Extending Content-based Publish/Subscribe Systems with Multicast Support

Viktor S. Wold Eide[1,2], Frank Eliassen[2], Olav Lysne[2], and Ole-Christoffer Granmo[1,3]

[1]University of Oslo
P.O. Box 1080 Blindern
N-0314 Oslo, Norway
viktore,olegr@ifi.uio.no

[2]Simula Research Laboratory
P.O. Box 134
N-1325 Lysaker, Norway
viktore,frank,olavly@simula.no

[3]Agder University College
Grooseveien 36
N-4876 Grimstad, Norway
ole.granmo@hia.no

Simula Research Laboratory
Technical Report 2003-03

July 2003

Abstract

Event-based interaction is recognized as being well suited for loosely coupled distributed applications. Current distributed content-based event notifications services are often architectured to operate over WANs (wide area networks). Additionally, one to one transport layer communication primitives are used. As a result, these services are not suitable for (parts of) applications having a combination of high event notification rates and locally a large number of interested parties for the same event notifications.

In this paper we describe the architecture of a distributed content-based event notification service, designed to take advantage of the available performance and native multicast support provided by current "off the shelf" network equipment. Our event notification service is designed primarily as a LAN (local area network) service and hence complementary to event notification services for WANs. A prototype has been implemented and experiments indicate that our service provides both scalability and high performance. A client may publish several thousand event notifications, carrying several MBytes of data, per second. The service is unaffected by the number of interested parties, due to the use of native multicast.

1 Introduction

Traditionally, the client/server interaction model has been used extensively for building distributed applications. However, a large class of distributed applications are better structured as a number of asynchronously processing and communicating entities. Such applications fit well to the publish/subscribe interaction paradigm, leading to an event-based interaction model. Event-based interaction provides a number of distinguishing characteristics, such as asynchronous many to many communication, lack of explicit addressing, indirect communication, and loose coupling.

Event-based systems rely on some kind of event notification service, as illustrated in Figure 1. A distributed event notification service is realized by a number of cooperating servers, also denoted *brokers* in the literature. Clients connect to these servers and are either *objects of interest*, *interested parties*, or both. An object of interest publishes event notifications, or just notifications for short. Some systems may allow/require the object of interest to advertise the notifications potentially generated before publishing. Interested parties subscribe in order to express interest in particular notifications. The responsibility of the event notification service is routing and forwarding of notifications from objects of interest to interested parties. In essence, the servers jointly form an overlay network of notification routers. A survey of the publish/subscribe communication paradigm and the relations to other interaction paradigms are described in e.g. [6].

Publish/subscribe systems differ with respect to the expressiveness of their subscription languages. In *channel-based* systems, e.g. as specified by the CORBA Event Service [8], an interested party may subscribe to a channel and in effect receive all notifications sent across that particular channel. *Subject-based* systems, such as e.g. TIBCO Rendezvous[14], provide somewhat finer granularity with respect to selection of notifications. An object of interest determines the most appropriate subject for each notification published. The content of a notification is not used by the service for forwarding. Subject-based systems may also support hierarchical subject names and/or wild-card expressions on subject identifiers to further improve the expressiveness of subscriptions. *Content-based* publish-subscribe systems, such as Elvin[12], Gryphon[9], Hermes[11], and SIENA[2] provide even finer granularity. In such systems notifications typically consist of a number of attribute/value pairs. A subscription may include an arbitrary number of attribute names and filtering criteria on their values. Hence, content-based systems increase subscription selectivity by allowing subscriptions along multiple dimensions.

Distributed content-based publish/subscribe systems, such as Gryphon, Hermes, and SIENA, are often architectured to operate over WANs, e.g. public networks or the Internet. A main concern is how to efficiently distribute event notifications between servers. E.g. in [9], the servers are treated as the communication endpoints.

In contrast, in this report we are mainly concerned about how to efficiently distribute very high rate event notifications between a large number of objects of interest and interested parties within a smaller region, e.g. a LAN or an administrative domain. A scalable and high performance event notification service allows development of new classes of applications which utilize event-based interaction, e.g. high performance parallel computing within clusters of powerful computers and real-time video streaming to clients hosted by heterogeneous computers and network connections. The application domain of real-time content analysis[4] covers both these areas. A service capable of handling the data rates of several concurrent high quality real-time video streams is definitely useful for other application domains as well. Highly important is how to transport the notifications all the way from the objects of interest and to the clients, i.e. not only between the servers.

With respect to the communication path between an object of interest and a server, it is important to ensure that only the relevant notifications are generated and sent, i.e. to support filtering at the source. Elvin relies on a quenching mechanism[12] where (parts of) the subscription database is sent to objects of interest. This strategy is described as relatively complex and is optional in order to support thin clients. With respect to the communication path between a server and the interested parties, efficient multicast is crucial in order to distribute each notification to a large number of interested parties. A notification sent by *native* multicast requires only a single send operation and propagates over each network link only once, regardless of the number of computers hosting interested parties and the number of interested parties hosted by each computer.

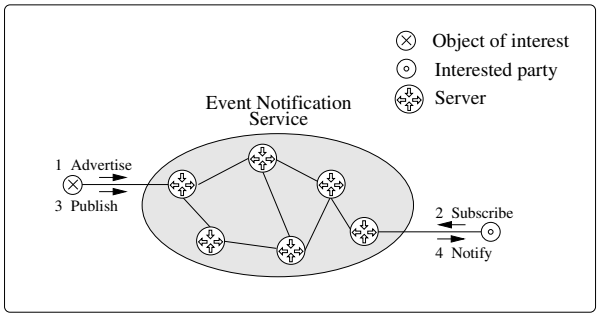

Figure 1: A Distributed Event Notification Service

Utilizing native multicast communication in channel-based and subject-based publish/subscribe systems is relatively straightforward. In such systems, a notification is basically mapped onto a channel or a subject which is then mapped onto a multicast address.

The principles and techniques used for routing and forwarding notifications between servers in distributed content-based publish/subscribe systems are similar to those used by IP routers in order to support IP multicast. In e.g.[1], an efficient multicast protocol for content-based publish/subscribe systems is described. The challenge of utilizing *native* multicast support for content-based publish/subscribe systems is well known[6]. Simulation results for some algorithms for distributing notifications in a network of brokers is presented in [9]. But to our knowledge, native multicast support has not been implemented in current content-based publish/subscribe systems. Hence, it may be desirable to enhance existing content-based publish/subscribe systems to take advantage of network level multicast communication. A major challenge is how to achieve this while affecting neither the API nor the semantics of the event notification service.

Some event notification services adopt a hierarchical approach where the intra domain and the inter domain cases are handled differently, e.g. [14], using specialized routing daemons between domains. A hierarchical approach is particularly useful when notifications have high regionalism, i.e. when notifications have high interest in certain parts of the network and little or no interest in other parts. Non-uniform distribution of subscriptions is likely due to e.g. distributed applications built with locality in mind for performance and cost reasons, location based services, security, what people are interested in, etc.

In this paper we describe the architecture, the implementation, and the measured performance for our distributed content-based event notification service. The service takes advantage of the native multicast support provided by current network equipment, such as switches and network interface cards. By limiting ourselves to the LAN/administrative domain case, we are able to make certain assumptions not acceptable in the WAN case. We envisage that instances of our event notification service software are executed inside LANs, but connect to a WAN event notification service, by e.g. gateways/routing daemons. We therefore view our work as complementary to WAN event notification services.

In our current implementation so-called mapping specifications are generated manually, but may be changed during runtime. The principle used to determine such specifications is to map notifications generated at a high rate onto separate multicast channels, i.e. to isolate such high rate traffic. Our approach is useful for a large class of applications and also a natural first step towards a more dynamic solution, where mappings are generated and updated automatically during runtime.

The rest of the report is structured as follows. First we provide some background information in Section 2. Then we present the requirements for our event notification service in Section 3. Based on these requirements, we describe the architecture of our service in Section 4. In Section 5 we describe our prototype. In Section 6 we describe some experiments and present some empirical results. In Section 7 we discuss some ideas for further work. Lastly, in Section 8 we conclude.

Method
$publish(notification\ n)$
$subscribe(string\ identity,\ pattern\ expression)$
$unsubscribe(string\ identity,\ pattern\ expression)$
$advertise(string\ identity,\ filter\ expression)$
$unadvertise(string\ identity,\ filter\ expression)$

Table 1: Interface of SIENA

2 Background

In this section we provide some more background information for content-based publish subscribe systems. Our description is biased towards SIENA[2], on which we have based our prototype implementation.

2.1 Event Notification Service API

An event notification service typically provides a method used by objects of interest to publish notifications and a method used by interested parties to register interest in notifications. Additionally, the event notification service may provide a method used by an object of interest to inform the event notification service about the kind of notifications potentially generated. Methods for unregistering are typically also available. As an example, Table 1 illustrates the interface of SIENA. Objects of interest and interested parties must identify themselves to the event notification service, which maintains references to the clients. A pattern is basically a sequence of filters[2], but in the rest of the paper we assume that filters are used for subscriptions.

2.2 Event Notifications

In SIENA, an event notification is basically a set of *type, name, value* tuples. The most common types are supported, e.g. string, integer, float, etc.

An example of a notification is given in Table 2. This particular notification contains a small region of a video frame - the luminance part, encoded in 8 bits of resolution. The video frame region is a rectangular block, $16x16$ pixels. The encoding is represented as a string, for illustration purposes. This block is intra coded, i.e. independent from blocks in earlier and later frames. The *pixels* attribute contains the pixel values for this $16x16$ block, illustrated as characters. This particular block has a $(horizontal, vertical)$ placement within the frame of $(1, 5)$. Video client software may translate spatial and temporal requirements into subscriptions, based on the particular encoding scheme used.

In this report we have used real-time video streaming as an example of an application domain requiring a high performance event notification service. The challenge of supporting heterogeneous receivers in video streaming applications is well known. As an example, a combination of a layered video compression algorithm and receiver driven multicast is described in [7], providing scalable multicast video transmission. Each layer encodes a portion of the video signal and is sent to a designated IP multicast address.

In an event-based approach, each video frame may be published as a number of notifications. E.g. in [3], an extension for the CORBA Event Service is described which supports stream events and multicast delivery. Content-based publish/subscribe systems have the potential of supporting even more fine grained selection, compared to direct use of multicast or a channel-based approach. Interested parties may subscribe to only a certain part of a video stream as explained above and thereby reduce resolution both spatially and temporally. Additionally, a content-based approach may better support parallel processing, as pointed out in [4].

Type	Name	Value
string	*media_type*	*video*
string	*media_source*	*fnasa.simula.no*
string	*encoding*	*/raw/luminance/8/16x16*
string	*block_type*	*intra_coded*
integer	*block_h*	1
integer	*block_v*	5
byte[ ]	*pixels*	*q34i23QR ... D*

Table 2: Event Notification Example

2.3 Filters

A filter is a sequence of attributes and constraints on the attributes. The constraints are specified in a constraint language, which also defines some supported operators. The expressiveness of such languages may differ, but an important design issue is the balance between the expressiveness of the language and the associated computational complexity[2]. A filter may be used in different contexts - for subscriptions or advertisements.

An example of a filter for subscriptions is given in Table 3. This filter may be used to express interest in notifications from a particular source, which contain video data, with a particular encoding, with a particular block type, but only the 10 leftmost (block) columns.

An example of a filter for advertisements is given in Table 4. This filter may be used to advertise notifications which will contain parts of a particular encoded video stream, but only the intra encoded blocks and only the two leftmost columns, where *block_h* is 0 or 1.

2.4 Filtering

In very simple publish/subscribe systems, each server may broadcast notifications to all other servers. All servers connect to a well known multicast address and notifications are sent to the multicast address and are then forwarded by the *multicast service*. Each server then filters notifications on behalf of their interested parties. The result of this late filtering is reduced scalability, as both network bandwidth and processing resources are wasted. This is the approach used in Mbus[10]. Mbus is designed to support *coordination and control* between different application entities hosted by different computers on a LAN. In [5] we have evaluated the suitability of Mbus as a LAN event notification service.

For most applications the number of notifications is likely to be significantly larger than the number of subscriptions. Hence a better approach is to broadcast subscriptions, which are then used to prune the delivery of notifications. The gain of this strategy clearly depends on the ratio of notifications to subscriptions.

Similarly, in publish/subscribe systems which support advertisements, the advertisements may be broadcast and used to prune the delivery of subscriptions, which are then used to reduce the flow of notifications.

It should be noted that in the last two approaches, subscriptions/advertisements are not forwarded unless they are more general than the current forwarded ones.

In Hermes[11], a different approach is used. So-called rendezvous nodes ensure that subscriptions and advertisements meet somewhere between objects of interest and interested parties, without any global broadcasts.

2.5 The Covering Relation

The covering relation is described in [2] and is important in order to understand the rest of the report. The relation $x \prec_Y^X y$ is read as x *matches* y or alternatively y *covers* x. X and Y indicate the type of x and y respectively and may be of type N (Notification), S (Subscription filter), or

Type	Name	Value/Expression
string	*media_type*	*video*
string	*media_source*	$fnasa.simula.no$
string	*encoding*	$/raw/luminance/8/16x16$
string	*block_type*	*intra_coded*
integer	*block_h*	$>= 0$
integer	*block_h*	< 10

Table 3: Subscription Filter Example

Type	Name	Value/Expression
string	*media_type*	*video*
string	*media_source*	$fnasa.simula.no$
string	*encoding*	$/raw/luminance/8/16x16$
string	*block_type*	*intra_coded*
integer	*block_h*	$>= 0\ AND\ < 2$
integer	*block_v*	*ANY*
byte[ ]	*pixels*	*ANY*

Table 4: Advertisement Filter Example

A (Advertisement filter). In the following, the symbol α represents an attribute in a notification and the symbol ϕ represents an attribute constraint in a subscription or advertisement filter. The most important relations are:

$\alpha \prec \phi \Leftrightarrow type_\alpha = type_\phi \wedge name_\alpha = name_\phi \wedge operator_\phi(value_\alpha, value_\phi)$: The attribute α matches the attribute constraint ϕ if and only if the types and names are identical and the operator returns true

$n \prec_S^N s \Leftrightarrow \forall \phi \in s : \exists \alpha \in n : \alpha \prec \phi$: : The notification n matches the subscription filter s if and only if each and every attribute constraint in s is matched by an attribute in n. Multiple constraints for the same attribute is interpreted as a conjunction

$n \prec_A^N a \Leftrightarrow \forall \alpha \in n : \exists \phi \in a : \alpha \prec \phi$: The notification n matches the advertisement filter a if and only if each attribute in n is matched by an attribute constraint in a. Multiple constraints for the same attribute is interpreted as a disjunction

$s \prec_A^S a \Leftrightarrow \exists n^N : n \prec a \wedge n \prec s$: The subscription filter s matches the advertisement filter a if and only if there exists a notification n which matches both s and a. In other words, if the set of notifications defined by s and the set of notifications defined by a have nonempty intersection

2.6 The Mapping Problem

In general, each notification is of interest to a subset of all interested parties. Theoretically and ideally, a multicast address may be used for each possible combination of computers hosting the interested parties. In practice this is not possible, as the required number of multicast addresses grows exponentially and quickly beyond practical limits.

In [9], some algorithms are presented, targeting this mapping problem. In the article, brokers, and not the computers hosting clients, are treated as the communication endpoints. In addition to the theoretically ideal algorithm, five algorithms are presented. The principles used in order to reduce the required number of multicast groups are to *reduce group precision* (brokers receive and filter out notifications which are of no interest to its clients), *send multiple multicast*, and

send over multiple hops. In all algorithms, except a so-called *group approximation algorithm*, the mapping is static. Simulation results on a wide area network are presented. The authors find that a flooding approach is viable over a range of conditions, but in case of high selectivity and high regionalism of subscriptions the non flooding approaches are significantly better.

3 Service Requirements

In this section we describe the requirements for our scalable and high performance LAN event notification service.

3.1 Exploit Locality

The programming of applications utilizing event-based interaction should to a reasonable extent be handled independently from the deployment, i.e. where clients are instantiated and where they are executed. A different API should not be necessary, e.g. when an object of interest and an interested party for performance reasons are deployed on the same computer. Therefore, our event notification service must *efficiently* support:

- *intra LAN communication*: between objects of interest and interested parties hosted by different computers connected via a LAN
- *intra host communication*: between objects of interest and interested parties hosted by different processes on the same computer
- *intra process communication*: between objects of interest and interested parties hosted by a single process

The first case is important for distributed applications executing on a LAN. Although LANs offer vast amounts of bandwidth and short delay, this is not automatically the case for an event notification service deployed on top of it. Care must be taken in order to provide performance close to the bare hardware capabilities. A high performance service may allow applications within the domain of high performance parallel computing to utilize event-based interaction for efficient communication, e.g. between powerful computers within a cluster or on a LAN.

By supporting the second case, it becomes easier to take advantage of the processing capabilities of multiprocessor computers. Additionally, different processes and hence address spaces provide protection, both for a single user and between different users. It also allows application development by using a single computer.

Considering the third case, if notifications published by an object of interest are not of interest to any clients outside the process itself, then no such notifications should ever leave the process. Having both objects of interest and interested parties inside the same process is useful in order to exploit locality, e.g. to avoid copying large amounts of data between different address spaces. The exchange of notifications in this case must happen directly, i.e. without relying on some other process hosted by the same or another computer.

3.2 Utilize Multicast

The event notification service must also be able to take advantage of native multicast support in order to reduce the demand for both *processing* and *network* resources.

By utilizing multicast, only a single send operation is required by a server in order to publish a notification. In effect, the processing requirements are independent of the number of other computers hosting interested parties and the number of interested parties hosted by each computer.

With respect to network resources, the benefit of using multicast depends on both the applications and the underlying LAN technology. For LAN technologies which are broadcast by nature (e.g. wireless) or by design (e.g. traditional Ethernet), the cost of sending a single packet is basically the same for multicast and unicast. If the event notification service uses one to one transport

layer communication, each notification in effect is broadcasted several times, i.e. *all* computers on the LAN receive the same notification several times, dramatically reducing the performance. For switched wired LANs, supporting multicast natively, the situation is somewhat similar. If the event notification service is incapable of utilizing native multicast and there are several computers hosting interested parties, each notification will propagate over some links several times.

It is important to consider the mapping of network layer multicast onto link layer multicast, because this mapping is not always one to one. As an example, several IP multicast addresses could map to the same Ethernet multicast address.

An event notification service utilizing IP multicast inherits its dynamic properties. An IP multicast address is dynamically associated to a group of computers, by means of protocols such as IGMP (Internet Group Management Protocol). As an example, consider the case where a server is hosted by a computer for which the IP address is changed. As long as the computer continues to register interest in the multicast addresses, other servers hosted by other computers do not need to be aware of this change. As a result, IP multicast may also simplify runtime reconfiguration.

3.3 Support Runtime Reconfiguration

We expect that a large class of applications built on top of an event notification service may have rather dynamic characteristics, but along different dimensions. When considering an event notification service architecture, it is important to distinguish between changes in: the *number of objects of interest*, the *number of interested parties*, the *location* of clients, the *notification types* used by objects of interest, the notification *publishing rates*, and the *subscriptions* made by interested parties. Therefore, in order to adapt to the communication pattern of the applications, it should be possible to change the way notifications are mapped onto multicast addresses during runtime, without affecting the semantics of the service. Such reconfigurations should happen quickly and the performance should remain close to normal during such periods.

3.4 Provide Robustness

A distributed event notification service should provide robustness and tolerate certain failures. As an example, process, operating system, host, and link failures must not render the whole service useless. A link failure may partition the LAN into groups of computers which are not able to communicate with each other. However, the event notification service should still continue to operate inside such partitions. The value of the service should degrade gracefully.

4 Architecture

Based on these requirements, we now describe the architecture of our event notification service. First we discuss some assumptions which are typically acceptable for LANs, but not always for WANs.

4.1 Assumptions

For LANs it is reasonable to assume a single administrative domain. It is also reasonable to assume that IP multicast is (made) available within a domain and that the network equipment supports multicast natively. The mapping between notifications and IP multicast addresses may be done locally, within the domain, and administratively scoped IP multicast addresses may be used (RFC2365). In other words, the mapping is invisible outside the administrative domain.

The number of computers inside a LAN is also relatively limited, which is important when considering both architecture and algorithms. Additionally, stationary computers within a LAN often have relatively large amounts of computational capacity.

For wired equipment inside a LAN it is reasonable to assume low latency, typically sub millisecond, and lots of bandwidth, typically 100Mbps - 1 Gbps switches and network interface cards. Additionally, both jitter and the risk of packet loss are likely to be low.

4.2 Exploit Locality

A single server may be sufficient within a LAN as long as the number of objects of interest, the number of interested parties, and the publication rates are low. In this case, it may even be considered reasonable that notifications for which there are no interested parties are filtered at the server side. However, as the publishing rate increases and the number of objects of interest increases a quenching mechanism becomes important.

Similarly, as the number of clients which have interest in the same notifications increases, a server which handle each client separately by utilizing unicast will run out of steam.

By replacing the single server with a number of servers the processing is distributed between the servers, but the total network bandwidth consumption will most likely increase.

A native multicast approach is required in order to reduce this bandwidth consumption problem and to distribute notifications to a potentially very large number of interested parties. But in order to utilize native multicast, the computers hosting clients also must execute some software in order to determine which multicast addresses to subscribe to.

This reasoning indicates that each computer which hosts clients also should execute some software in order to handle quenching and subscriptions to IP multicast addresses. Therefore, in our architecture each computer hosting clients execute part of the event notification service, i.e. the software responsible for the intra process, the intra host, and the intra LAN event notification service. As a result, the event notification service software is executed cooperatively by computers within a LAN, which are often fairly powerful. However, some computers may act as *dedicated servers*, i.e. hosting no clients, or *thin clients*, i.e. interacting with the service through a server on another computer, but in this report we will not discuss these cases any further.

The software for the intra process case provides clients with the event notification service API. The intra host software is responsible for aggregating subscriptions for all the clients on the host as well as for executing the LAN event notification service protocol. In which context the software is actually executed on a particular computer is an implementation and deployment issue which may be realized in different ways, e.g. within a client process, within a separate process, within the operating system, or combinations of these. As an example, the intra process software may be implemented as a library, while a possible implementation of the intra host software may be a daemon process which is started whenever the first client is instantiated on a particular computer. In this case, the daemons hosted by different computers within the LAN exchange information and cooperatively realize the distributed event notification service.

In the following we will continue to use the term "server" and generally assume that all computers which host clients also host servers.

In our current approach, each server informs the other servers about the most general subscriptions made by their locally interested parties, i.e. subscriptions are used in order to prune the delivery of notifications.

4.3 Utilize Multicast

In order to take advantage of multicast, the challenge of mapping event-based communication onto multicast communication must be addressed. In the following we discuss our approach and issues related to this mapping problem. Note that we plan to extend our event notification service in order to take advantage of different transport protocols concurrently, but in this paper the emphasis is on utilizing native multicast support.

Additionally, note that the service described in this paper does not give any guarantees with respect to race conditions. As an example, interested parties may receive notifications even after the *unsubscribe*() method has been called. Clients are required to handle such cases. This is similar to the *best-effort* service as provided by SIENA[2].

Advert. Filter	$\rightarrow$	Communication Identifier
f_1^A	$\rightarrow$	$udp\_ipm : //239.0.10.1 : 6666$
f_2^A	$\rightarrow$	$udp\_ipm : //239.0.10.2 : 6666$
f_3^A	$\rightarrow$	$udp\_ipm : //239.0.10.3 : 6666$
f_4^A	$\rightarrow$	$udp\_ipm : //239.0.10.4 : 6666$

Table 5: Mapping Specification Example

4.3.1 Mapping Specification

Table 5 gives an example of a mapping specification, where each row specifies an advertisement filter and a communication identifier. The communication identifier consists of a protocol name and a protocol specific part. The protocol used for all entries in this table is *udp_ipm*, our protocol for encapsulating notifications in UDP packets and transmission by IP multicast. The protocol specific part specifies different IP multicast addresses and a port number.

For now we assume that each server has a private copy of the mapping specification table. The table is required in order to handle subscriptions and publications.

First we describe how a subscription made by an interested party may make a server register interest in IP multicast addresses. Then we describe how a server maps notifications onto IP multicast addresses, which are then forwarded to the appropriate servers by the multicast service.

4.3.2 Subscriptions

When an interested party subscribes with the filter s^S as parameter, its server (executing on the same computer) checks if s^S is covered by subscriptions already made by any of its clients. If s^S is covered by current subscriptions, the server just register this interested party.

Otherwise the server must also make sure all the other servers become aware of this new subscription (e.g. by sending s^S on a well known multicast address, which all servers have registered interest in). Additionally, the server consults the mapping specification in order to determine which communication channels may potentially carry notifications covered by s^S. The table is checked sequentially. If f_j^A covers any notifications which are also covered by s^S, i.e. $s \prec_A^S f_j$, the server must make sure it will receive these notifications. As an example, if s^S is the subscription filter in Table 3 and f_2^A is the advertisement filter given in Table 4, then the server must register interest in the multicast address specified by $udp\_ipm : //239.0.10.2 : 6666$, since $s \prec_A^S f_2$. Observe that a subscription filter may cover (partially) *several* advertisement filters. In order to maintain the semantics of the service, the server therefore may have to register interest in several multicast addresses.

4.3.3 Publications

When an object of interest publishes a notification n^N, its server (executing on the same computer) checks if any subscriptions made by other servers cover n^N. If this is the case, the mapping specification is consulted in order to determine the associated communication identifier. The advertisement filters are checked sequentially. If n^N is covered by f_j^A, then the server sends n^N to the associated multicast address. The server also checks if there are any locally interested parties. If this is the case, these are also notified.

As an example, assume that n^N is the notification given in Table 2, s^S is the subscription filter given in Table 3, and f_2^A is the advertisement filter given in Table 4. If f_1^A does not cover n^N and f_2^A covers n^N and there is another server which has made the subscription s^S on behalf of its client(s), then n^N is sent to the multicast address specified for f_2^A, i.e. $udp\_ipm : //239.0.10.2 : 6666$.

Note that in our current architecture a notification is sent only once, on the multicast address associated with the first advertisement filter which covers the event notification.

4.3.4 Mapping Mismatch and Filtering

A notification is only forwarded by a server if it is covered by one or more subscriptions. If there is only a single interested party, only notifications covered by its subscriptions are forwarded. Filtering is performed early, by the server executing on behalf of an object of interest.

Depending on both the current mapping specification and the current subscriptions, some filtering may happen on the interested party side. If an interested party has specified a restrictive subscription filter and another interested party has specified a more general subscription filter and all notifications are mapped to the same multicast address, then the server executing on behalf of the first interested party must discard some notifications arriving on the multicast address.

The penalty for mapping mismatches is paid in terms of wasted network bandwidth and computational resources, raising the question of how to determine mapping specifications.

4.3.5 A Simple Mapping Heuristic

The following simple heuristic is currently used to generate mappings: Notifications generated at a high rate, of large size, and not of interest to all interested parties are mapped onto separate multicast addresses.

E.g., assume that a single mapping entry is used initially, which maps all possible notifications to a single multicast address. The effect of this mapping specification is late filtering, i.e. by servers hosting interested parties. As long as the rate of notifications is low or all interested parties have similar interests, this is most likely acceptable for a LAN service. For broadcast based LANs this is the effect anyway from a network point of view, although not from a processing or power consumption point of view.

However, a problem arises if one or more objects of interest start generating notifications at a very high rate (e.g. for publishing real-time video) and only some of the interested parties have subscribed to these notifications. Many servers would then have to discard these high rate event notifications, wasting network and processing resources.

Our approach allows a new entry to be installed into the mapping specification table, which maps all notifications part of a high rate notification "stream" to a different multicast address. When the mapping specification is updated, each server must check subscriptions made by its interested parties against the new mapping specification and register interest in the appropriate multicast addresses. Following the example above, the servers executing on behalf of clients interested in the high rate notifications (e.g. the video stream) would find their subscriptions to match the new advertisement filter and register interest in the newly announced multicast address. Similarly, the server executing on behalf of the publisher (e.g. the video server), would map and then send these high rate event notifications to this multicast address. As a result, the event notifications generated at a high rate are only forwarded to computers hosting interested parties which have actually subscribed to (some of) these notifications.

4.4 Runtime Reconfiguration

In the following we argue that some of the runtime changes discussed in Section 3.3 are handled by the IP multicast service while others should be handled by updating the mapping specification.

Due to the distributed architecture, changes in the *number of objects of interest* and the *number of interested parties* will have little effect on the service. The consumption of processing and network resources is distributed between the computers hosting clients. Additionally, by utilizing IP multicast, changes in the number of interested parties will have little impact on the service. A notification is sent at most once by a server anyway.

The dynamic properties of IP multicast also simplifies changes in client *location*. An object of interest which changes location may continue to publish notifications through a server. The computer hosting the server may send to IP multicast groups without joining them. An interested party which changes location will continue to receive notifications since its server registers interest in the appropriate IP multicast addresses and the computer joins IP multicast groups accordingly. Consequently, the number of clients and their location may change radically during runtime.

The *notification types* used by objects of interest are not likely to change very often. But by updating the mapping specification during runtime and hence reconfiguring the service, new types of notifications may get introduced and efficiently handled.

With respect to changes in notification *publishing rates*, some objects of interest may generate notifications with a relatively fixed rate while others may generate notifications sporadic. If it is possible to determine the parts of the "event notification space" *potentially* generated at a high rate, either a priori or during runtime, a mapping specification which partitions the "event notification space" accordingly may be used to reconfigure the service. For all notifications generated at a low rate, a single or only a few multicast addresses are sufficient since the client side filtering in this case is acceptable.

Clients which frequently subscribe and unsubscribe to the *same* notifications are handled similarly to changes in the number of interested parties. For interested parties which frequently change their *subscriptions* in order to receive a different part of the "event notification space", such changes are most likely limited to within part of the "event notification space". E.g., client software written in order to receive notifications carrying stock ticker information will not register interest in notifications carrying video data. By updating the mapping specifications, different parts of the "event notification space" may be further partitioned or merged in order to better match the subscriptions made by clients.

In effect, the mapping specification introduces a level of indirection between event-based communication and the underlying multicast communication and allows runtime reconfiguration. The IP multicast service is also capable of handling some of the runtime changes. Our approach, of manually specifying a mapping, is valuable when the mapping specification needs relatively few updates in order to maintain efficiency.

The problem of distributing a new mapping specification to all servers is not addressed in this paper. But in order to maintain the semantics of the service, it is important that either all servers change the mapping specification or none. This is a well known problem in the distributed systems field for which many techniques exist.

4.5 Robustness

The robustness of our architecture is due to the fact that there are no central computers (ignoring configurations with dedicated servers and thin clients). Each computer hosting one or more clients, also executes part of the event notification service. We use a soft state approach, relying on refresh and timeout mechanisms, in order to handle crashed processes, crashed computers, link failures, etc. Each server (hosted by some computer) periodically informs the other servers (hosted by other computers) about the notifications of interest to any of its clients, e.g. by sending the aggregated subscriptions on a well known multicast address. Each server expires subscriptions which have not been refreshed for some time. The reason for subscriptions not being refreshed, is of no concern. As an example, if a server does not receive any subscriptions for some time due to e.g. a link failure, then no notifications generated by its client(s) are sent. Whenever the failed link comes up and the server starts receiving relevant subscriptions, the server will again start sending notifications.

5 Prototype

Our prototype is based on the event notification service software developed in the SIENA[2] project. More specifically we have extended the software provided in the siena-java-1.4.2.tar.gz package. This software is written in Java and so are our extensions.

In our current prototype, the server software is linked into the client code and executes within the same address space, as illustrated in Figure 2. A server executes the intra process, the intra host, and the intra LAN event notification service software on behalf of clients hosted by the same process. It should be noted that we consider restructuring this software in order to have a single

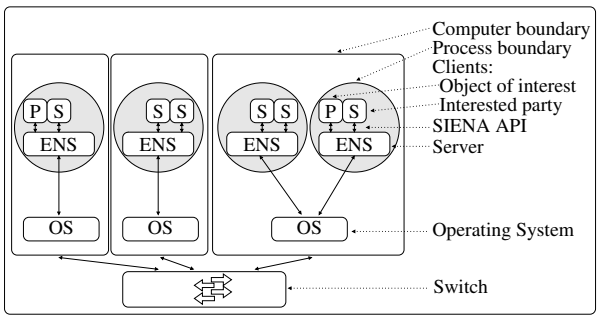

Figure 2: The LAN Event Notification Service

instance responsible for aggregation on a host basis and another part which is executed within each process.

Each server maintains references to its interested parties and their subscriptions. The server also maintains state to keep track of subscriptions made by other servers (on behalf of their clients) in order to determine if a notification is of interest to any other server, hosted by another process on the same computer or another computer on the LAN.

5.1 Intra Process Communication

For intra process communication, a server relies on method calls. Interested parties must implement an interface which define a so-called *notify* method. When a server receives a notification, the server notifies all its clients, for which the subscriptions cover the notification.

5.2 Intra host and Intra LAN Communication

Currently, IP multicast is used to forward notifications from one server to other servers, both within a single computer and between different computers on a LAN.

IP multicast provides some mechanisms which determine if an IP multicast packet is delivered to other processes on the same computer and if the packet is sent out on the LAN. An IP multicast packet is sent out on the LAN if the value of the *time to live* field is 1 or larger. If a so-called *loopback* socket option is set, then a packet is delivered to the other processes on the same computer which have registered interest in this particular IP multicast address. These two mechanisms may be used to forward notifications to other servers hosted by this computer, other servers hosted by other computers on the LAN, or both.

If two servers hosted by the same computer register interest in the same multicast address, then only a single instance of each packet is received by this computer. Aggregation is handled by the multicast software in the operating system, i.e. packets containing event notifications are copied to the different servers by the operating system.

5.3 Subscription Forwarding

Each server aggregates subscriptions on behalf of their interested parties and periodically forwards these subscriptions to all other servers by IP multicast. A separate IP multicast address is used, in order to reduce the risk of operating system buffers being overwritten. Currently, each server forwards its subscriptions independently of the subscriptions made by other servers, i.e. subscriptions are not aggregated, neither on a host basis nor on a LAN basis.

5.4 Packet Senders and Packet Receivers

Servers rely on so-called *packet senders* and *packet receivers* in order to send and receive notifications respectively. An instance of a packet receiver is handled by a separate thread. The thread is waiting for packets on a particular multicast address. A packet sender on the other hand does not have any associated thread, but is executed by the calling thread. Packet senders and packet receivers are handled by a soft state approach, i.e. they are instantiated on demand and timed out whenever not used for some time.

5.5 Outline of Server Algorithm

Each server performs the following actions:

- Periodically: (1) Forwards aggregated subscriptions to the other servers, (2) time out subscriptions which have not been refreshed, and (3) time out unused packet senders and packet receivers
- When one of its interested parties subscribes: Unless the subscription is covered by earlier subscriptions made by its interested parties, immediately forwards the subscription to the other servers
- When a subscription is received from another server: Stores/resets timeout value for the received subscription
- When a notification is received from one of its objects of interest: (1) Unless the notification is not of interest to any other server, forwards the notification to the multicast address associated with a covering advertisement filter and (2) notifies by method call each of its interested parties which have subscriptions covering the notification
- When a notification is received from one of the other servers: Notifies its interested parties which have subscriptions covering the notification, by method calls
- On demand: (1) Instantiates packet senders/receivers or (2) updates mapping specification

6 Empirical Results

In the following we describe the experiments conducted in order to measure the performance and the scalability of our service.

6.1 Environment

For the experiments, standard dual 2GHz AMD Athlon PCs running the Linux 2.4.19 operating system have been used. The PCs were connected by 100Mbps (Mbits per second) switched Ethernet, provided by a Cisco Catalyst 2950XL switch. The switch was configured with IGMP snooping enabled, a technique where the switch maps network layer multicast to link layer multicast by looking for e.g. IGMP host join messages encapsulated within the IP part of packets. The computers were equipped with Intel Ethernet Pro 100 and 3Com 3c905C network interface cards. The software was compiled and executed by a standard Java edition from SUN, version 1.4.1-b21.

6.2 Experiments

For the experiments some client software was written - the object of interest and the interested party code. Each notification had the following attributes: source, type, sequence number, and array. The mapping specification had four advertisement filters, where the source and the type attributes and their values were used to map notifications to potentially four different IP multicast addresses. Interested parties used the sequence number value for measuring the number of

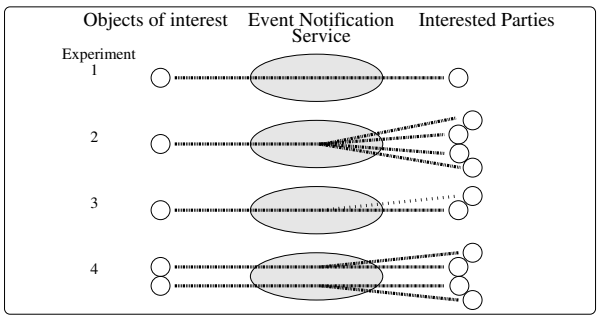

Figure 3: The Different Experiment Configurations

notifications received per second. The length of the array was used to adjust the size of the notifications. A maximum size of 1450 bytes/notification was chosen in order to avoid fragmentation in the protocol stack. The publishing rate for the object of interest was configurable.

The different experiment configurations are illustrated in Figure 3. Each experiment was expected to give information about a certain aspect - (1) the throughput, when notifications are forwarded from an object of interest to a single interested party, (2) the scalability, when several interested parties register interest in the same notifications, (3) the ability to support interested parties with different needs, and (4) the ability to map parts of the "event notification space" to different multicast addresses in order to provide isolation between different (parts of) applications.

Note that the mapping specifications and the subscription filters used in the following experiments have been configured manually to test the performance potential of our service. Clearly, poorly chosen mapping specifications may have reduced the performance.

6.2.1 Experiment 1: One to One Throughput

In the first experiment, a single interested party subscribed to the notifications generated by a single object of interest. The purpose of this experiment was to measure the maximum number of notifications per second transfered between an object of interest and an interested party, for the the intra LAN, the intra host, and the intra process cases. Each client was hosted by a process which also hosted an instance of the server. In the intra process case, a single server was shared between the object of interest and the interested party.

In order to avoid buffer overruns and loss of notifications for the intra LAN and intra host cases, the size of the socket buffers in the operating systems were increased to 2 MBytes. Without this increase lots of notifications were lost, especially for large sized notifications.

The object of interest (and its server) was capable of publishing roughly twice as many notifications per second as the interested party (and its server) was able to receive. Therefore, the publication rate was reduced for the intra host and the intra LAN cases, in order to match the maximum receive rate of the interested party.

The measured throughput, in notifications per second, is given in Table 6. The intra process case provides best performance, although only a single CPU is utilized. The thread which invokes the publish method executes both the server code and the notify method of the interested party. For the intra host and the intra LAN experiments, two CPUs were utilized concurrently, either on the same computer or on different computers.

For the intra LAN and the intra host cases a maximum of approximately 6.5 MBps (MBytes per second), roughly 52Mbps, was measured. These tests were CPU bound, limited by the maximum receive rate of the interested party. Note that more than half the network link capacity was utilized. For comparison, a television quality MPEG-2 encoded video, Main profile in the Main Level, 720 pixels/line $*$ 576 lines, requires maximum 15Mbps[13].

Locality	Notification size in bytes			
	100	500	1000	1450
	Notifications received per second			
Intra LAN	9000	7000	5500	4500
Intra Host	9000	7000	5500	4500
Intra Process	115000	100000	85000	75000

Table 6: The Maximum Number of Notifications Received per Second

6.2.2 Experiment 2: One to Many Scalability

The purpose of the second experiment was to measure the scalability of the service. Four interested parties, each hosted by a separate computer, subscribed to and received the same notifications.

For this experiment, the measured numbers of notifications received per second by each interested party, were *the same* as in the intra LAN case in the first experiment. Hence, the three additional subscribers did not affect the server executing on behalf of the object of interest, neither with respect to processing nor with respect to network bandwidth consumption. The switch was able to handle the copying of packets to the appropriate ports.

It should be noted that if the event notification service had not been able to utilize multicast, the computer hosting the object of interest would have become IO bounded, i.e. the maximum rate of the network link would have been exceeded. In the 1450 bytes/notification case, a unicast-based service would have hit an IO bottleneck even for only two interested parties. The aggregated data rate for the four interested parties in the 1450 bytes/notification case was 26.1 MBps (4 * 4500 notific./sec. * 1450 bytes/notific.).

6.2.3 Experiment 3: One to Many Heterogeneity

The purpose of the third configuration illustrated in Figure 3 was to verify that our event notification service is able to support interested parties hosted by heterogeneous computers and/or network connections. Each client was hosted by a separate computer. One of the interested parties subscribed to and received only some of the event notifications, i.e. only notifications with a particular value for the type attribute. The mapping specification used, mapped these notifications to a separate IP multicast address.

The measurements confirmed that consumption of both network bandwidth and processing resources were reduced accordingly for this interested party and its server.

6.2.4 Experiment 4: Many to Many Isolation

The purpose of the forth experiment illustrated in Figure 3, was to verify that different (parts of) applications may be isolated by using an appropriate mapping specification. Two objects of interest generated notifications with different source attribute value. The mapping specification used, mapped notifications with different value for the source attribute to different IP multicast addresses. The notifications from each object of interest were received by two interested parties. Each client was hosted by a separate computer.

The measured numbers of notifications per second, received by each interested party, were the same as in the intra LAN case in the first experiment. The aggregated publishing rate for the 1450 bytes/notification experiment was 13.05 MBps (2 * 4500 notific./sec. * 1450 bytes/notific.). For a 100Mbps LAN based on broadcast technology, the throughput most likely would have been reduced. This indicates the strength of our event notification service when coupled with switched LAN technology with native multicast support.

7 Further Work

In our further work, we will develop an algorithm for calculating mapping specifications in order to handle dynamic changes in applications and the environment in a more adaptable way. The input to such an algorithm may include the number of multicast addresses, the LAN characteristics (e.g. broadcast, switched), information about imperfections in network to link layer multicast mapping (e.g many to one), some statistics about the past as well as the likely future. The information about the past may be provided by each server measuring and generating statistics about the notifications received and required and the notifications received but discarded. The information about the future may be QoS parameters included in advertisements made by objects of interest, e.g. notification rate, size, and distribution.

We plan to enhance our event notifications service to concurrently utilize a combination of different protocols. The motivation is that different parts of applications have different requirements with respect to e.g. throughput, reliability, and delay. Clients may then indicate QoS parameters in subscriptions and advertisements.

We also would like to avoid the broadcast of subscriptions between servers. A server could hold back subscriptions already covered by subscriptions made by other servers. This is similar to the approach used by IGMP, where only one host sends a membership report for a particular IP multicast address during each time interval.

8 Conclusion

Event-based interaction is inherently many to many communication. Therefore, event-based communication does not map well, performance wise, onto one to one communication primitives. The challenge of utilizing network and link layer multicast support for event notification services is well known, but to our knowledge no implementations for content-based publish/subscribe systems exist.

In this paper we have presented the architecture of a distributed content-based event notification service where notifications are mapped onto multicast communication. The service is targeted at usage within a local area network or an administrative domain. We envisage that the service will be connected to a wide area network event notification service by means of a gateway.

A prototype has been implemented and experiments confirm that our service has a potential for providing both high performance and scalability. Objects of interest may publish several thousand notifications, carrying several MBytes of data, per second. For the experiments performed, the service was unaffected by the number of interested parties, due to the ability of the service to take advantage of native multicast support in network and end system devices.

The scalability and performance allows new application domains to take advantage of event-based interaction. The performance is e.g. more than sufficient for real-time streaming of very high quality video. Application domains requiring parallel processing, e.g. real-time content analysis, may also take advantage of such a service.

9 Acknowledgments

We would like to thank all persons involved in the DMJ (Distributed Media Journaling) project for contributing to the ideas presented in this paper. The DMJ project is funded by the Norwegian Research Council through the DITS program, under grant no. 126103/431.

References

[1] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems. In *Proceedings of ICDCS*, pages 262–272. IEEE, 1999.

[2] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.

[3] D. Chambers, G. Lyons, and J. Duggan. Stream Enhancements for the CORBA Event Service. In *Proceedings of the ACM Multimedia (SIGMM) Conference, Ottawa*, pages 61–69, October 2001.

[4] V. S. W. Eide, F. Eliassen, O.-C. Granmo, and O. Lysne. Scalable Independent Multilevel Distribution in Multimedia Content Analysis. In *Proceedings of the Joint International Workshop on Interactive Distributed Multimedia Systems and Protocols for Multimedia Systems (IDMS/PROMS 2002), Coimbra, Portugal*, LNCS 2515, pages 37–48. Springer-Verlag, Nov. 2002.

[5] V. S. W. Eide, F. Eliassen, O. Lysne, and O.-C. Granmo. Real-time Processing of Media Streams: A Case for Event-based Interaction. In *Proceedings of 1st International Workshop on Distributed Event-Based Systems (DEBS'02), Vienna, Austria*, pages 555–562. IEEE Computer Society, July 2002.

[6] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys (CSUR)*, 35:114–131, June 2003.

[7] S. McCanne, M. Vetterli, and V. Jacobson. Low-Complexity Video Coding for Receiver-Driven Layered Multicast. *IEEE Journal of Selected Areas in Communications*, 15(6):983–1001, August 1997.

[8] Object Management Group Inc. CORBA services, Event Service Specification, v1.1. *http://www.omg.org/*, 2001.

[9] L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R. Strom, and D. Sturman. Exploiting IP Multicast in Content-Based Publish-Subscribe Systems. In *Proceedings of Middleware 2000*, LNCS 1795, pages 185–207. Springer-Verlag, 2000.

[10] J. Ott, C. Perkins, and D. Kutscher. A message bus for local coordination. *RFC3259*, 2002.

[11] P. R. Pietzuch and J. M. Bacon. Hermes: A Distributed Event-Based Middleware Architecture. In *Proceedings of 1st International Workshop on Distributed Event-Based Systems (DEBS'02), Vienna, Austria*, pages 611–618. IEEE Computer Society, July 2002.

[12] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content Based Routing with Elvin4. In *Proceedings of AUUG2K, Canberra, Australia*, June 2000.

[13] R. Steinmetz and K. Nahrstedt. *Multimedia: Computing, Communications & Applications.* Prentice Hall, 1995.

[14] TIBCO Software Inc. TIBCO Rendezvous FAQ. http://www.tibco.com/solutions/products/active_enterprise/rv/faq.jsp, 2003.