Improving Scalability of Event-Driven Distributed Objects Architectures

Daniel Mencnarowski, Krzysztof Zieliński
{mencnar, kz}@ics.agh.edu.pl
Dept. of Computer Science, University of Mining & Metallurgy
Al. Mickiewicza 30, Cracow, Poland
tel:+48 (12) 617 39 82, fax:+48 (12) 617 39 66

July 26, 1999

1 Introduction

Distributed software architecture promote development focused on modular building blocks and their interconnection. Implementing software connectors [6] thus becomes a key aspect of the system scalability and performance. Software interconnection and middleware technologies such as CORBA [19, 11], RMI [12], Jini [20], ILU [7] and ActiveX [5] provide valuable services for inter-component communication. Practical understanding of their mechanisms applicability, such as distributed events and asynchronous notification, is still often minimal. The concept of these mechanisms is built around well known publisher/subscriber pattern [4, 8], called also observed/observer paradigm [16].

The traditional client server model has been deployed intensively in almost all computing industries. The simple request/reply interaction maps well to a large class of applications where one entity provides a service to a set of local or remote entities. Despite its success and wide acceptance, the request/reply model, in comparison publish/subscribe, falls short in all three areas [21]: scalability, configurability and high-availability. It is seen especially well in event-driven computing typical of such systems like location and monitoring systems, modern customer-centric enterprise software, manufacturing plants management etc. Despite of many superior features of publisher/subscriber pattern its efficient implementation and usage is still a very challenging issue.

The goal of this paper is to present practical techniques for improving scalability of distributed objects architecture built around event-driven paradigm and evaluation of their effectiveness in case of location systems. As a case study the ABng location system [15, 9] constructed using publisher/subscribernos pattern over CORBA compliant middleware is investigated in this paper. The system is rather complex and consists of several repositories that may contain hundreds of observable entities that are CORBA objects, hence the event notification techniques efficiency is sole for its scalability and performance.

The main contribution of this paper is elaboration and evaluation of three main concepts that play a crucial role in improving scalability of event-driven architectures: (i) events dissemination quality of service (QoS), (ii) notification tree based events-routing, and (iii) transparent multicast event-channel notification service. They are accompanied by the technique based on selective events tracing built around the view abstraction [15]. These techniques applicability and efficiency have been practically verified by the ABng system implementation and performance testing.

The structure of the paper is as follows. In Section 2 the event-driven distributed object architectures features are elaborated in more details. Section 3 presents the ABng system architecture. Next in Section 4 techniques of improving scalability of ABng system are specified. Section 5 is dedicated to transparent multicast based event-channel service implementation. The results of the performance study are presented in section 6. The paper ends with a conclusion.

2 Event-Driven distributed objects architectures

The event-driven highly scalable distributed object oriented system architecture are based on three basic technologies [21]:

Publisher/subscriber communication paradigm. This a design pattern in which one dedicated component takes the role of the publisher (observed). All components dependent on changes in the publisher are its subscribers (observers). The publisher maintains a registry of currently-subscribed components. Whenever a component wants to become a subscriber, it uses the subscribe interface offered by the publisher. Analogously it can unsubscribe. When the publisher changes state, it sends notification to all its subscribers. The subscribers in turn retrieve the changed data at their discretion.

Subject based addressing. This is a technique of addressing information instead of the information provider. This simple concept gives incredible flexibility and opens opportunities to build large-scale information dissemination system in a broad range of industries.

Reliable multicast protocol. The most existing middleware systems is based on TCP/IP point-to-point communication, so when the message has to be communicated to the group of subscribers spread around the computer network it must be replicated. The most suitable solution of this problem is based on UDP/IP multicast, but it requires adding reliability lacking in UDP based communication.

All these three concept complement each over providing the complete highly scalable solution. Unfortunately a such solution rather complex and difficult in implementation.

The most middleware exploits the potential of publisher/subscriber pattern only, which offers the following freedom in its implementation:

- The publisher can decide which internal state changes it will notify its observers about. It may also queue several changes before calling `notify()`.
- subscription and ensuing notification can be differentiated according to event type. This allows subscribers to get messages only about events in which they are interested.
- The publisher can send selected details of the data change when it notifies its subscribers, or can just send a notification and give the subscribers the responsibility to find out what changed.

In more general terms we differentiate between the push and the pull model. In the push model, the publisher sends all changed data when it notifies the subscribers. The subscriber have no choice about if and when they want to retrieve the data - they just get it. In the pull model the publisher only sends minimal information when sending a change notification - the subscriber are responsible for retrieving the data they need. Many variants are possible in the middle ground between these two extremes.

The basic publisher/subscriber pattern may be enhanced in many way. The Event Channel concept was proposed by OMG in its Event Service Specification [17]. This pattern strongly decouples publisher and subscribers. This service is further elaborated by Notification Services Specification [18]. Another variant is proposed by the Jini Distributed Event Specification [20] which introduces agents that may be thought of as distributed events adapters. These concepts will be further elaborated in context of ABng system implementation in Section 4.

3 Abng system

The system, called *Active Badge*, was originally invented and developed at Olivetti Research Laboratory, Cambridge, UK [10] in 1990-92. It uses a hardware infrastructure whose key components are infra-red sensors, installed in fixed positions within a building, and infra-red emitters (*active badges*) that are worn by people or attached to equipment. Sensors are connected by a wired network which provides a communication path to the controlling device, called a poller, and distributes low-voltage power. An active badge periodically transmits an infra-red message containing a globally unique code (a badge identifier) using the defined data link layer protocol [3]. Messages are received and queued by sensors.

The information about the current location of users and equipment is provided to various applications, such as presentation tools which display location data or applications which use location data to control the users' environments. The software for the original Active Badge system developed at ORL used the ANSAWare [1] distributed environment.

The ABng project focussed on the redevelopment of the Active Badge system software using modern component and object-oriented technologies. The system was developed using CORBA-compliant environments: Orbix [13] or, alternatively, OmniORB [2] and OrbixWeb [14]. It is based on an object model in which all logical and physical elements of the Active Badge system (users, locations, sensors, badges, etc.) are represented as CORBA objects.

The system has a layered architecture which hides details of gathering location data. This makes it possible to replace a localization method based on infra-red sensors and emitters by another one. In ABng, location data are represented using abstract notions of *location* and *locatable* objects rather than in terms of sensors and badges. A location is a part of an environment obtained by partitioning the space according to an arbitrary, user-defined rule. Typically, an office space can be divided into buildings, floors, rooms, etc. A locatable is an object which can be observed by the system and whose location changes within the monitored environment. A locatable can be a person or a piece of equipment, such as a computer, a printer or a book.

The basic ABng concept is a *view*, which is a collection of location and locatable objects, i.e. it represents a part of the environment space and a subset of objects that can be localized within this part. The localization precision of a view's locatables is determined by the size of locations belonging to the view. Within a system, a number of view objects can exist, each of which can hold information concerning the current locations of users or equipment belonging to different groups and provided at different levels of abstraction with different precision. The ABng system exploits the following fundanental concept:

- General templates for each entity, as well as for a repository, were designed,
- These templates support the implementation of each entity eliminating the overhead related to representing each entity as a separate CORBA object,
- A light-weight notification mechanism for repository clients was designed.

In the ABng, three types of entities, and as a result, three types of entity interfaces have been derived, with:

- **static attributes**: Most ABng entities have a fixed and relatively small number of attributes. Such entities are accessed via interfaces, in which each entity attribute is represented by a corresponding IDL attribute.
- **dynamic attributes**: Another approach is to treat an entity as a collection of attributes of arbitrary types and to provide an access to them via an appropriate interface. Such an interface offers a pair of accessor operations to set and retrieve the value of a single attribute in which an attribute is referred to by its name and value is encoded using the IDL *any* type. The interface permits the retrieval of all attributes as a list.
- **hybrid attributes**: This type of an interface explicitly defines those attributes which are common for all objects representing entities. Additionally, it uses the *Dynamic Attribute* paradigm to provide access to object-specific attributes.

The location and locatable entities possess hybrid attributes. Every entity inherits from the *Entity_Observed* interface which enables other objects to register their interest in changes to an entity's state. When the state of an entity changes, any registered parties are informed about it.

Objects built according to any of the three templates are stored in repositories which also have a generic interface with the following operations:

- *add* – creates and adds a new entity to the repository. The initial state of the created object is determined by the contents of the *Description* structure passed as an argument. This method returns the object reference of the new entity.

- *update* – replace the state of the entity denoted by a reference with values stored in the *Description* structure.
- *remove* – remove an entity denoted by a reference.
- *find* – returns a collection of references of those entities which match the criteria specified in the *Pattern* structure passed to the operation.
- *find_values* – returns a collection of the states of the entities matching the given criteria. In other words, this operation returns the matching object by value rather than by reference.
- *complex_query* – works like the *find* method but additionally registers the observer (*obs*) for all returned entities in a repository. It also possesses an *out* parameter by which the sequence of entity descriptions is returned. This operation practicaly implements subject-based addressing described in previous section.

The entity repository interface inherits from the *Entity_Rep_Observed* interface, enabling registration of interest in the arbitrary collection of entities. This facilitates registration of an observer for a large number of entities.

The repository template is used for implementation of 14 different repositories of entities communication with each other via events which together consists ABng system architecture. Location aware applications are getting access to relevant information by registration with the view.

4 Techniques of scalability improvement

The complex event-driven distributed system implementation requires particular attention to be put on the techniques of the event notification mechanisms scalability. In this section the techniques exploited during the ABng system implementation will be very shortly specified.

4.1 Quality of Service

The Quality of Services (Qos) concept draw a particular attention in Notification Service [18] and Jini specifications [20] and is one of the most fundamental that introduces flexibility and scalability to event-driven systems. The QoS policy is implemented usually by events adapters with functionality defined by the user. The examples of such adapters are:

Store-and- forward agent allowing the event generator to send the notification to one entity the store-and-forward agent that will forward the notification to all the event listeners, perhaps with a particular policy.

Notification filter is an object that may be local to either the event generator or the event listener. This agent gets the notification, and spawns a tread that will respond to the notification, using a method supplied by the object which originally interested in events of that kind.

Notification mailbox stores notifications for another object until the that object requests that the notification be delivered. This design allows the listener object that registered interst in the event type to select the times at which a notification can be delivered, without losing any notifications that would have otherwise have been delivered.

With the ABng project the store-and-forward agent modification has been used to construct light-weight notification mechanisms built into the repository template. It is called the notification tree and described in next section. The notification filters have been used per view repository to concentrate notifications in a way to help minimize network traffic. For each view is possible to define the queuing period. The filter queues several sightings by the queuing period - burst delay time, before calling the notify function which passes the collected events as a one network message.

4.2 Notification tree based events routing

A typical ABng application uses a significant amount of information encapsulated by a number of ABng objects. To obtain this information an application interacts with the appropriate ABng objects. To optimize the performance of these interactions, ABng implements a caching algorithm – i.e. a Smart Proxy Layer (SPL), based on the smart proxy mechanism available in Orbix and OrbixWeb. When the application obtains an entity reference (for instance by executing the repository's *find* method), the smart proxy of the entity object is instantiated within the application's address space and the entity's attribute values are cached. When the application enquires about an attribute value, this value is retrieved from the cache and no remote call is performed.

If a value of an entity attribute changes, all proxies active in different applications have to be notified about this change. For this purpose, a notification mechanism based on the Observer [8] architectural design pattern (also known as Publisher/Subscriber) has been designed. Every proxy registers itself as an observer of the entity it represents. Each time an attribute of the entity is updated, the proxy is notified and marks its cache as invalid. The next query by the application for an attribute value causes the proxy to contact the entity and retrieve the whole description. Registering smart proxies directly within an entity object would be very inefficient, since for every smart proxy registration a corresponding proxy object in the server containing the entity would be created. To solve this problem, a notification dispatching mechanism is employed. This is depicted in Figure 1 and explained below.

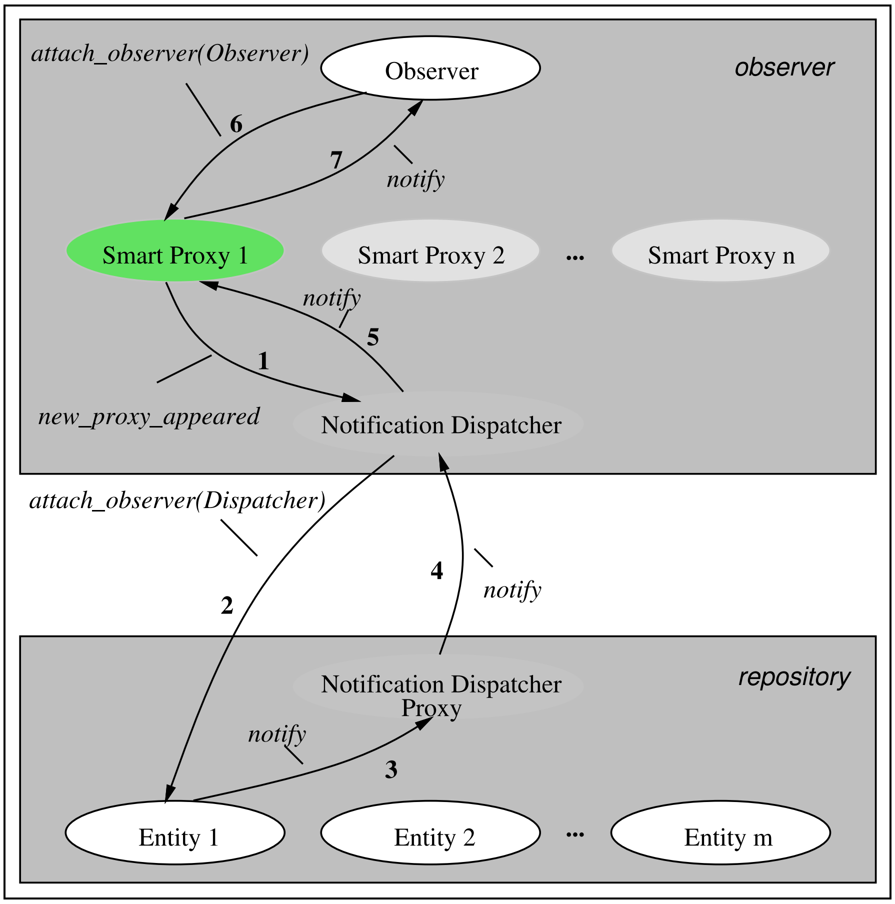

Figure 1: ABng notification mechanism

When a proxy representing the first entity from a given repository is instantiated within an application, an object, called the *notification dispatcher* associated with the repository, is created. The dispatcher will represent all proxies associated with entities contained in the given repository and dispatch notification mes-

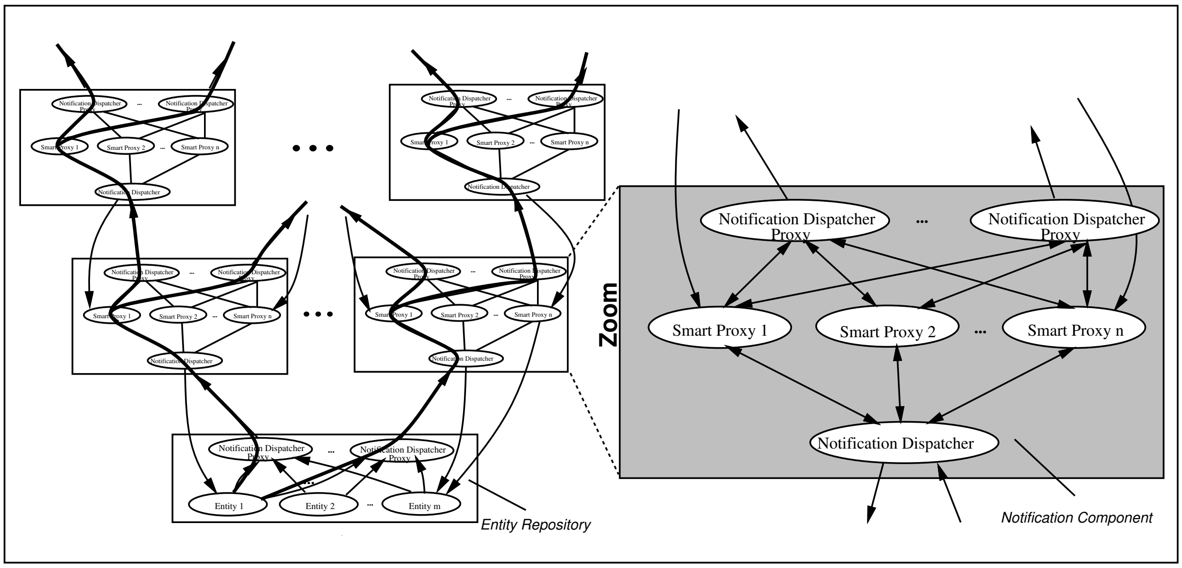

Figure 2: Notification Tree

sages to the proxies. The smart proxy does not directly subscribe itself to the corresponding entity. Instead, it calls the notification dispatcher (arrow 1 in Figure 1). The dispatcher casts the smart proxy reference to the reference of an ordinary proxy and calls the real stub of the registration method (*attach_observer*) passing its own reference as an argument (the dispatcher has to provide the observer interface). This call results in a real remote invocation on the entity (arrow 2). If the dispatcher contacts the entity server for the first time, the dispatcher proxy is instantiated within the server at the same time. Within the repository the number of existing dispatcher proxies is equal to the number of applications which contain proxies observing the repository's entities. The entity stores a dispatcher reference (a pointer to a dispatcher proxy) in a registry of its observers.

When the state of an entity changes, the entity notifies all notification dispatchers via their proxies (arrow 3 and 4). On the application's side, the dispatcher obtains a reference of the updated entity, which, in fact, points to the local smart proxy. The dispatcher forwards notification to the smart proxy (arrow 5). Finally, the proxy marks its cache as invalid.

In addition to smart proxies maintaining caches, a user's application can also contain ordinary objects which are interested in asynchronous notification regarding entity updates. This notification is also performed using the described mechanism. An application object that wants to be notified about changes to an entity's state calls the *attach_observer* operation of the entity of interest (arrow 6). This call, however, is not transmitted to the entity. It only affects a local registry of entity observers, which is maintained by the smart proxy. After the proxy is notified about entity update, it forwards this notification message to all entity observers contained within the application (arrow 7).

It should be noted that the above mechanism is completely transparent to the application. An application which uses this mechanism must be linked with a library containing smart proxies and dispatchers.

The proposed solution for notification has the following structural features:

- The notification dispatcher is a CORBA object which represents a collection of smart proxies.
- The smart proxies are not CORBA objects and may be notified using local method invocations.
- The collection of repository entity smart proxies in the client space is represented in the repository by a single notification dispatcher proxy. It saves a significant amount of memory and makes the space occupied by the repository server independent of the number of entities in the collection and the global number of existing proxies.

- A client may not be aware that a notification dispatcher is used. Its activity is completely transparent to the client even from the programming point of view.
- The number of notification dispatcher proxies in a repository server is dependent only on the number of clients in the system which contain entity smart proxies for entities in that server.
- The proposed notification mechanism is selective, which means that only notification dispatchers that have registered smart proxies corresponding to the changed entity in the repository are notified.

It is necessary to point out that the existence of a notification dispatcher does not influence the scalability of the system in terms of the number of observer clients.

This last issue can be addressed using replication. A client component with a collection of smart proxies and a notification dispatcher could be generalized as a notification component shown in Figure 2. Scalability with respect to the number of clients can then be achieved by organizing the system into a notification tree built from notification components. Each smart proxy has registered several notification dispatchers from the next layer up in the tree. In the root node the repository of entities exists. In other nodes only smart proxies are present. The proposed architecture represents a distributed collection of entities which could be highly available for a large number of clients despite geographic distribution.

The propagation of a repository entity update is marked in Figure 2. It is easy to see that the proposed solution has similar scalability to notification based on multicast over IP communication protocols that propagate messages down a multicast tree. The advantage is that the notification tree does not require any multicast protocol support.

4.3 Event service

When passing data among processes and computers there are possible two approaches; each of the servers producing data sends it directly to all clients interested in the topics, another approach uses an additional server that cooperates with both parties; producing events and accepting them.

It acts as a consumer of events for servers producing them, on the other hand it acts as a producer for all the consumers, thus separating these two categories of parties. Moreover it decouples the communication between these groups, therefore allowing seamless configuration changes on both sides without the need to reconfigure or inform the other side.

The *OMG* in its effort to provide a complete set of system services (Common Object Service Specification) introduced Event Service specification defining various styles of gathering and distributing events.

The Event Service specification introduces a notion of an event channel – an entity that intermediates among parties gathering events from producers, then sending them to all consumers. This way the communication between both parties is decoupled, thus avoiding unnecessary dependencies, the clients need only to know how to get the reference of a given event channel, they do not need to know identities of other parties participating in the information exchange.

There exist two types of event channels:

untyped event channels – providing means for handling *CORBA Any* types.

typed event channels – specialized for forwarding events represented as calls to an object implementing an interface that passes data only in one direction, the implementation of this kind of a channel uses *DSI*, *DII* and *Interface Repository* to achieve its goals. Each instance of an event channel may implement another specialized interface.

There are two possible types of event communication:

push style ; The producer plays an active role sending events to the event channel, while the consumer awaits for events in a passive way,

pull style ; The consumer plays an active role querying the event channel for new events, while the producer is waiting for queries about the events.

4.4 Notification service

The CORBA Notification Service is a service that extends Event Service in many ways:

- providing events with well–defined data fields for structured events thus enabling fine grained event filtering
- configuring various quality of service parameters at various levels: channnel, proxy or a single event
- event suppliers may be made aware of what event types the consumers are expecting, thus avoiding sending unnecessary events that will not be accepted by any consumer
- when a new event type is made available by suppliers, the consumers may discover the new offer and subscribe
- standarize the event channel factory interface

The goal of Notification Service was to satisfy requirements: scalability, performance and quality of service.

4.5 Domain specific event filtration

The Notification Service filter interfaces are a poweful tool for limiting event traffic, but often it may be desirable to provide a custom filter with ability to retain events, perform some analysis and forward only important ones.

Such a custom event filter can be set up with two base configurations:

- as a specialized implementation of Event or Notification Service or any other event gathering and broadcasting specification, or
- using one of the mathods suggested by Event Service specification to provide custom event filtering; create two event (or notification) channels or with primary purpose for gathering events from all suppliers and sending them to the filter object and another one supplied with events by filter object delivering the events to all consumers

The latter method may prove to both elegant (no custom build services) and efficient, especially if the Event (or Notification) Service is supplied as a liked library and a channel may be instatiated in the same process as the filter object, this way removing unnecessary inter–process invocations.

5 Transparent multicast EC implementation overview

The unicast based systems face an inevitable bottleneck when addressing many clients. They are forced to send the same content of data to all parties. This approach causes excessive CPU load of the sending machine, and moreover network load that cannot be easily cured.

The *CORBA* and it's generic protocol – *GIOP*, that all other communication protocols are about to be based upon, is inherently connection oriented and based on point–to–point transport. As it will be shown later this approach leads to inefficiency and bottlenecks.

Amid the fact that *OMG* members and standarization bodies are aware of the problem and in fact there was created a new specification – the Open Communication Interface that is supposed to allow new communication protocols to be used, including multicast ones, it has weak industry support so far. Moreover even with the *OCI* interface supported the use of a new unstandarized protocol causes all parties involved in the communication to use the same protocol and this leads to a lot of interoperability problems.

Unless there will be standarized multicast protocol endorsed by *OMG*, as there is *TCP/IP* and *IIOP* for unicast communication, all multicast based extensions will be vendor specific.

Because of the above issues it only makes sense to use multicast protocols for specialized services, just like CORBA Event Service or Notification Service. Yet there should be made any effort to make the clients of these services independent of the internal implementation startegy and protocols used.

The main idea for keeping the described Event Service implementation compliant with the specification is usage of the layer of event channel daemons, which hides the internal usage of multicast protocol from IIOP based producers and consumers, and intermediates between the process of event channel and consumers, receiving multicast data stream using a propriety protocol and sending it to a set of local consumers.

These issues can be solved in a number of ways:

- changing IIOP protocol into a new one supporting multicast communication, using a vendor depended API or a standard *OCI* interface. Yet, this solution limits the usability of such a system to specific set of vendors or even a single one. Moreover it requires modifications in clients code or at least rebuilding them. Until a multicast protocol is adopted by CORBA standardization bodies, this solution has too many limitations to be considered seriously,
- building event channel factory and/or event channel daemons as library files, so an implementation could be linked with a client's executable. This approach would limit number of hops that the message needs to go through. This would result in better performance, but the libraries required to send/receive events would be specialized for a particular programming language, thus limiting CORBA heterogeneous features,
- optimizing pull consumers; the multicast communication is used only for push consumers, however introducing a few additional changes in the system architecture one could use multicast communication also for pull consumers. However, this feature was not taken into account in the design phase, due to the fact, that pull consumers are used rarely as compared to push consumers. Provided that this feature will become essential for the smooth work of a system, it can be easily added without much harm for the current design and implementation.

Introducing event channel daemons amid its undisputed advantages has also some drawbacks; the administrative cost of keeping the daemons running, an additional delay of events on its way from a producer to consumers.

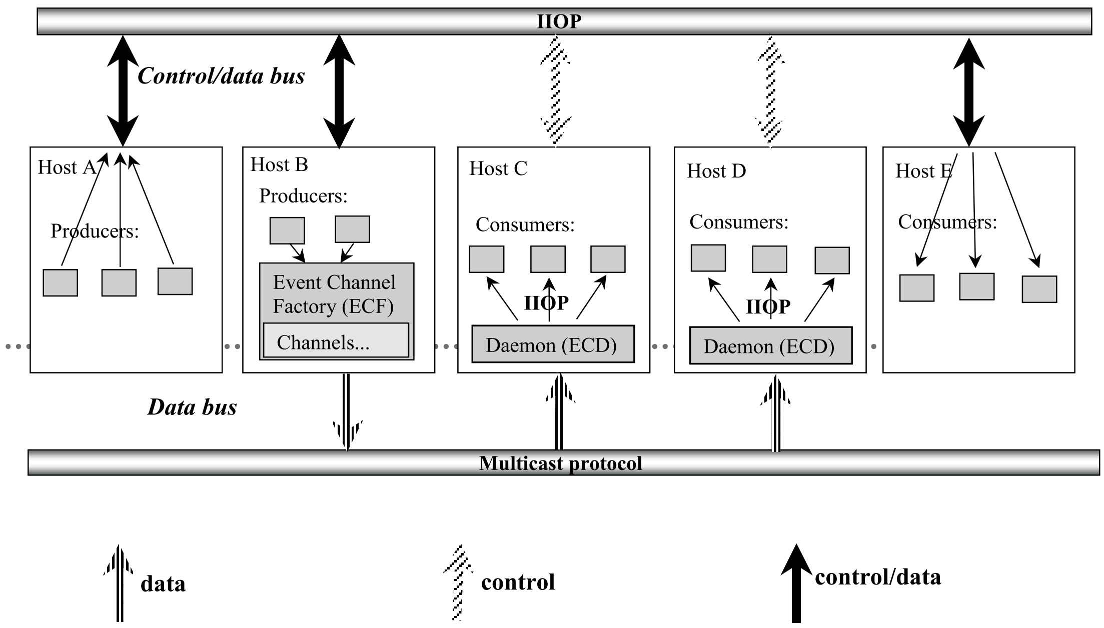

Figure 3: Event Service implementation architecture

The central part of the referred system is an event channel factory (ECF) with a proprietary interface (the standard does not define one) to create and manage event channels. It should be located on a machine, which is a source of most of events, just to reduce network traffic caused by calls of remote producers. Event channels are created inside the address space of the factory, yet some functional parts of them (daemons) are located on other machines.

On each machine with push consumers an event channel daemon (ECD) should be activated, although the daemon may forward data to a set of computers in a local area network. All daemons are registered in the factory, so it can have a complete knowledge of ECDs locations.

Each event after being accepted by an event channel is transmitted using a multicast protocol (marshalled using the CDR (Common Data Representation)) to a set of event channel daemons that have active consumers for the respective event channel. Only push consumers can benefit from the advantages of the multicast protocol. Pull clients use direct connection-oriented IIOP calls to retrieve events from event channels. When a push consumer connects to one of the event channels, the factory checks whether an event channel daemon is present on the consumer's host, where the consumer is located, using information from the supplied object reference. If there is a daemon, it is instructed to forward all events in a specific event channel to the interested consumer using standard CORBA invocations. This approach allows integration of the event channel implementation with any of CORBA compliant clients (both consumer and producers). If the channel is unable to find an appropriate daemon, the event channel will push events to the consumer directly using IIOP. In this way, in the absence of event daemons it behaves just like a normal event channel implementation.

6 Performance evaluation

There were two areas the event delivery architectures improvements were tested for performance results:

- using pacing interval quality of service parameter to achieve a tradeoff between resource consumption (both CPU and network load) and timely event delivery
- introducing alternative communication protocols for events' transmission

6.1 Pacing interval

The pacing interval is one of the QoS features in Notification Service specification. It defines maximum period of time the event source or event channel will collect and buffer events to be sent with one invocation as a sequence of events to a consumer.

The consumer must be prapared to receive sequences of events instead of individual events. This mechanism may prove beneficial for resource consumption parameters especially when the events are generated at a great pace and events' delivery may be delayed for some time.

For the purpose of performance tests this parameter was used in *ABng* system and was called "burst delay".

Plase note that each used of the ABng system emits an event every 10 seconds, and observes all other users' generated events.

As one may observe in Figure 4 the system is capable of serving quite a large number of users while consuming reasonable amount of resources even for "immediate forwarding" mode, however the consumption might be reduced if the view server is configured to delay all sightings before forwarding them to observers. The difference in system performance is especially significant for greater number of observers and users.

Using deferred forwarding a single message encloses a sequence of sightings instead of just one for immediate forwarding, the longer the sequence is the better performance could be achieved as compared to the case where all items are sent separately.

The performance penalty caused by immediate forwarding mode is made up of two parts; the one inclined by the amount of data that needs to be transmitted and a constant costs provoked by assembling an IIOP request.

Each time a new sighting is provided by producers to the view server, it is stored in a buffer. Periodically a separate thread examines contents of the buffer and sends its contents to all recipients, however the thread wakes up only after a configurable amount of time after receiving first not forwarded piece of information, so there is a guarantee that the maximum delay of sighting in a group is equal to the previously configured burst delay time.

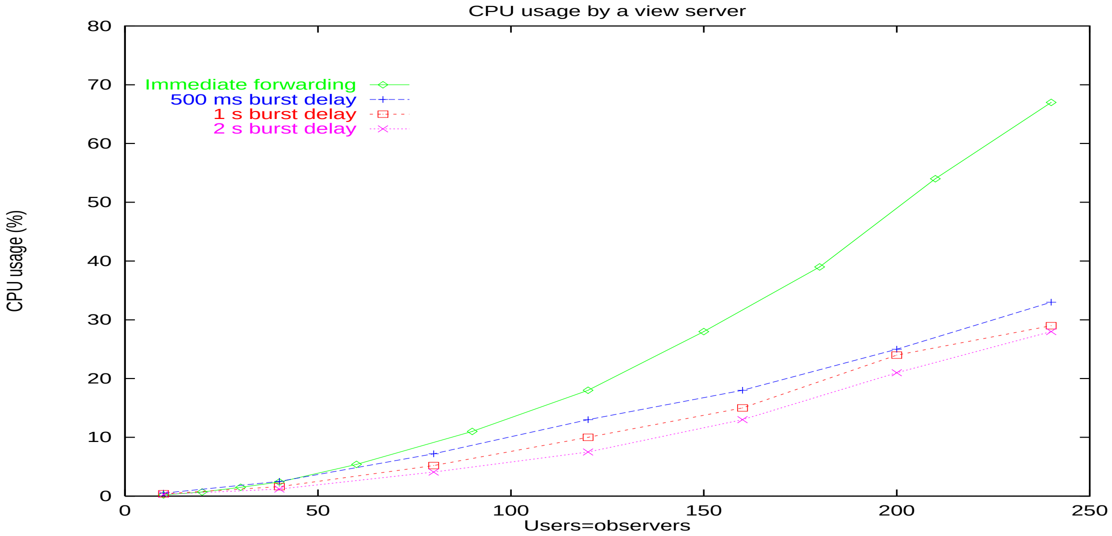

Figure 4: CPU usage by a view server performing notification with different burst delays

It is worth to give attention to the fact that buffering mechanism also causes some delays and below a certain number of users and observers the immediate forwarding mechanisms may prove to be more efficient than the deferred approach. It is possible to write an adaptive algorithm that switches between deferred and immediate mode, depending on the average number of sightings that are required to be forwarded to all observers. Another technique worth consideration is a system where an administrator limits the amount of resources that may be consumed by the system, so the server will be changing burst delay depending on the system load. Then if there are only a few people in the building the system would forward sighting without a delay, on the other hand with the heavy system load the burst delay will be increased to match resource utilization limits.

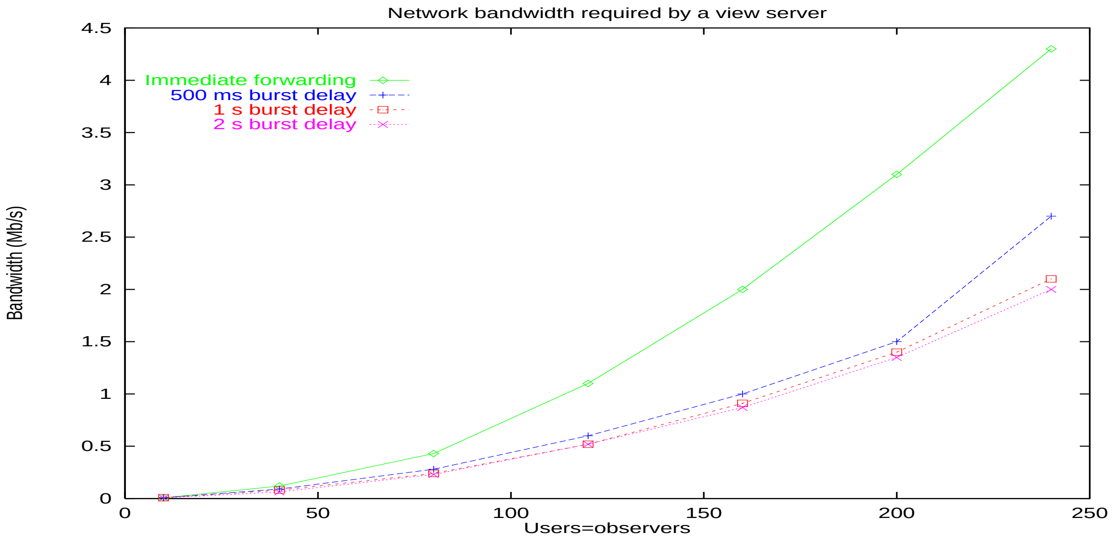

Figure 5: Network bandwidth required by a view server

It is easy to note that the influence of buffering mechanism gains even better results sooner than it does for CPU usage. The reason for the difference is additional overhead of information prepended to messages sent across network by layers, that the CPU is not involved in.

6.2 Multicast event channel

7 Conclusions

References

[1] APM Ltd., Cambridge UK. *ANSAware 4.0 – Application Programmer's Manual*, 1992.

[2] AT&T Research Laboratory. *OmniORB programmers' guide*, 1998.

[3] Frazer Bennett and Andy Harter. Low bandwidth infra-red networks and protocols for mobile communicating devices. Technical Report 93.5, Olivetti Research Laboratory, Cambridge, UK, 1993.

[4] Frank Bushmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *A System of Patterns*. Wiley, 1996.

[5] D.Chappell. *Understanding ActiveX and OLE*. Microsoft Press, Redmond WA, 1996.

[6] Richard N. Taylor Eric M. Dashofy, Nenad Medvidovic. Using off-the-shelf middleware to implement connectors in distributed software architectures. *21st International Conference on Software Engineering*, May Los Angeles, CA USA, 1999.

[7] ftp://ftp.parc.xero.com/pub/ili/ilu/ilu.html. *ILU – Inter-Language Unification*.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[9] Andy Harter and Andy Hopper. A distributed location system for the active office. *IEEE Network. Special Issue on Distributed Systems for Telecommunications*, 8(1), January 1994.

[10] Andy Harter and Andy Hopper. A distributed location system for the active office. *IEEE Network. Special Issue on Distributed Systems for Telecommunications*, 8(1), January 1994.

[11] Mitchi Henning and Steve Vinoski. *Advanced CORBA programming with C++*. Addison Wesley Longman, 1999.

[12] http://java.sun.com/products/jdk/rmi/index.html. *Remote Method Invocation*.

[13] Iona Technologies Ltd. *Orbix 2.3 programming guide*, 1998.

[14] Iona Technologies Ltd. *OrbixWeb 3.0 programmers' guide*, 1998.

[15] Krzysztof Zieliński Jakub Szymaszek, Andrzej Uszok. Building a scalable and efficient component-oriented system using corba – an active badge system case study. *Distributed Ststems Engineering*, (5), 1998.

[16] Thomas J. Mowbray and Raphael C. Malveau. *CORBA Design Patterns*. John Wiley & Sons Inc., 1997.

[17] Object Management Group. *Common Object Services – Event Service*, 1998.

[18] Object Management Group. *CORBA Notification Service*, 1998.

[19] Object Management Group. *CORBA/IIOP Specification, Revision 2.2*, 1998. OMG formal/98-07-03.

[20] Sun Microsystems. *Jini Distributed Events Specification*, 1999.

[21] TIBCO Inc. *TIBCO Object Bus – Whitepaper*, 1998.