

# Analyzing the behavior of event dispatching systems through simulation

Giovanni Bricconi<sup>1</sup>, Elisabetta Di Nitto<sup>2</sup>, Alfonso Fuggetta<sup>2</sup>, Emma Tracanella<sup>1</sup>

<sup>1</sup> CEFRIEL, via Fucini 2, 20133 Milano, Italy  
{bricconi tracanel}@cefriel.it

<sup>2</sup> DEI, Politecnico di Milano, piazza L. da Vinci 32, 20133 Milano, Italy  
{dinitto fuggetta}@elet.polimi.it

**Abstract.** The pervasive presence of portable electronic devices and the massive adoption of the Internet technology are changing the shape of modern computing systems. Applications are being broken down into smaller components that can be modified independently and that can be plugged in and out dynamically. All this drives the development of flexible, high scalable middleware. An interesting category of such middleware supports the event-based paradigm. In this paper we focus on the scalability issues we have faced in the development of an event-based middleware called JEDI. In particular, we focus on improving performances of such middleware when the number and distribution of components grows. In order to evaluate design alternatives, we have taken a simulative approach that has allowed us to analyze the design alternatives before actually implementing them.

## 1 Introduction

Modern computing systems are more and more oriented to serve scenarios in which any device is provided with a computational capability and is able to interact with the other devices it gets close to [7]. In this context, the need for scalable middleware that supports such interaction is growing. Such a middleware has to allow easy reconfiguration and plug in of new components. Moreover, it has to enable anonymous and multicast communication in order to support scenarios where components do not know what other components are around and which of them would be interested in their messages.

The kind of middleware that at the moment seems to address these requirements is based on the event-based approach, where applications are structured in autonomous components that communicate by generating and receiving *event notifications*. A component usually generates an event notification when it wants to let the “external world” know that some relevant event occurred in its internal state<sup>1</sup>. The *event dispatcher* provided by the middleware propagates the event to any component that has declared its interest by issuing a *subscription*. A subscription, therefore, can be seen as a constraint on the content of the events; when an event respects the condition

---

<sup>11</sup> In the following we will use the terms event notification and event indifferently since, from the viewpoint of event-based middleware, we do not need to distinguish between the occurrence of an event and the generation of the corresponding notification.

stated by a subscription we say that the event is *compatible* with that subscription. The propagation of events is completely hidden to the components that generated them, thus the event dispatcher implements a multicasting mechanism that fully decouples event generators from event receivers. This provides two important effects. First, a component can operate in the system without being aware of the existence, number and location of other components. Second, it is always possible to plug a component in and out of the architecture without affecting the other components directly. These two effects guarantee a high compositionality and reconfigurability of a software system, and make event-based middleware particularly suited for the development of systems in which components operate autonomously and are loosely coupled. The underlying hypothesis is that an agreement exists between components on the structure of events. Several event-based platforms are currently available, either as research prototypes or as commercial products. [4] and [5] present the JEDI system and a taxonomy of some of the other existing platforms.

A main problem of all systems is scalability and performance. It can be noticed, in fact, that if the dispatcher has to manage every component subscription and notification, it can easily become a bottleneck for the whole system. To solve this problem in JEDI the event dispatcher is implemented as a distributed system composed of several *dispatching servers* organized in a hierarchy. Each of these servers shares with the others part of the received subscriptions and events in order to guarantee that connected components communicate properly.

In order to evaluate the performance and scalability of our solution and to identify possible improvements, we have taken a simulative approach. This allows us to determine the best alternative before actually implementing and deploying any possible solution. In this paper we compare through simulation the current implementation of JEDI with an alternative design, and we discuss advantages and disadvantages of both approaches. The rest of the paper is structured as follows. Section 2 presents an overview of the event-based middleware we are developing and of the alternative design we are facing with. Section 3 describes the simulation model of JEDI and Section 4 presents the results gathered from simulation. Section 5 discusses the related work and, finally, Section 6 provides some conclusions.

## 2 JEDI

JEDI stands for Java Event-based Distributed Infrastructure. It supports asynchronous and decoupled communication among distributed elements that are called *active objects* (AOs for short). Each active object interacts with other AOs by explicitly producing and consuming events. Event notifications in JEDI have a name and a number of parameters. For instance, `SoftwareReleased(Editor, 1.3, WinNT)` notifies that version 1.3 of a software called Editor has been released for WindowsNT. Subscriptions are syntactically similar to notifications; they have a name and some parameters. A subscription is *compatible* with every event that has the same values for the same fields. To allow the creation of more flexible subscriptions the operator “\*” has been introduced, representing a kind of wildcard. For instance, subscription `*(Editor, *, Win*)` is compatible with all the notifications (including the one above) having any name, three parameters, and concerning all the Editor versions that run on WindowsNT, Windows98, Windows2000, ...

## 2.1 Subscriptions and Event Propagation

In JEDI subscriptions and notifications are managed by a hierarchy of dispatching servers (DS). Whenever an AO issues a subscription, the DS connected to that AO stores such subscription in its internal tables and forwards it to its parent, which, in turn repeats the procedure until the subscription arrives to the root DS. When a notification is generated, the receiving dispatching server forwards it to all its AOs and descendant dispatching servers that previously sent compatible subscriptions. Moreover, the dispatching server sends the notification to its parent that, in turn, acts in a similar way (without sending the notification back). Therefore, a notification can reach each AO that has issued a compatible subscription, regardless of the AO position in the hierarchy. The system guarantees that causally related notifications are received by AOs in the same order in which they are generated.

The hierarchical approach is certainly more scalable of a centralized one. However it requires DSs to get coordinated by propagating subscriptions and notifications. Such coordination traffic has to be kept as limited as possible in order to ensure good level of performance. In the next section we propose an improvement of the approach presented above.

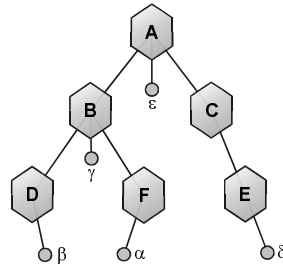
## 2.2 Improving the Event Propagation Algorithm: Advertisements

As discussed in the previous section, in the current version of JEDI, events emitted by AOs always reach the root of the dispatching hierarchy even if there is no subscriber that can be reached through the root DS. So, when the number of events emitted in the time unit grows, the DSs located at the higher levels of the hierarchy may become overloaded. To relief this situation we introduce a new event propagation algorithm based on the hypothesis that AOs perform a new operation called *advertisement*. AOs use advertisement to specify which kind of events they will produce. The rules that define the matching between events and advertisements are the same holding between events and subscriptions (see previous section). Moreover, we say that an advertisement is *compatible* with a subscription if at least one event compatible with both of them exists. AOs can dynamically issue advertisements and withdraw them during their life cycle.

Each DS uses advertisements and subscriptions to create proper routing tables that cause events to be propagated only through paths leading to interested subscribers. In the new algorithm we propose, advertisements are routed toward the root of the dispatching hierarchy exactly as it was illustrated for subscriptions in the previous section. When receiving an advertisement, a DS looks for compatible subscriptions that are then forwarded to the sub-tree that has generated the advertisement.

As an example, let us consider the hierarchy shown in Fig. 1 where hexagons represent dispatching servers, while circles represent AOs. Let us suppose that AO  $\beta$  issues a subscription. According to the algorithm previously described, such subscription is notified to dispatchers D, B, and A. Now suppose that  $\alpha$  issues an advertisement compatible with the subscription of  $\beta$ : F receives this advertisement and forwards it to B that detects the compatible subscription. Hence B propagates this subscription to F (notice that the subscription now is stored on every DS that connects  $\alpha$  to  $\beta$ ). Finally B propagates the advertisement to A that checks if the sub-tree rooted

at C has communicated other compatible subscriptions. If not, it simply stores the advertisement in its internal tables. Thanks to this subscription propagation approach, all the events generated by  $\alpha$  are routed to  $\beta$  through dispatcher B, and do not reach A unless a compatible subscription has been issued by any AO connected to A or to the subtree rooted by C.



**Fig. 1.** Connections between dispatching servers and agents

Of course, new subscriptions compatible with an advertisement can be issued even after an advertisement has been propagated (or during the propagation). A dispatcher that receives a subscription, while routing it toward the root of the hierarchy, checks in its internal tables for compatible advertisements, and, if any, routes the subscription toward the senders of these advertisements.

Summarizing, advertisements and subscriptions are forwarded up to the root. In addition, subscriptions are sent toward advertisements so that they can be used to create virtual paths between the event producers and consumers.

To further limit the traffic caused by propagation of subscriptions and advertisements, we have exploited some optimization techniques introduced in [2] and [3]. In these papers it is shown that, given a pair of advertisements (or subscriptions)  $x$  and  $y$ , it may happen that all the events compatible to  $y$  are compatible to  $x$  too. In this case we say that  $x$  covers  $y$ . Whenever an advertisement (subscription), issued in a subtree of the dispatching hierarchy, is covered by another advertisement (subscription) issued in the same subtree, such advertisement (subscription) does not need to be propagated toward the root since it does not provide any additional information.

### 3 The Simulation Model

In order to define a simulation model that properly describes the interesting characteristics of JEDI, we have collected numeric data on the behavior of two existing applications that have been developed on top of JEDI in the past years. The analysis of such mass of data has allowed us to identify the main components of the simulation model and their principal characteristics.

The main components of the simulation model describe dispatching servers and active objects. Dispatching servers are described in terms of the operations they can perform, (i.e., manage subscriptions, advertisements, notifications, ...) and the distribution of the time spent in performing these operations as they have been

determined from the measured data. Since the advertisement mechanism has not been developed so far in JEDI, the time needed to process advertisements has been estimated by considering that the algorithm that manages them is similar to the one used for subscriptions and exploits similar data structures.

Regardless of the application-dependent task they perform, AOs interact with the event-based middleware to issue subscriptions and advertisements, and to generate events. Based on this, for the purpose of our model, we have identified four elementary behaviors for AOs and we have associated them to proper simulation components called *agents*. Based on the behavior they embody, agents can be of four different types: *sinks*, *sources*, *proactive* and *reactive agents*. A source represents an AO that can only send events (and advertisements) during its life. A sink is able only to receive event notifications (and to send subscriptions). Proactive agents take the initiative by generating events and then wait for some reply events. Reactive agents wait for events and then generate new events.

In event-based middleware, inputs to the dispatching hierarchy (notifications, advertisements, and subscriptions) are correlated through the compatibility and covering relations defined in Section 2. This correlation captures the characteristics of the applications built on top of the middleware and it has an impact on the load of dispatching servers and on the network traffic. We have created a synthetic *correlation model* that, at simulation startup, automatically generates subscriptions, advertisements, and notifications to be assigned to each agent. This correlation model is based on few parameters that describe various application-dependent phenomena such as the distance covered by notifications in order to reach their subscribers, the coverage relationships defined on advertisements and subscriptions, etc.

The main parameter we have introduced is the *spreading coefficient* ( $sc$ ). It provides an indication of the distance covered by events in the dispatching hierarchy. This coefficient is defined in the  $[0, 1]$  interval and occurs in the following formula:

$$R(n) = sc^n \quad (1)$$

$R(n)$  is the probability that an event issued by an agent directly connected to a dispatcher A has to be received by some agent directly connected to a dispatcher at a distance  $n$  from A. The distance between dispatchers is calculated on the basis of the topology of the dispatching hierarchy. For instance, in the topology of Fig. 1 dispatchers A and D are at a distance 2. Intuitively, the more  $sc$  (spreading coefficient) is close to 1, the more it is likely that events have to be spread across the whole hierarchy of DSs. Conversely, when  $sc$  is low, events and corresponding subscriptions are mostly localized in some dispatching hierarchy sub-tree. During simulation, we use the spreading coefficient as an input parameter, in order to test the behavior of the event-based system in different event propagation conditions. To capture the effect of covering relations for subscriptions and advertisements, we have introduced other simulation parameters that for space reasons are not presented in this paper.

The environment we have selected to perform simulations is called OPNET [6]. In OPNET a model of the system to be simulated is defined by selecting components from proper libraries and by defining the way these components are connected together. OPNET provides a number of predefined libraries that model network components such as hubs, routers, and TCP/IP stacks. Using these libraries it is possible to easily define simulation models of local networks as well as WANs and

wireless systems with mobile objects. In addition, OPNET allows users to define their own libraries to model the behavior of specific application-dependent elements. We have exploited this feature to define our simulation model and we have relied on the existing libraries as for modeling the underlying network infrastructure.

Before starting a simulation, the simulation model is customized by assigning values to the following parameters:

- The characteristics of the underlying physical network in terms of connectors bandwidth, latency, protocols, and topology.
- The topology of the system, i.e. the structure of the event dispatching hierarchy and their location on the physical nodes.
- The spreading coefficient and the parameters associated to the covering relations.
- The number and location of connected agents and their types.
- For each agent, the mean values of the distributions defining their life cycle (number of subscriptions, notification frequency, ...).

## 4 Simulation Results

The goal of simulation has been to understand when the usage of advertisements is advantageous in term of performances of the most critical parts of the system (root DS and network channels) and what happens when the bandwidth of network connections between dispatching servers decreases.

In order to set up the simulation scenarios, we have first identified the operational conditions of our system. For instance, we have defined the maximum number of events manageable by a dispatching server, we have analyzed the relations between the number of physically different nodes of computation and the load on the LANs, we have observed the bandwidth consumed by notifications to define the proper channel sizes to interconnect the various LANs, etc.

During simulation we have assumed that dispatchers are organized in a quaternary balanced tree. We have considered trees having 5, 21, or 85 dispatching servers organized in trees of 2, 3, or 4 levels respectively. Each dispatching server manages 52 agents located on 4 hosts connected to the same LAN. The number of agents connected to a dispatching server has been determined on the basis of some preliminary simulations in order to avoid saturation of the root DS. In all simulation scenarios the 52 agents connected to a dispatching server are categorized as follows: 16 proactive agents, 24 reactive agents, 8 sources, and 4 sinks. Proactive agents and sources send events every 8 seconds on average. The duration of simulation has been selected so that the initial transitory can be disregarded. The LANs connecting a dispatching server to its agents are 10Mbit/sec Ethernet networks. These are connected together through 1Mb/sec or 64 kb/sec communication channels, depending on the scenario being considered. We have always adopted the TCP protocol.

Fig. 2 and Fig. 3 compare the performance of the root DS when event propagation is exclusively based on subscriptions and when subscriptions and advertisements are used together to define the event routing tables. We call these two cases *subscriptions-based* and *advertisements-based*, respectively. The figures show the average percentage of time spent by the root dispatching server (i.e., the potential

bottleneck of the system) in managing notifications (Fig. 2) and subscriptions (Fig. 3) plotted against the spreading coefficient introduced in Section 3. Intervals are drawn with a confidence level of 90%. The results are referred to a dispatching hierarchy of 21 dispatching servers connected through 1 Mb/sec links.

Fig. 2 shows that in the advertisements-based case the root DS spends much less time in handling notifications compared to subscriptions-based case. In fact, in the first case, the root DS has to handle only the events which actually need to be routed in subtrees different from their originators, while in the latter case, it handles all the events that are generated in the system. In the subscriptions-based case the growth in the spreading coefficient causes more events to be transmitted to other sub-trees thus resulting in more work on the side of the root dispatching server. Because of the growth in workload, in this case, the root dispatching server gets saturated for spreading coefficients higher than 0.4. In this case, therefore, the advantages of the advertisements-based approach against the subscriptions-based one is quite relevant. This result is also confirmed by the graphic of Fig. 3 showing that the advertisements-based approach does not result in an appreciable growth in processing time for handling subscriptions propagation for the root DS. We have also analyzed the processing time of advertisement, and we have noticed that this value is quite low (between 2.65% and 2.9%) and does not seem to be influenced by the spreading coefficient.

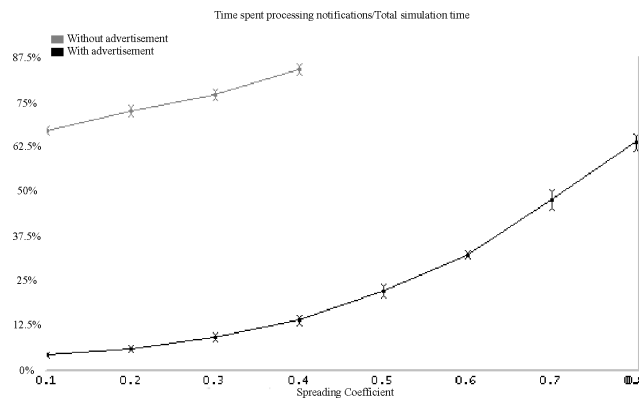
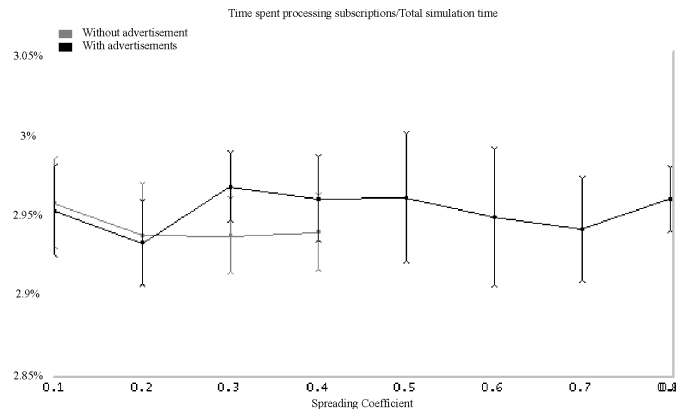


Fig. 2. Time spent by the root dispatching server in processing notifications

Similar simulations have been performed with dispatching hierarchies of 5 and 85 dispatchers. In the case of 85 dispatcher the subscriptions-based approach is not usable because, even for  $sc=0.1$  the root DS gets saturated. The case of 5 dispatchers shows that for a small system the two approaches produce very similar results, with a difference of at most 5% in favor of the advertisement-based case.

The advantages of the advertisements-based algorithm increase when the bandwidth of the communication channels dedicated to the traffic of the event-based infrastructure decreases. This case applies when the event based middleware is deployed across the boundaries of a single organizations. Fig. 4 shows the results of a set of simulations performed on a model with 21 dispatchers connected among each other through 64 kb/sec links. The graphics show the utilization of communication links connecting dispatching servers at different levels in the hierarchy. The graphics on the left-hand side refer to the traffic directed toward a dispatching server and its

controlled AOs (they reside on the same LAN), while those on the right-hand side refer to the traffic directed in the opposite direction. When the subscriptions-based approach is exploited, the 64 kbit/sec channels entering into the root DS get saturated for values of sc higher than 0.5. Conversely, the traffic outcoming from the root DS is more limited than in the advertisements-based approach. In this last case, in fact, subscriptions are sent downward to establish the routing between senders and subscribers.



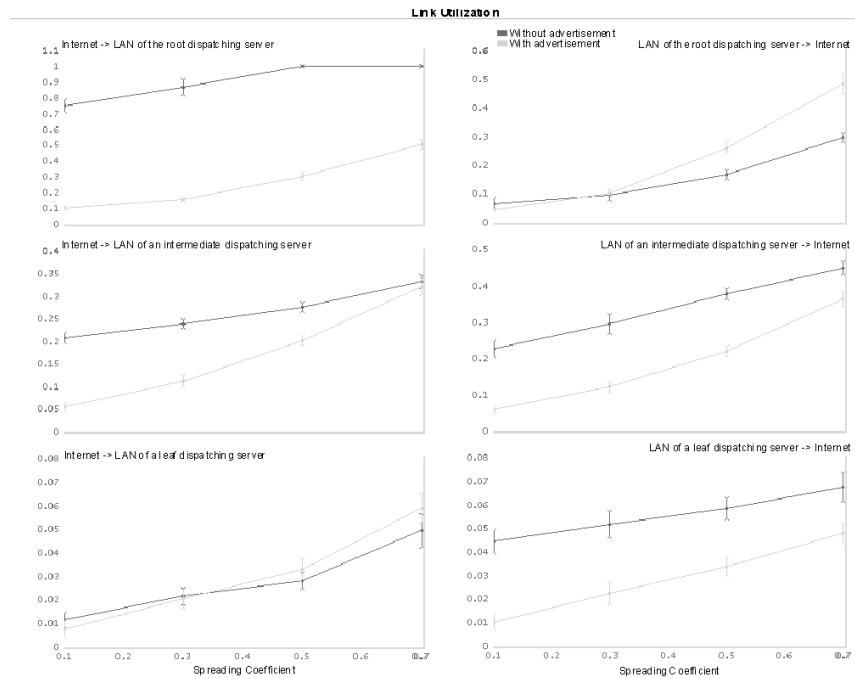
**Fig. 3.** Time spent by the root dispatching server in processing subscriptions

Concluding, in general the advertisements-based approach offers better performances in case of large systems where communications tend to be localized among groups of neighbor components. The advantages of the approach compared to the subscriptions-based approach are particularly evident when low speed communication channels are used. The subscriptions-based algorithm remains attractive for its simplicity in those cases where the dispatching infrastructure is not working under heavy load.

## 5 Related Work

While a number of researches and practitioners are focusing on the development of event based middleware, we know of few efforts devoted to understand performance and scalability of such middleware. A discussion of some preliminary requirements for scalable event-based middleware is presented in [7]. In particular, authors point out at the importance of having mechanisms for limiting network traffic among distributed dispatching servers located on wide-area network. With this respect, the advertisement algorithm we have developed for JEDI addresses, at least partially, some of these requirements. In the context of commercial systems, we are aware of two middleware, Smartsockets [8] and TIB/Rendezvous [9], providing a distributed implementation of the event dispatcher. In both cases, however, distribution seems to be exploited to achieve reliability on a small size system more than scalability due to the massive distribution of components and efficiency of event propagation.

The problem of providing mechanisms for efficient multicast of events is tackled in [1]. In this paper authors present a new algorithm to verify the compatibility of notifications with subscriptions. Differently from what we do, they assume that subscriptions are known in advance at any node of the dispatching server network, and do not focus on how they are actually propagated. Based on this information, they can establish optimized paths for event notifications. The performances of the proposed algorithm are evaluated through simulation.



**Fig. 4.** Utilization of links between LANs at various levels in the hierarchy

Our work has its premises in a previous work of Carzaniga et al. ([2] and [3]), where an advertisement approach is presented and simulation is used for the first time in the context of the evaluation of event-based middleware. Differently from them, we have focused our effort on hierarchical systems. Moreover our algorithm avoids that advertisements and subscriptions floods the network of DSs. In the simulation model we have introduced two additional types of agents (proactive and reactive) and we have defined a model of locality based on the spreading coefficient. Also, we have tuned the simulation parameters by analyzing existing applications and relied on existing and proved models of network devices and protocols provided by a commercial simulator.

## 6 Conclusions

We have shown that the development of a scalable event-based middleware such JEDI requires special care in the definition of the mechanisms that allow dispatching components to distribute events regardless to the physical location of their originators and subscribers. We have exploited simulation with the purpose of validating the design alternatives we have defined. In particular, we have shown that the advertisements-based approach works well when event traffic is localized, while the subscription-based approach provides acceptable performances when events have to be dispatched to components distributed all over the dispatching hierarchy, assuming that the root dispatcher has been properly dimensioned. Based on the above observations we argue that the choice of the event propagation model depends on the purpose and structure of the application that is going to exploit it. Therefore, event-based infrastructures should not bundle themselves on a specific approach. Instead, they should be designed in such a way that the application designer is free to choose the approach that better suits his/her needs.

We are currently consolidating the simulation model and validating it through a proper analytical model. We aim at defining some load balancing mechanisms that avoid or contrast saturation in dispatching servers. Finally, we are extending the semantics of our event-based middleware by introducing new operations such as the possibility of generating events expecting a reply from their receivers.

## 7 Acknowledgements

We are grateful to Antonio Carzaniga, Gianpaolo Cugola, and Prof. Giuseppe Serazzi.

## 8 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 '99 -- Int'l Conference on Distributed Computing Systems.
2. A. Carzaniga, "Architectures for an Event Notification Service Scalable to Wide-area Networks". PhD Thesis. Politecnico di Milano. December, 1998.
3. A. Carzaniga, D.S. Rosenblum, and A.L. Wolf, "Achieving Expressiveness and Scalability in an Internet-Scale Event Notification Service.". Nineteenth ACM Symposium on Principles of Distributed Computing (PODC2000), Portland OR. July, 2000.
4. G. Cugola, E. Di Nitto, and A. Fuggetta, "Exploiting an Event-Based Infrastructure to Develop Complex Distributed Systems". Proceedings of the 20th International Conference on Software Engineering (ICSE 98), Kyoto (Japan), April 1998.
5. G. Cugola, E. Di Nitto, and A. Fuggetta, "The JEDI event-based infrastructure and its application to the development of the OPSS WFMS". To appear on IEEE Transactions on Software Engineering.
6. MIL3 Inc., "OPNET MODELER" reference guides. Vol. 2,3 and 11.
7. B. Segall and D. Arnold, "Elvin has left the building: A publish/subscribe notification service with quencing". Proceedings of AUUG97, September 1997.
8. Talarian, "Mission Critical Interprocess Communications - an Introduction to Smartsockets", White paper.
9. TIBCO, "TIB/Rendezvous", White Paper. <http://www.rv.tibco.com/rvwhitepaper.html>.