An Event Notification Framework based on Java and CORBA

M. TOMONO
1st Transmission Division, NEC Corporation
1753 Shimonumabe, Nakahara-Ku, Kawasaki, 211
JAPAN
tomono@trd.tmg.nec.co.jp

Abstract

Event notification is essential in network management. Many components in a distributed NMS need an event notification mechanism, and so an event notification framework is expected to facilitate the development of the components. CORBA is a promising platform to build such a framework, but its performance is not sufficient for monitoring large-scale networks. This paper proposes an event notification framework that has high performance and customizability. The framework is based on Java and CORBA, and has customization points at which event handling capabilities can be added. Java gives the framework flexibility in the sense that event handling capabilities can be added and replaced dynamically by using downloadable Java programs. High throughput of more than 1000 events per second has been achieved by event batching.

Keywords

Event Notification, Network Management, Java, CORBA, Performance Evaluation, Object-oriented Framework

1. Introduction

The increasing size and complexity of managed networks require network management systems (NMSs) to be distributed and flexible. CORBA is considered to be a promising platform to build distributed NMSs, and CORBA-based NMS architectures have been discussed [7, 6, 10]. CORBA is a distributed object middleware standard, on which client objects invoke operations on server objects in a location transparent fashion. CORBA provides various services named CORBAservices that are useful in developing distributed applications [8]. These features are expected to facilitate the development of distributed NMSs, and also to give NMSs high flexibility and scalability.

Event notification informs network operators of failures and state changes of managed networks. Since more than 1000 events can occur in an event rush because of failure propagation among network elements, NMSs need high

(c) 1999 IFIP

performance. The CORBA Event Service included in the CORBAservices can be used to implement event notification in CORBA-based NMSs, but the services are built on top of the CORBA remote invocation mechanism, which takes milliseconds on standard-class computers. This implies that the naive use of the Event Service results in notification throughput that is not sufficient for monitoring large-scale networks. Thus, the implementation of high-performance event notification is crucial in developing CORBA-based NMSs.

On the other hand, the customizability of event notification is important from the point of view of software productivity, since many components in distributed NMSs are needed to have various event handling capabilities, e.g., event forwarding, format conversion, event classification, event filtering, and so on. The CORBA Notification Service, which is an extension of the Event Service, provides capabilities including event filtering and QoS [9], but does not provide an interface to add general event handling capabilities which NMS components need. An event notification platform that has a general interface for customization would bring the high productivity in developing NMS components.

We propose an event notification framework that has high performance and customizability. Implemented as an object-oriented framework [1], the framework has customization points at which event handling capabilities can be added. Since the framework is implemented in Java, it has flexibility in the sense that event handling capabilities can be added and replaced dynamically by using downloadable Java programs. High throughput of more than 1000 events per second has been achieved by batching events into a sequence and invoking a sending operation for the sequence instead of for each event.

Section 2 presents event notification in CORBA-based NMSs and its issues. Section 3 illustrates the architecture of the proposed event notification framework. Section 4 shows the results of performance experiments. Section 5 discusses the merits and issues of the framework followed by the concluding remarks.

2. Event Notification in Distributed NMSs

2.1 A Distributed NMS

Figure 1 shows a basic architecture of distributed NMSs. The system consists of clients and servers, which are connected using CORBA. Each NMS client comprises GUI components with which network operators monitor and control managed networks. NMS Servers are connected with network elements (NEs) by particular management protocols, e.g., CMIP, SNMP, and store managed objects (MOs), which model NEs. Servers receive events from NEs and disseminate them to clients. Clients display the received events on the screen in appropriate fashions.

Both servers and clients comprise a variety of components, most of which behave as an event supplier, an event consumer, or both. This means that many NMS components are needed to have the capabilities of event generation, event queuing, and event dissemination.

Figure 2 illustrates a detailed architecture. All the components are connected using CORBA except for the connections between NEs and Gateways. This paper explains the components focusing on the event notification aspect, although NMSs have two kinds of control flows, request flows from clients to servers and event notification flows from servers to clients.

(c) 1999 IFIP

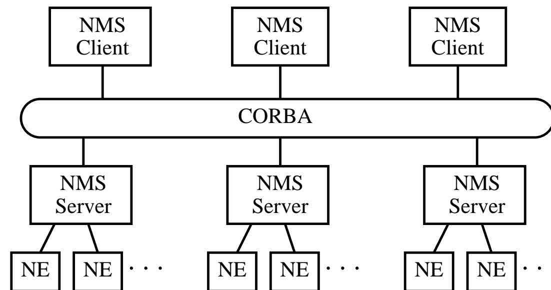

Figure 1: A Distributed NMS

- Gateway
  A gateway mediates between NEs and an alarm manager. NEs generate and emit events in various formats depending on NE types, while a gateway receives the events and converts them into the common format that an alarm manager accepts.

- Alarm manager
  An alarm manager is responsible for classifying, storing, and disseminating events. It receives events from gateways and forwards them to clients and an MO manager. It gives an identifier to each event and classifies events into alarms, state changes of NEs, and the others. It also finds the correspondence between an event that informs a failure and an event that informs the restoration of the failure. Alarm correlation is another task of alarm managers.

- Log manager
  A log manager stores events as a log. It receives events from an alarm manager and stores them into a database. It responds to log retrieval requests from clients.

- MO manager
  An MO manager stores MOs in a database and provides operations for manipulating the MOs. It forwards the events received from an alarm manager to a map manager, converting the event format into the one that can be accepted by the map manager.

- Map manager
  A map manager stores network maps, which describe network topologies using symbols, links, and background pictures. Browsing the network maps, operators identify the MOs that they must monitor and control. A map manager receives events from an MO manager and changes the color of the symbols corresponding to the MOs that are involved in the events. The manager propagates the color changes from the bottom to the top of the network map hierarchy. The manager generates events that inform the symbol color changes, and disseminates them to clients.

- Alarm table
  An alarm table receives alarms from alarm managers, and lists them on the screen.

(c) 1999 IFIP

- Mapview/Treeview

  A mapview displays network maps on the screen, and a treeview displays the hierarchy of network maps. Both of them receive symbol color change events from map managers and change the symbol colors on the screen to help a network operator identify the MOs at which failures have occurred.

Gateways, alarm managers, MO managers, and map managers are needed to have the capabilities of event generation, format conversion, queuing, dissemination, and so on. The other components need event queuing capability in order to avoid bad influence on event suppliers by buffering events.

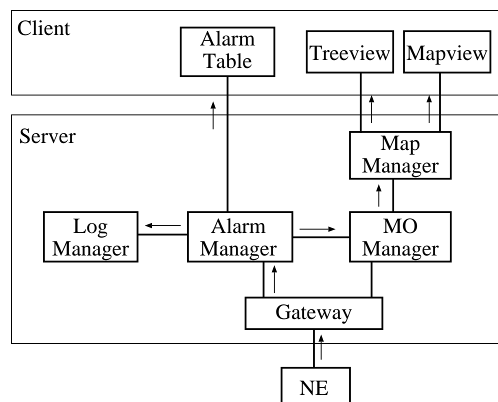

Figure 2: A Distributed NMS Architecture

2.2 CORBA Event Service

The CORBA Event Service provides a framework of asynchronous event notification [8]. Figure 3 illustrates its architecture. The Event Service consists of suppliers, consumers and event channels. Suppliers emit events and consumers receive events. An event channel is a mediator which receives events from suppliers and forwards them to consumers.

The Event Service has two notification models, push model and pull model. In the push model, suppliers put events to consumers, invoking consumers' push operation. In the pull model, consumers get events from suppliers, invoking suppliers' pull operation.

An event channel comprises proxy suppliers and proxy consumers. PPC and PPS in Figure 3 stands for *ProxyPushConsumer* and *ProxyPushSupplier* respectively (note that Figure 3 describes the push model). PPCs receives events from push suppliers connected to the event channel, and transmit them to PPSs, which send them to push consumers connected to the event channel.

The CORBA Notification Service extends the Event Service with capabilities including event filtering, structured events, quality of service, and so on [9]. Application developers utilize these capabilities through the interfaces of the Notification Service.

(c) 1999 IFIP

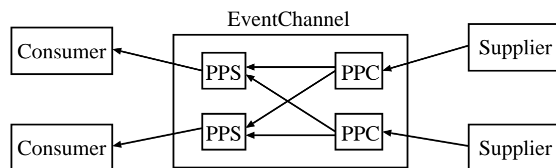

Figure 3: CORBA Event Service

2.3 Issues

Since many NMS components in servers and clients must have the event handling capabilities as mentioned above, a common platform of event notification can benefit the productivity of developing the components. The CORBA Event Service is a promising one, but it does not provide an interface to add event handling capabilities which NMS components need. Thus, a platform that has customizability to add event handling capabilities necessary for NMS components is needed. The following are the major requirements of the platform.

- Customizability
  The platform must be easy to customize according to the requirements of each NMS component. It is important to separate customization points from the common structure and to specify the interface of the customization points.
- Flexibility
  The reconfiguration of a distributed NMS costs much, because servers and clients are distributed over many computers. Moreover, since event notification is one of the most critical functions of NMSs, stopping the system for reconfiguration should be avoided. Hence, the platform needs flexibility for dynamic reconfiguration with low cost.
- Performance
  The naive implementation of event notification using CORBA remote method invocation, which takes milliseconds, is not sufficient to monitor large-scale networks, especially when an event rush occurs because of failure propagation among network elements.

Customizability and flexibility are involved with the productivity of developing distributed NMSs. An object-oriented framework based on Java offers a solution. Customizability is provided by an object-oriented framework, which is a reusable, semi-complete program that can be specialized to produce custom software [1]. An object-oriented framework has the basic structure, and an application developer can customize the framework by adding new objects to it.

Java brings flexibility. Java programs can be downloaded through networks and be linked dynamically without stopping the system. By utilizing this feature, event handling capabilities can be added and replaced dynamically through networks. This kind of dynamic configuration is widely used in network management. An example is *Management by delegation* [2], where programs are delegated to a remote site and executed there in order to improve the performance and flexibility of the system [3, 6, 12, 13, 14]. Dynamically-configurable event services have also been developed focusing on event filtering or alarm correlation [11, 15].

(c) 1999 IFIP

Java programs, however, suffer from poor performance because they are executed on a bytecode interpreter. They run several times slower than C++ programs. Although the problems would be solved by technologies including just-in-time compiler, the performance issue is the major hurdle in developing the event notification framework based on Java and CORBA. Event batching is a method of enhancing notification throughput, where events are batched into a sequence and are disseminated by one operation for the sequence.

3. Event Notification Framework

3.1 Architecture

As mentioned above, an event notification platform needs customizability, and an object-oriented framework is a solution. An object-oriented framework is the skeleton of an application that can be customized by an application developer [4, 5]. In the case of the event notification platform, the skeleton has the common event handling capabilities including receiving, queuing, and disseminating events, while the customization points are the particular event handling capabilities including format conversion, event identification, alarm correlation, and so on. Hence, the notion of object-oriented framework fulfills the customizability requirement. We develop the event notification platform as an object-oriented framework, and refer to it as an event notification framework.

The event notification framework comprises a set of objects which form the skeleton with customization points called hooks. A hook is the specification of a customization point and the framework is customized by attaching an object to a hook. We refer to the object as hook object. A hook has a default hook object, which can be replaced by another object that has the interface specified by the hook. In general, default hook objects have no capabilities, and the framework with only default hook objects behaves as a CORBA event channel. A variety of event handling capabilities can be added to NMS components just by replacing default hook objects with particular objects which provide the capabilities.

Figure 4 illustrates the architecture of the event notification framework. It consists of PPCs, PPSs, and two kinds of hooks. A CHook, which stands for consumer hook, is placed between a PPC and PPSs. An SHook, which stands for supplier hook, is placed after a PPS. By attaching hook objects to CHooks and SHooks, the framework can be customized for various purposes. It is possible for each CHook to have a different object as well as for all CHooks to share an object. The former case is used when different operations are needed depending on the suppliers. The latter case is used when the same operation is executed for all events. Hook objects that are different depending on consumers are attached to SHooks.

Hook objects include the following examples. *Event filters* delete events that satisfy the given conditions. Alarm managers, log managers and map managers can have a event filter. *Format converters* change the data formats of events. A gateway has a format converter that translates events described in particular management protocols into the system's internal format. *State updaters* change MO's attribute values according to the contents of events. An MO manager has a state updater that changes the operational state of an MO when it receives an event that describes a failure of the NE that the MO represents. *Event converters* change event contents. For example, an alarm manager classifies the source MOs of notified events into several categories and changes event contents according to the classification. *Alarm correlators*

(c) 1999 IFIP

eliminate redundant events by extracting cause events. This needs to traverse the links representing relationships between MOs.

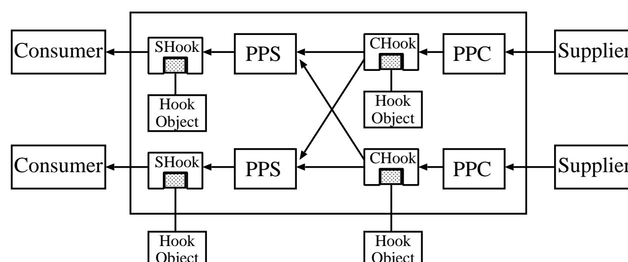

Figure 4: Event Notification Framework

3.2 Dynamic Configuration

The downloadability of Java programs enables the dynamic configuration of the event notification framework. By downloading hook objects written in Java and adding them to the specified hooks, the framework can be customized without stopping the system.

An example of CORBA IDL (Interface Definition Language) for dynamic configuration is as follows.

```
interface HookManager {
  void addHookObject(String name, String location, String hook);
  void removeHookObject(String name, String hook);
}
```

The argument `name` means the class name of the hook object. The argument `location` means the URL that describes the machine, directory, and file name where the class code is stored. The argument `hook` means the hook to which the object is attached.

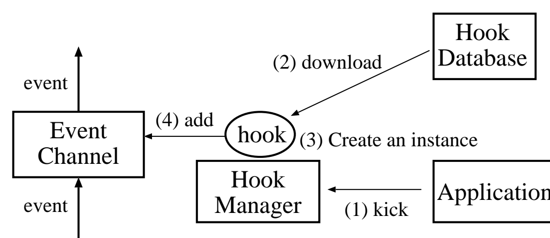

Figure 5: Dynamic Configuration of Hook Operations

Figure 5 depicts the procedure of adding an object to a hook. (1) The application invokes an addHookObject operation of the hook manager. (2) The hook manager downloads the class code specified by `name` from the hook database specified by `location`. (3) The hook manager creates an instance

(c) 1999 IFIP

of the class from the downloaded class code. (4) The hook manager adds the instance to the specified hook.

3.3 Event Batching

The naive implementation of event notification using CORBA remote invocation mechanism is not sufficient for large-scale managed networks. While one remote invocation takes milliseconds on standard-class computers, more than 1000 events can occur in an event rush because of failure propagation among network elements.

The performance of CORBA remote method invocation is affected by two factors, the number of remote invocations and the amount of transferred data. An experiment shows the former has a great influence on the performance (see Section 5). Therefore, reducing the number of remote invocations will improve the event notification performance.

The event-batching method proposed by the CORBA Notification Service is a solution. The method packs events into a sequence and invokes a sending operation for the sequence instead of for each event. The method specifies two major parameters: *maximum batch size* and *pacing interval.* Maximum batch size means the maximum length of sequences in which events are batched. If the size is large, the CPU time spent by one sequence becomes long, which causes a long wait during which a supplier is blocked in case of multiple suppliers. On the other hand, if the size is small, the effect of batching events is obviously small. Thus, there is the optimal value. Pacing interval means the maximum period of time a supplier or an event channel batch events into a sequence. If this value is large, the delay of event arrival becomes large. If this value is small, the number of batching events is also small. Thus, there is the optimal value.

4. Performance Evaluation

4.1 Performance Analysis

This section presents the performance analysis of the event notification framework. In the event channel in Figure 3, we suppose each PPS has a queue, to which PPCs put data. Each PPS has a thread, which gets data from its queue and sends it to consumers. PPCs are controled by CORBA's thread.

We use the following parameters. n_1 is the number of PPCs. n_2 is the number of PPSs. t_1 is the time for a PPC to receive a sequence, in which events are batched, from a supplier. t_2 is the time for a PPS to send a sequence to a consumer. t_3 is the time for a PPC to send a sequence to a PPS. τ is the total processing time. τ_1 is the total proceeing time of one PPC in τ. τ_2 is the total proceeing time of one PPS in τ. τ_w is the total time the event channel does indirect tasks in τ including the time to wait for data arrival, the time spent for thread synchronization and memory management, and so on. We suppose the event channel runs on one CPU. t_1 and t_2 do not include the processing times of the other CPUs on which suppliers and consumers run. m is the batch size. p_1 is the number of sequences that a PPS receives in τ, and p_2 is the number of sequences that a PPS sends in τ. The following relationships hold.

(c) 1999 IFIP

$$p_1 = \frac{\tau_1}{t_1 \cdot 1 + t_3 \cdot n_2} \cdot n_1 \cdot m \quad (1)$$

$$p_2 = \frac{\tau_2}{t_2} \cdot m \quad (2)$$

$$\tau = \tau_1 \cdot n_1 + \tau_2 \cdot n_2 + \tau_w \quad (3)$$

The denominator of (1) is the time for a PPC to receive a sequence from a consumer and to deliver it to n_2 PPSs. τ_1 divided by the denominator is the number of sequences which a PPC delivers to each PPS in τ_1 (equivalently in τ). Thus, (1) means the number of individual events that a PPS receives from n_1 PPCs in τ.

t_1 comprises the following factors. t_{c1} is the time for a CORBA remote invocation (not including marshaling or unmarshaling). t_{d1} is the time for an individual event to transfer between two machines including marshaling and unmarshaling. t_2 comprises t_{c2} and t_{d2} in the same fashion. t_{ch} is the time for a PPC Hook operation to be executed for an individual event. t_{sh} is the time for a PPS Hook operation to be executed for an individual event.

$$t_1 = t_{c1} + m \cdot t_{d1} + m \cdot t_{ch} \quad (4)$$

$$t_2 = t_{c2} + m \cdot t_{d2} + m \cdot t_{sh} \quad (5)$$

When the event channel works without a jam, the equilibrium of input and output holds for each PPS, that is, $p_1 = p_2$. By letting $p = p_1 = p_2$, we obtain throughput p/τ by removing τ_1 and τ_2 from these equations.

$$\frac{p}{\tau} = \frac{1 - \frac{\tau_w}{\tau}}{(\frac{t_{c2}+t_3}{m} + t_{d2} + t_{sh}) \cdot n_2 + \frac{t_{c1}}{m} + t_{d1} + t_{ch}} \quad (6)$$

In general, t_{ci} $(i = 1, 2)$ is much larger than t_{di} in the case that an individual event size is not very large. Thus, t_{ci} is dominant when m is small, but t_{d1} and t_{d2} become dominant when m is large. This shows the basis of the event-batching effect.

Note that equation (6) is an approximation. The crucial factor is τ_w, which can be affected by n_1 and n_2. When n_1 and n_2 are large, τ_w can be large because of overheads such as thread synchronizaiton and memory management, and so the throughput can be small. $\tau_w = 0$ is the optimal situation, where the throughput is the maximum.

4.2 Experimental Environment

We have conducted performance experiments on throughput and memory consumption. The environment was as follows. Suppliers ran on a single-CPU Pentium 120MHz computer with 80MB memory running Windows 95 (machine A). An Event Channel ran on a single-CPU PentiumII 266MHz computer with 64MB memory running Windows NT 4.0 (machine B). Consumers ran on a single-CPU PentiumII 233MHz computer with 95MB memory running Windows 95 (machine C). These computers were connected by a 10Mbps Ethernet segment which is isolated from other segments. The Java VM was

(c) 1999 IFIP

JDK1.1.5 with Symantic's JIT compiler, and the ORB was Visibroker for Java 3.0. Each supplier on machine A had a thread, so did each consumer on machine C.

The event data type used in the experiments was defined as a struct with five member items in CORBA IDL. The data type of the items are *long, string, string, struct A, struct B*. Struct A has two *string* type member items, and struct B has three *string* type member items. The sizes of all the data were 100 bytes in application programs, although they become larger on the memory.

4.3 CORBA Remote Invocation

First, we evaluated the performance of CORBA remote method invocation. We measured the time to transfer one event data between two machines, and the time to transfer 20 events batched into a sequence.

Transferring one event from machine A (supplier) to machine B (event channel) took 6.0 msec, and that from machine B to machine C (consumer) took 4.8 msec. Transferring 20 events from machine A to machine B took 18.7 msec, and that from machine B to machine C took 11.4 msec. The results imply transferring 20 events one by one will take 100 milliseconds or more, which means the number of remote invocations has a great influence on the performance.

4.4 Throughput of Basic Notification

The throughput of basic notification is shown in Figure 6 (a). Events are transferred one by one, not batched. No hook objects are added to the event channel.

The throughput of more than 90 events per second is obtained when the number of consumers is one. The throughput is almost independent of the number of suppliers while the increase of consumers reduces the throughput.

The dashed line represents the prediction based on equation (6) in the ideal condition $\tau_w = 0$. Here, we let $t_{ch} = 0, t_{sh} = 0$ (no hook objects), $m = 1$, $t_3 = 0$ (ignored). It is difficult to evaluate the exact values of t_1 and t_2. We estimate these values by dividing the data transfer times mentioned in Section 4.3 according to the ratio of the two machines' CPU powers. For example, the time to transfer one event from machine A (120 MHz CPU) to machine B (266 MHz CPU) was 6.0 msec, and so t_1 is estimated to be $6.0 \times 120/(266 + 120) = 1.87$ msec. Likewise, t_2 is estimated to be 2.24 msec.

Looking at Figure 6 (a), when the number of consumers is small, the gap between the prediction and the experimental result is large. The reason would be that τ_w becomes large since the CPU of machine B can be idle (note that the CPU power of machine B is larger than that of machine A).

4.5 Throughput of Event-Batching Notification

The relationship between batch size (the number of events in a sequence) and throughput is shown in Figure 6 (b). Both the number of suppliers and the number of consumers are one. No hook objects are added. After rising from size = 1 to size = 20, the throughput becomes saturated around 1200-1400 events per second.

(c) 1999 IFIP

The throughput of event-batching notification is shown in Figure 6 (c). The batch size is 20. No hook objects are added. The throughput of more than 1200 events per second is obtained when the number of consumers is one. The throughput is almost independent of the number of suppliers while the increase of consumers reduces the throughput. We also evaluated the throughput in the case that the batch size is 100, and it was nearly the same with the throughput in the case that the batch size is 20.

The dashed line represents the prediction based on equation (6) in the ideal condition $\tau_w = 0$. Here, we let $t_{ch} = 0, t_{sh} = 0$ (no hook objects), $m = 20$, $t_3 = 0$ (ignored). It is difficult to evaluate the exact values of t_1 and t_2. As we did in the basic notification, we estimate these values by dividing the data transfer times mentioned in Section 4.3 according to the ratio of the two machines' CPU powers. For example, the time to transfer batched events from machine A (120 MHz CPU) to machine B (266 MHz CPU) was 18.7 msec, and t_1 is estimated to be $18.7 \times 120/(266+120) = 5.81$ msec. Likewise, t_2 is estimated to be 5.32 msec. The experimental throughput exceeds the prediction in some points. This may be caused by the rough estimates of t_1 and t_2.

We also measured the memory consumption. It rises slowly when the number of suppliers is small, but soars to 5 MB when the number of suppliers is large. We also measured the memory consumption in case that the batch size is 100, and found it soars to nearly 16 MB.

4.6 Throughput with Hook Operations

We have evaluated the relationship between the processing time of hook operations and throughput, which is shown in Figure 6 (d). Both the number of suppliers and that of consumers are one. The batch size is 100. The hook object is attached to a CHook. The throughput is inversely proportional to the hook processing time.

The relationship between throughput and the processing time of hook operations is quite clear, and it would be easy to predict the throughput of the framework with particular hook objects added to hooks.

In this experiment, we observed the throughput in the case of no CHook objects was 1502 events per second. Thus, the throughput of this case is the following.

$$\frac{p}{\tau} = \frac{1}{\frac{1}{1502} + t_{ch}} = \frac{1}{0.00067 + t_{ch}} \quad (7)$$

The dashed line represents the prediction based on equation (7), and it is very near to the experimental result.

4.7 Endurance Test

We have conducted an endurance test, in which the event channel was running for 72 hours. Both the number of suppliers and that of consumers is one. The supplier continued to emit events batched in a sequence whose size is 100. Throughputs and memory consumptions were sampled every one minute, and the maximum throughput and minimum throughput in every two hours were recorded, so was the maximum memory consumption. The throughput is

(c) 1999 IFIP

stable around 1500 events per second. Also, memory consumption is stable between 1 MB and 2 MB. The consumer checked if all the events have arrived using serial numbers attached to events, and there were no missing events.

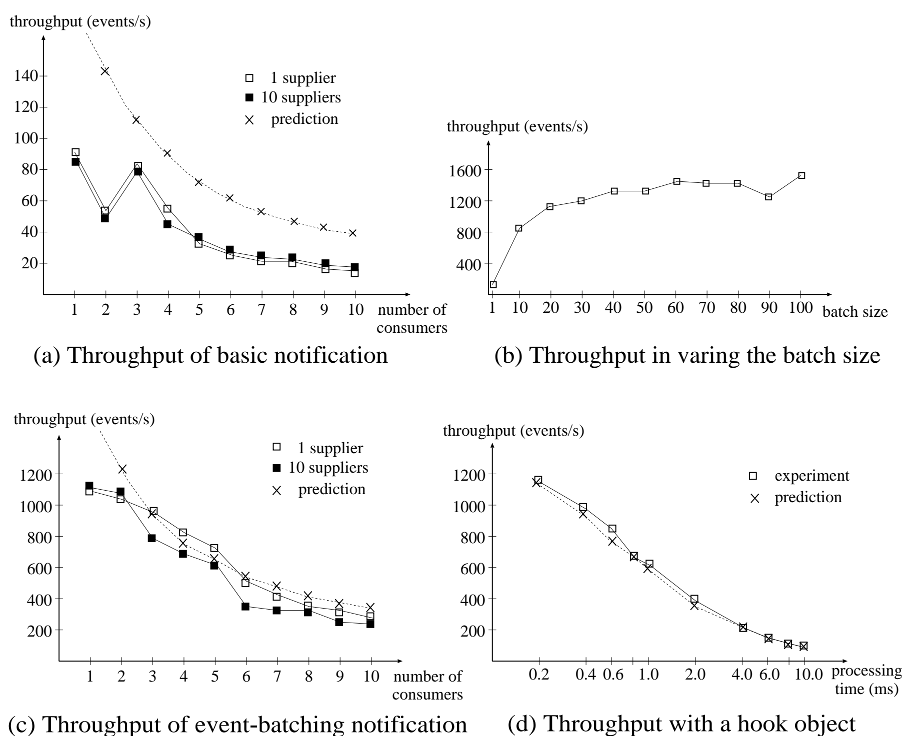

Figure 6: Experimental Results

5. Discussions

5.1 Productivity

The proposed event notification framework brings high productivity in developing NMS components since component developers have only to implement hook objects according to the requirements of each component. This enhances not only productivity but also reliability of the components since the framework encapsulates the skeleton of event notification mechanism. The skeleton provides the common capabilities including receiving, queuing, and disseminating events, which are difficult to implement because they need high performance and robustness in multi-thread programming.

Java brings portability, which allows the framework to run on various operating systems as long as a Java virtual machine is available on them. We have confirmed that the framework was able to run on Windows 95, Windows NT 4.0, and Solaris. This portability reduces the development cost of NMS components significantly.

(c) 1999 IFIP

The framework is based on the so-called white-box approach in the sense that an NMS developer can customize the framework at source code level. Compared with the so-called black-box approach, where an NMS developer utilizes an event channel complied with the CORBA Event Service or Notification Service as an individual black-box component, this makes the framework more customizable. The framework provides a general interface to add new event handling capabilities while the black-box component has only the interfaces that the CORBA Notification Service gives to specify the fixed capabilities including event filtering, QoS, and so on.

5.2 Performance

The performance experiments show that the framework has a high throughput by using event-batching notification. A throughput of more than 1200 events per second was achieved on standard-class computers. The event data used in the experiments, however, was rather simple as mentioned in Section 4.2, and the throughput of sending more complex data could be lower because of CORBA's marshaling and unmarshaling cost.

Giving the right values to event-batching parameters mentioned in Section 3.3 is crucial. Maximum batch size is especially important because it affects both throughput and memory consumption. In the experiments above, a larger batch size brought better performance, but a smaller size brought smaller memory consumption. Therefore, there is the optimal value between the two extremes. As shown in Figure 6 (b), the transfer rate nearly saturates where the sequence size is 20 or more. This saturation point can be regarded as the optimal value.

Pacing interval is also important. If it is too large, event delivery can be delayed. A precise analysis has yet to be done, but the following consideration would be useful. Denote the maximum batch size as M and the time to receive one event as T. The pacing interval I should be larger than $T \cdot M$, but it is not needed to be very large. Suppose $I = T \cdot M + d$. Here, d is the time for the event channel to batch M events received in I and to send them to a consumer. If the event channel receives events continuously, it can batch them up to M and send the batched events to a consumer within I. In this case, the maximum throughput is obtained. If the event channel receives m events in I ($m < M$), the time to batch them and to send them to a consumer is shorter than d, since m is smaller than M. In this case, the event channel can send as many events as it receives. However, throughput depends on various factors including the number of consumers and network traffic, and more precise analysis would be necessary.

The memory consumption is around 1 MB when the number of suppliers and consumers are small, but it soars as the number of suppliers increase. The excessive increase of memory consumption decreased the throughput. We also observed that the memory consumption exceeded the total memory which the queues were expected to use. The cause of this phenomenon has yet to be examined.

6. Conclusions

The paper has presented the architecture of an event notification framework based on Java and CORBA. The framework is customizable for various purposes by attaching event handling capabilities to hooks, and is suitable in developing NMS components which needs various event handling capabilities.

(c) 1999 IFIP

Dynamic installation of hook objects using Java programs enhances the flexibility of the framework. Event-batching notification significantly improves the performance of the framework, which enables the framework to be used for large-scale distributed NMSs. A future work is to exploit a guideline to determine the optimal values of batching parameters and queue size, the load balance between hook operation and event transfer, and so on, since these values can vary with machine power, network traffic, and the topology of event channel connection.

Acknowledgements

The author would like to thank Shin Nakajima, Hiroyuki Hayashi, Yoshiko Ito of NEC Corporation and Wang Li of NEC Informatec Systems for helpful discussions.

References

[1] Fayad, M. E., Schmidt, D. C., Object-oriented Application Framework, *Communications of the ACM*, vol.40, No.10, pp.32-38, Oct. 1997.

[2] Goldszmidt, G., Yemini, Y., Distributed Management by Delegation, Proc. of 15th International Conference on Distributed ComputingSystems, Jun. 1995.

[3] Goto, T., Tohjo, H., Yoda, I., GDMO and Behaviour Program Transmission in TMN Agent, *Proc. of DSOM'97*, pp.156-166, Oct. 1997.

[4] Johnson, R. E., Foote, B. Designing Reusable Classes, *Journal of Object-Oriented Programming*, vol.1, No.2, pp.22-35, Jun. 1988.

[5] Johnson, R. E., Frameworks = Components + Patterns, *Communications of the ACM*, vol.40, No.10, pp.39-42, Oct. 1997.

[6] Keller, A., Service-based Systems Management: Using CORBA as a Middleware for Intelligent Agents, *Proc. of DSOM'96*, 1996.

[7] Kinane, B., Distributed Public Network Management Systems Using CORBA, *Proc. of ICODP'95*, Feb. 1995.

[8] Object Management Group, CORBAservices: Common Object Services Specification, Revised Edition, 95-3-31 ed., Mar. 1995.

[9] Object Management Group, Notification Service Joint Revised Submission, OMG TC Document telcom/98-01-01, Jan. 1998.

[10] Pavlou, G., From Protocol-based to Distributed Object-based Management Architectures, *Proc. of DSOM'97*, pp.25-40, Oct. 1997.

[11] Mansouri-Samani, M., Sloman, M., A Configurable Event Service for Distributed Systems, *Proc. of the third International Conference on Configurable Distributed Systems*, pp.210-217, May. 1996.

[12] Suzuki, M., Kiriha, Y., Nakai, S., Dynamic Script Binding for Delegation Agent, *Proc. of DSOM'96*, 1996.

[13] Tomono, M, Yamanaka, A, Tonouchi, T., Nakajima, S., An Implementation of Customizable Services with Java/ORB Integration, *Proc. of GLOBCOM'97*, pp.1719-1723, Nov. 1997.

[14] Yamanaka, A., Nakajima, S., Tomono, M., Tonouchi, T., A HORB-based Network Management System, *Proc. of ICODP/ICDP'97*, pp.99-109, May 1997.

[15] Yemini, S. A., Kliger, S., Mozes, E., Yemini, Y., and Ohsie, D., High Speed and Robust Event Correlation, *IEEE Communication Magazine*, pp.82-90, May. 1996.

(c) 1999 IFIP