computer

Figure 5 presents the results (mean total time of a single test case in μs) of the tests done on a uniprocessor Sun Ultra 1143 MHz, with the batching QoS policy set on and with different batch sizes. The monitoring service was working in the regular mode of delivery. Results are given for 1, 2, 3 and 5 monitoring clients attached to the monitored test server respectively.

Performance of the single-threaded version of the test is given in the bottom row of the table in Figure 5, and thus only the edge results are drawn as level of reference lines in the graph, for its better clarity. For another comparison, the non-instrumented version of the test took 2010 μs, while an instrumented one without a monitoring client took 2325 μs (15% more, which is the overhead cost of an 'idle' service infrastructure, i.e. the one intercepting events, but not emitting any notifications). A relative overhead with regards to the non-instrumented test's performance, expressed in percentage, is given in the third column of the table in Figure 5 (for clarity, calculated for the test with one monitoring client attached only). Its definition is as follows:

$$\text{relative\_overhead} = \frac{\text{time}_i - \text{time}_n}{\text{time}_n},$$

where time_i is the time of performance of instrumented test with monitoring client and time_n is the time of performance of non-instrumented test.

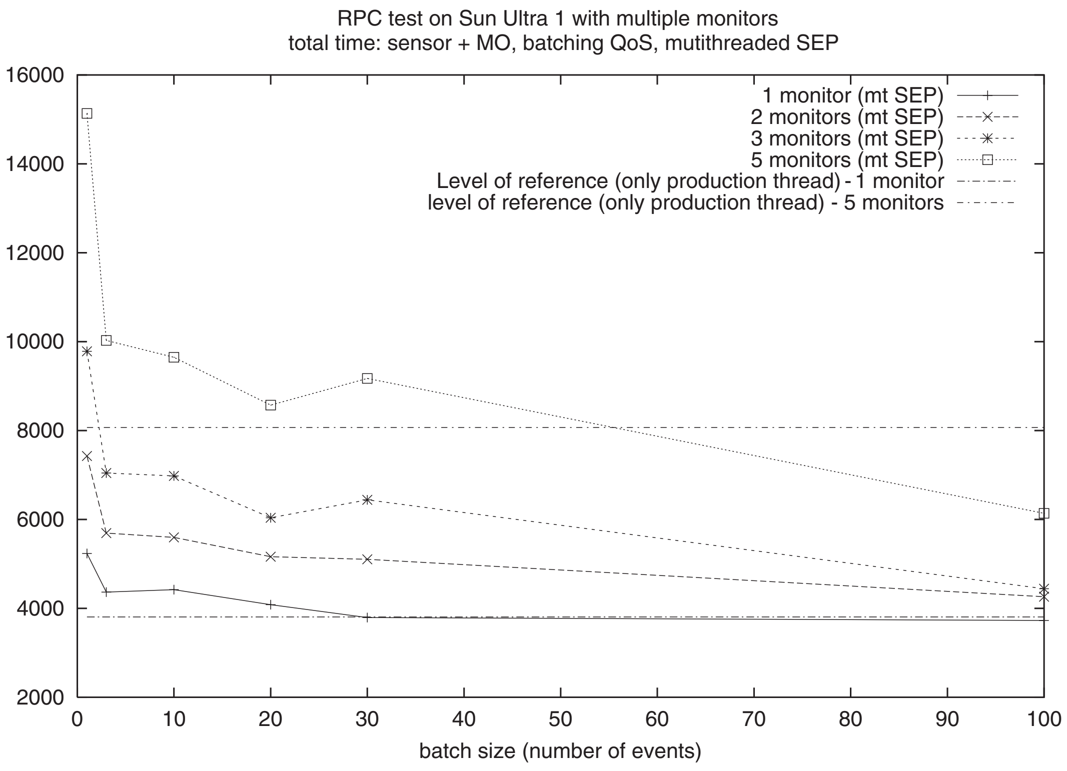
Generally, the results of the multi-threaded version of the service for batch sizes less than 30 are worse than the level of reference (3807 μs) determined by the single-threaded implementation. This overhead is due to inevitable context switches that must be done on this uniprocessor to allow the monitoring thread to access the CPU. Note that the computation to be performed by the production thread at the server side (processing the request, unmarshalling parameters, and marshalling and sending the reply through the network) is long enough for the o.s. thread scheduler to pre-empt it and give timeslots to the monitoring thread during that time. Moreover, the monitoring thread emptying the queue must impose a writer lock on the queue's mutex, thus making the production thread wait before it is able to put new event reports into the queue. Thus, emerging context switches spoil the performance of the multi-threaded service implementation when there is only one processor available. The batching QoS mechanism helps to reduce the resulting additional overhead, but a satisfactory equal level is reached only with batch sizes around 20–30, which are surprisingly high. Note, however, that the business computations used in the tests were trivially short, and as we can assume a more or less fixed timing overhead per notification, the relative overhead with longer real-world computations should be much smaller.

One more rationale behind this test was to verify the scalability of the CORBA MOOMS implementation with regard to the number of monitoring clients. Here, each additional monitoring client took about 1 ms more in the single-threaded version, while the multi-threaded one had results worse (with a range of about 40%) for small batch sizes. Note that one monitoring client more means two threads more in the test application server, and thus more context switching overhead. A big batch size (100) outperformed the results of the single-threaded version with multiple monitoring clients, showing that the benefits of the batching QoS policy may be visible only after its appropriate tuning.

Figure 6 presents results of the same tests done again on a Sun Ultra 1, but with the periodic notification QoS policy set on. Here, the multi-threaded service performs not only much better (1 ms or more) than with the batching QoS on, but also better than the single-threaded level of reference (34–44% versus 89% of relative overhead). This phenomenon can be explained by a different rate of forced context switches and by the multi-threading itself, supported by a beneficial activity of the thread scheduler of the operating system. Note that on a uniprocessor the multi-threaded implementation of the service works in two phases, executed in time-sharing:

(i) phase 1: sensing, generating and buffering (only intercepting the events and intra-server generation of event reports), performed by the production thread;
(ii) phase 2: clearing the SEP queues and delivering event reports to monitoring client(s) (an I/O-intensive phase), performed by the monitoring threads.

In the batching QoS policy there is a 'hard' guarantee of changing the service's activity from phase 1 to phase 2



batch size	1 monitor Total time [μs]	relative overhead with 1 mon.	2 monitors Total time [μs]	3 monitors Total time [μs]	5 monitors Total time [μs]
1	5243	160%	7426	9782	15134
3	4367	117%	5693	7044	10027
10	4419	119%	5597	6978	9649
20	4084	103%	5162	6037	8572
30	3794	88%	5102	6441	9171
100	3728	85%	4263	4442	6138
single-thread ref. level	3807	89%	4916	5994	8068

FIGURE 5. Total time of a test case on a Sun Ultra 1, with batching QoS and multiple monitoring clients.

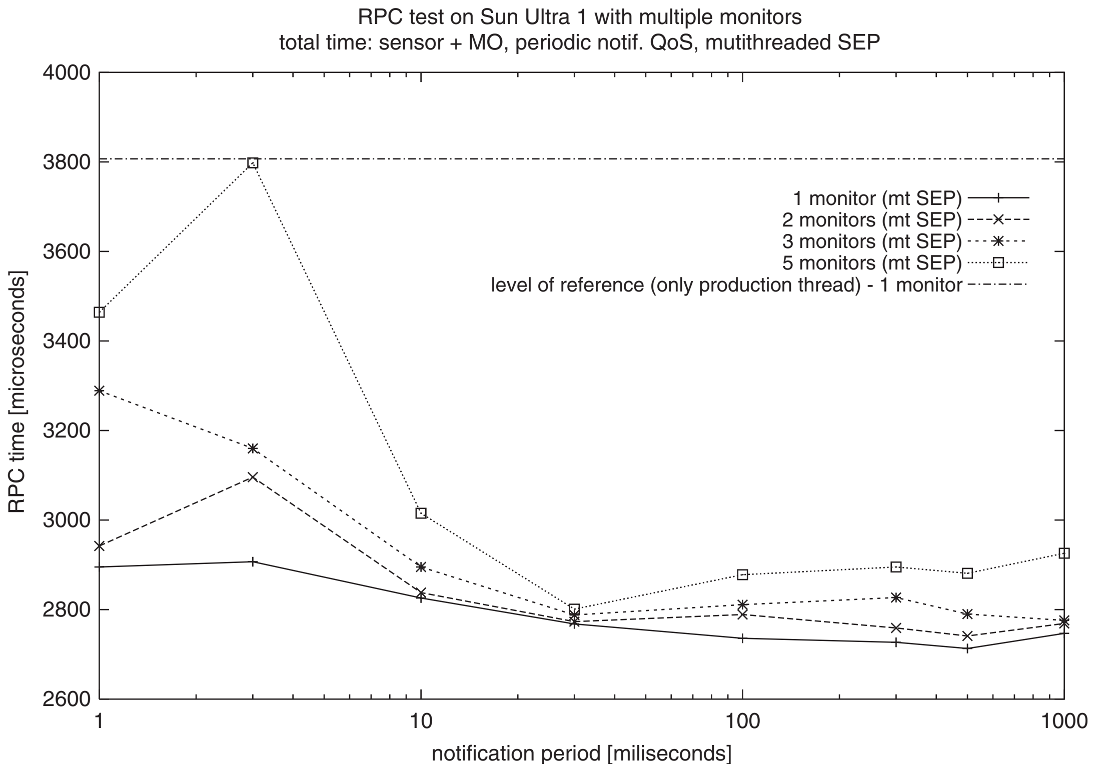
when the batch limit is reached and the threads' priorities are equal, even if the production thread is still busy. On a uniprocessor this means an inevitable context switch between threads during the business computation, which causes an overhead. Therefore, a context switch happens every 3, 10, 30 etc. events, depending on the batch size, i.e. quite often, hence a high overhead.

In the periodic notification QoS policy, a change of phases (thus potentially a context switch) only happens once at the end of each notification period. If the business computation at the test server plus the time needed to dispatch a reply to the test client is still shorter than the notification period, then the operating system may easily schedule the computation (as seen by the business client) in a series of uninterrupted timeslots and thus avoid the context switching overhead in a vast majority of cases. This justifies the advantage of periodic notification over batching QoS policy.

Additionally, an effective implementation of multi-threading and periodic notification in the service infrastructure may allow the monitoring thread to take advantage of the time when the production thread is idle at the server side. That is, the time of the network transmission of a result plus the time to generate a new invocation at the client side and to deliver the request to the server together may easily be enough time for the monitoring thread to do a major part of its job (phase 2) then, while the production thread is idle. On the contrary, in the single-threaded version of the service, the production thread must do all the functions in phase 2, and the client is blocked during all that time. This brings out the multi-threading advantage.

Generally, the resulting overall relative overhead of about 35% on a uniprocessor computer would be acceptable in many application areas.

Note also that with multiple monitoring clients the multi-threaded version with periodic notification scales much better than the single-threaded one. The first remark is that for notification periods longer than 3 ms even with five monitoring clients the results are far below the first level of reference, i.e. much better than in the single-threaded version with one monitoring client only. Also, the notification



notific. period [ms]	1 monitor Total time [μs]	relative overhead with 1 mon.	2 monitors Total time [μs]	3 monitors Total time [μs]	5 monitors Total time [μs]
1	2895	44%	2942	3289	3464
3	2907	44%	3096	3160	3798
10	2826	40%	2838	2895	3015
30	2768	37%	2773	2788	2801
100	2736	36%	2789	2811	2878
300	2727	35%	2749	2827	2895
500	2713	34%	2741	2790	2881
1000	2747	36%	2769	2776	2926
single-thread ref. level	3807	89%	4916	5994	8068

FIGURE 6. Total time of a test case on Sun Ultra 1, with periodic notification QoS and multiple monitoring clients.

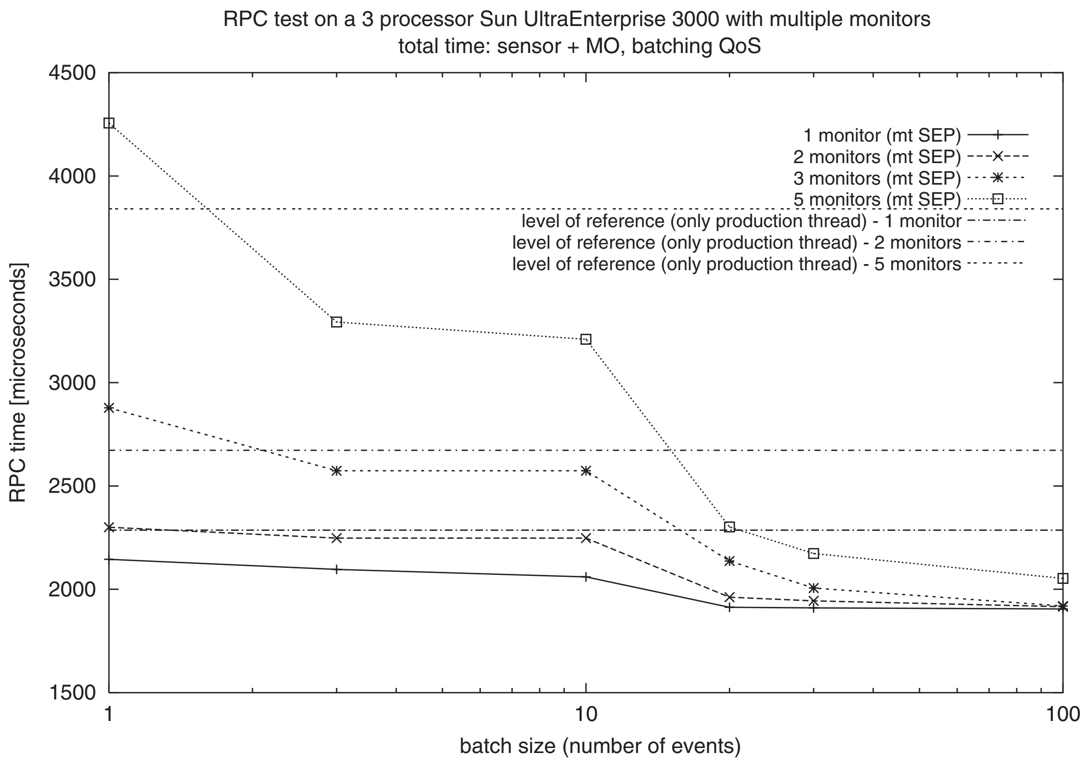
periods longer than 10 ms assure a better stability of the the multi-threaded service's work, as the tests' results are less dispersed. With these notification periods one additional monitoring client takes only tens of microseconds (about 35 μs on average).

This is a clear evidence of the effective work of the mechanism of non-copying smart pointers used to maintain the event report data structures in the implementation of SEP queues, and the multi-threaded model working successfully in the periodic notification QoS in these test conditions. Observation of benefits of these implementational solutions is possible due to the fact that the results of this test are not blurred by the effects of context switching.

6.3. Results of tests on a multiprocessor computer

A truly effective performance of the multi-threaded implementation of a monitoring service can be expected on a multiprocessor machine. The following section describes results of the tests in which the test server was executed on a 3-processors Sun Ultra Enterprise 3000 (3$\times$hyperSPARC 400 MHz). Figure 7 presents results on that machine with the batching QoS policy set on, with 1, 2, 3 and 5 monitoring clients observing the monitored test server. Performance of the single-threaded version of the test is given in the bottom row of the table in Figure 7, however for clarity reasons, only its results with 1, 2 and 5 monitoring clients are drawn as level of reference lines in the graph. A non-instrumented test took 1553 μs, while an instrumented one without a monitoring client took 1662 μs (only 7% overhead cost of a non-emitting service infrastructure).

The results are in line with the expectations. The multi-threaded system performs better than the single-threaded one. For up to three monitoring clients attached, the results are well below the level of reference and the lines are nearly



batch size	1 monitor Total time [μs]	relative overhead with 1 mon.	2 monitors Total time [μs]	3 monitors Total time [μs]	5 monitors Total time [μs]
1	2145	38%	2300	2878	4256
3	2096	35%	2248	2574	3293
10	2060	33%	2149	2486	3210
20	1913	23%	1962	2137	2302
30	1910	23%	1944	2006	2173
100	1905	23%	1916	1919	2053
single-thread ref. level	2286	47%	2673	3101	3841

FIGURE 7. Total time of a test case on a 3-processors Sun Ultra Enterprise 3000 400 MHz, with batching QoS and multiple monitoring clients.

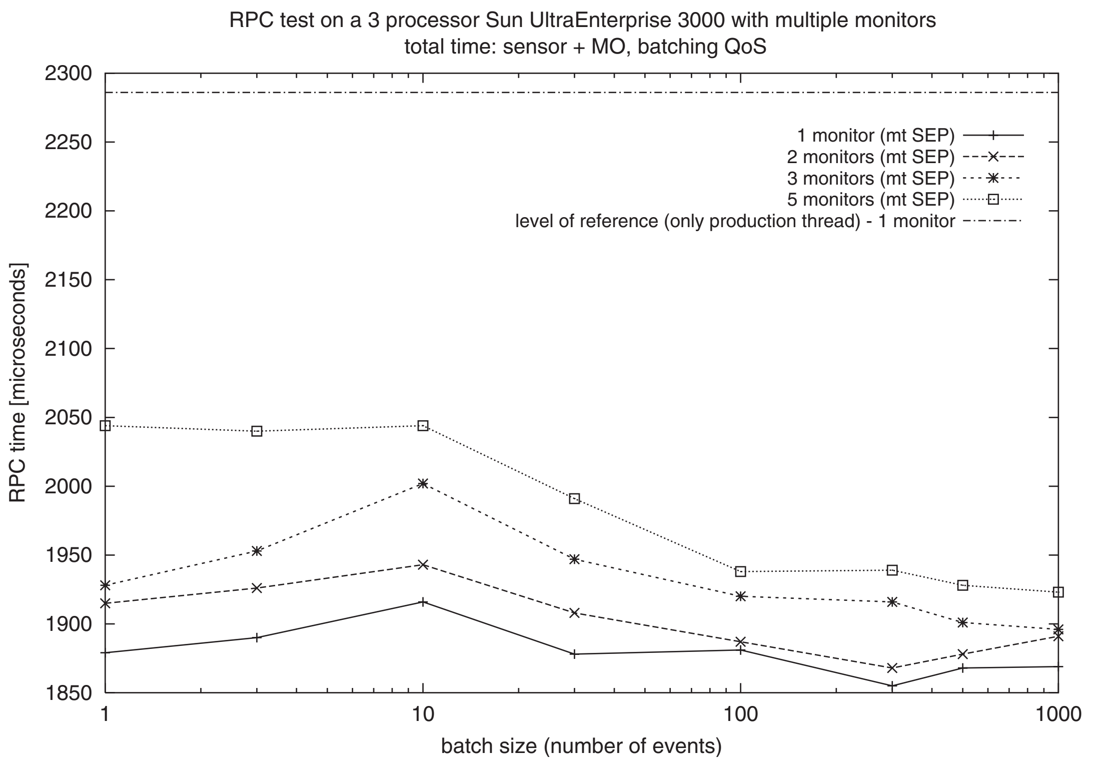
flat, which suggests that each of the active threads has been assigned a separate CPU. For more monitoring clients (and thus more active threads), the results start to deteriorate. This leads to two main conclusions:

(i) As long as there are enough processors to allocate to arising threads, the multi-threaded service performs very well and the overhead related to new monitoring clients is minimal (generally, in tens of μs), also due to the mechanism of non-copying smart pointers used in the implementation of SEP queues.

(ii) When there are more threads than processors, context switches spoil the ideal performance, but the batching mechanism helps to overcome this effect. Even with batch size 3 the performance of all tests is better than their single-threaded level of reference, including the case with five monitoring clients. This is much less than that was needed on a uniprocessor with a three-times slower clock. An obvious conclusion is that multi-threaded software architectures require multiprocessor machines and good OS thread schedulers to scale well and allow batching QoS to work properly.

Tests with a bigger number of monitoring clients have not been performed as they are quite easily predictable and the problems with scalability of heavily multi-threaded applications have been well described in literature [22, 23]. In an application with a huge number of monitoring clients attached, even the available number of CPUs may not be enough and the monitoring-thread-per-queue implementation may turn out inefficient and non-scalable. In such application scenarios a single monitoring thread or a pool of monitoring threads (= number_of_available_processors—number_of_production_threads) should perform better, depending on the underlying hardware architecture. A CORBA Event Service or Notification Service could also be used to deliver notifications in such cases.

Figure 8 presents tests' results on the Sun Ultra Enterprise 3000 with the periodic notification QoS policy and 1, 2, 3



notific. period [ms]	1 monitor Total time [μs]	relative overhead time [μs]	2 monitors Total time [μs]	3 monitors Total time [μs]	5 monitors Total time [μs]
1	1879	21%	1915	1928	2044
3	1890	22%	1926	1953	2040
10	1916	23%	1943	2002	2044
30	1878	21%	1908	1947	1991
100	1881	21%	1887	1920	1938
300	1855	19%	1868	1916	1939
500	1868	20%	1878	1901	1928
1000	1869	20%	1891	1896	1923
single-thread	2286	47%	2673	3101	3841

FIGURE 8. Total time of a test case on a 3-processors Sun Ultra Enterprise 3000 400 MHz, with periodic notification QoS and multiple monitoring clients.

and 5 monitoring clients observing the monitored test server respectively. Here, the differences between results are very small and the characteristics of the graph is nearly flat (note the resolution of the y-axis), at least for one and two monitoring clients attached. An effective parallelism, a minimal rate of context switches and a good work of the Solaris thread scheduler result in a relatively small overhead—within the range of 20%. It probably could be even smaller, if further optimization of the MOOMS service implementation is done, e.g. the expensive CORBA. Any type used for implementation of the variable part of the event report was replaced with a much more efficient IDL union or a family of fixed compile-time definitions of event report structures.

7. CONCLUSION

There seem to be no significant structural differences between MOOMS and JMX, apart from the fact that MOOMS offers only a subset of the JMX functionality. Thus, similar implementational solutions and mechanisms could be used in JMX to provide good performance of its monitoring information dissemination layer and the results of MOOMS's performance evaluation might help to formulate several guidelines for more efficient implementations of JMX and other monitoring frameworks.

The presented performance study has confirmed that the cost of dissemination of event reports to remote monitors is the main source of execution-time overhead introduced by the monitoring service infrastructure. Multithreading used to relieve the production thread of actual delivery of monitoring information helps in a majority of configuration settings, particularly well over multiprocessor hardware architectures (as long as the total number of production and monitoring threads does not outnumber the available CPUs too much). Only with extremely intensive computations (with infrequent idle time periods) performed on uniprocessors, usage of a single-threaded service implementation could be recommended.

Batching and periodic notification QoS mechanisms ruling notifications' dissemination should assist a multithreaded implementation in reducing overheads. The key issue is to choose a particular QoS policy that adapts best to the underlying hardware platform and application scenario, and to tune the chosen mechanism with appropriate parameters, i.e. batch size or notification period, to achieve best performance results. Such a tuning should be careful, as the choice must be conscious of possible consequences. Note that both mechanisms eventually amount to buffering notifications and delaying their delivery, i.e. from the monitoring client's user's point of view, they both deliver sequences of event reports, either fixed (batching) or of unpredictable arbitrary length (periodic notification). For example, using large batch sizes (to decrease the number of context switches on a uniprocessor) may be dangerous, because some event reports may not be delivered to the monitor at all or too late if the number of generated event reports is less than the batch size limit and suddenly there is a break in generation of monitored events and the flush on timeout is not implemented or the timeout period set is too long.

Periodic notification is a safer QoS mechanism in this respect and additionally it gives better timing results on both uni- and multi-processors, as it takes advantage of idle CPU cycles with some help of the system's thread scheduler. However, by choosing this QoS policy, the monitoring system's user must be prepared for large bursts of notifications coming if the event generation rate is high. Fortunately, mechanisms like filtering at the client side may help to overcome this problem, eventually adapting the notification rate at the user interface to the capabilities of human perception.

Last but not least, avoiding repetitive copying of monitoring information inside a monitored server, on the way from generation of an event report until its dissemination to multiple monitoring clients, is also essential, particularly if all this is done by the CPU shared with the production thread(s) and other monitoring thread(s). The actual mechanism to be used for that purpose depends upon the programming language of the service implementation.

REFERENCES

[1] Sun Microsystems Inc. (2002) *Java Management Extensions Instrumentation and Agent Specification*, v1.1. http://java.sun.com/products/JavaManagement.

[2] Feldkuhn, L. and Erickson, J. (1989) Event management as a common functional area of open systems management. In *Proc. IFIP 6.6 Symp. on Integrated Network Management*, Boston, MA, USA, pp. 365–376. North-Holland.

[3] Mansouri-Samani, M. and Sloman, M. (1994) Monitoring distributed systems. In Sloman, M. (ed.), *Network and Distributed Systems Management*, chapter 12. Addison-Wesley.

[4] Laurentowski, A. and Zieliński, K. (1999) A synergistic approach to monitoring distributed software components. In *Proc. 3rd European Research Seminar on Advances in Distributed Systems (ERSADS'99)*, Madeira, Portugal, University of Lisboa, BROADCAST, Swiss Federal Institute of Technology (EPFL), Lausanne.

[5] Laurentowski, A. (2001) *A Distributed Monitoring Service for Software Objects*. PhD Thesis, University of Mining and Metallurgy in Kraków, Poland.

[6] Borland Corp. (2001) *AppCenter*. http://www.borland.com/appcenter/

[7] Zieliński, K., Laurentowski, A., Szymaszek, J., and Uszok, A. (1995) A tool for monitoring heterogeneous distributed object applications. In *Proc. 15th Int. Conf. on Distributed Computing Systems*, Vancouver, Canada, May, pp. 11–18. IEEE CS Press.

[8] Kaelbling, M. and Ogle, D. (1990) Minimizing monitoring costs: choosing between tracing and sampling. In *Proc. 23rd Int. Conf. on System Sciences*, January, pp. 314–320. IEEE Computer Society Press.

[9] Schroeder, B. A. (1995) On-line monitoring—a tutorial. *IEEE Comp.*, pp. 72–78.

[10] Object Management Group. (2000) *Notification Service Specification*. New Edition, v.1.0.

[11] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1994) *Design Patterns*. Addison-Wesley.

[12] Stallings, R. (1993) *SNMP, SNMP v2.0 and CMIP. A Practical Guide to Network Management Standards*. Addison Wesley.

[13] Seigneur, J.M. (2003) *An overview of DMTF and CIM* www.cs.tcd.ie/Jean-Marc.Seigneur/Cim/Overview.htm.

[14] Debusman, M. and Kroeger, R. (2001) Widening Traditional Management Platforms for Managing CORBA Applications In *Proceedings of Distributed Applications and Interoperable Systems Conference*, Kraków, Poland 17–19, pp. 245–256. Kluwer Academic Publishers.

[15] The Open Group Systems (2002) *Systems Management: Application Response Measurement API ver. 2.0* Technical Standard www.opengroup.org.

[16] Object Management Group. (1997) *Event Service Specification*. OMG doc. 97-12-11.

[17] Segue Software Inc. (1999) *Silk for CORBA—a White Paper*. http://www.segue.com/.

[18] Rackl, G., Lindermeier, M., Rudorfer, M. and Suess, B. (2000) MIMO—an infrastructure for monitoring and managing distributed middleware environments. In Sventek, J. and Coulson, G. (eds), In *Proc. Middleware 2000—IFIP/ACM Int. Conf. on Distributed Systems Platforms*, Lecture Notes in Computer Science 1795, April, pp. 71–87. IFIP/ACM, Springer Verlag.

[19] Stroustrup, B. (1997) *The C++ Programming Language* (3rd edn). Addison-Wesley.

[20] Lo, S.-L. and Pope, S. (1998) The implementation of a high performance ORB over multiple network transports. In *Proc. Middleware'98 Conf.* Also appeared in a special issue of the *Distributed Sys. Eng. J.*

[21] Lo, S.-L. and Riddoch, D. (1999) *The OmniORB2 version 2.8, User's Guide*. AT&T Research Laboratories, Cambridge (formerly: Olivetti & Oracle Research Laboratory). www.uk.research.att.com/omniORB/omniORB.html.

[22] Nichols, B., Buttlar, D. and Farrell, J. P. (1996) *Pthreads Programming: A POSIX Standard for Better Multiprocessing*. O'Reilly & Associates, Inc.

[23] Schmidt, D. C., Stal, M., Rohnert, H. and Buschmann, F. (2000) *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Wiley & Sons.