Modelling Publish/Subscribe Communication Systems: Towards a Formal Approach*

R. Baldoni, M. Contenti, S. Tucci Piergiovanni and A. Virgillito
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113, 00198, Roma, Italy
email: {baldoni,contenti,tucci,virgi}@dis.uniroma1.it

Abstract

Publish/subscribe is a widespread communication paradigm for asynchronous messaging that naturally fits the decoupled nature of distributed systems, allowing simple and effective development of distributed applications. In this paper we propose a framework which includes a model of a publish/subscribe computation and a few properties on the computation, namely completeness and minimality, that capture, from an application viewpoint, the expected behavior of a publish/subscribe system with respect to the semantic of the notification of the information. Finally, we provide also a centralized implementation of publish/subscribe system which produces minimal and complete computations.

1 Introduction

In the last years, a growing attention has been paid to the publish/subscribe (pub/sub) communication paradigm as a means for disseminating information through distributed systems on wide-area networks. Participants to the communication can act as publishers, that submit information to the system, and subscribers, that express their interest only in specific types of information. A subscriber will then receive only published information matching its interest. Main advantages of a pub/sub communication paradigm with respect to a classical message passing are the following: the interacting parties do not need to know each other, partners do not need to be up at the same time, and the sending/receipt does not block participants. Therefore it provides a many to many communication facility and a full decoupling of the participants in space, time and flow that matches the loosely coupled nature of distributed applications.

*This work has been partially supported by a grant from EU IST Project "EU-PUBLIC.COM" (#IST-2001-35217)

Since pub/sub has been largely recognized as an effective approach for information diffusion, lots of pub/sub based systems, both research contributions [CRW01, BCM+99, SAB+00, PPPL+00] and commercial products [GKP99, OPSS93], has been presented and are actually used in several application contexts. From the research side, on one hand, a lot of work has been done in this field from the software engineering community, focusing on scalability, efficient information delivery or efficient and expressive information matching [ASS+99, OAA+00, CCC+01, FLPS00]. On the other hand, little has been done from the point of view of the distributed system community to define, for example, which are the communication semantics offered to users with respect to the notification of an information by a publish/subscribe communication system. The result is that such semantics are often "non-deterministic" and very difficult to compare. Let us note that "non-determinism" in publish/subscribe systems assume a twofold meaning: the first is related to the classical notion of sources of non-determinism present in a distributed system, such as failures, unpredictable message transfer delay, concurrent execution of processes [RST91]. The second one is a consequence of the decoupling in time typical of a publish/subscribe system. This decoupling can have two distinct forms:

1. an information item x can be published by a process while an intended receiver is not running;
2. an information item x can be published by a process while an intended receiver is running but is not interested to receive the information x yet.

In both cases, non-determinism can be reduced if x remains available within the publish/subscribe system for a fixed amount of time.

In this paper we propose a first attempt to model a publish/subscribe communication system as a classical distributed computation (publish/subscribe computation). This

model first introduces a definition of availability of an information x in a publish/subscribe computation, then it provides some properties (namely, completeness and minimality) on the semantics of the notification of an information x based on the notion of x's availability. Informally, completeness states that each available information is delivered to processes subscribed to that information. Minimality states that an information is delivered at each process at most once. These properties represents building blocks to formalize classes of QoS with respect to the underlying applications. The paper also presents two simple implementations of a publish/subscribe communication subsystem that produce minimal and complete computations. These implementations are both based on the notion of an intermediary which mediates the interaction among processes. The intermediary is presented first as a single process and then as a network of distributed servers.

The paper is structured as follows: Section 2 gives an overview of the related work in this area, Section 3 defines the general computational model. Section 4 defines the concepts and properties cited above, while in Section 5 the implementations are presented and discussed. Section 6 concludes the paper.

2 Related Work

Publish/subscribe communication systems can be classified in two main classes: topic-based (e.g. TIB/Rendezvous [OPSS93]) and content-based (e.g. SIENA [CRW01] and Gryphon [BCM$^+$99]). In a topic-based system, processes exchange information through a set of predefined subjects (topics) which represents many-to-many logical channels. Content-based systems are more flexible as subscriptions are not related to specific topic but to specific information content and each single information item actually can be seen as a single dynamic logical channel. In the following we first describe TIB/Rendezvous as an example of a topic-based publish subscribe system, then we present Gryphon and SIENA as two of the most relevant examples of content-based systems.

TIB/Rendezvous is a commercial product supporting a topic-based publish/subscribe interaction model. Processes are supposed decoupled in space but coupled in time, since information exchange guarantees are provided only for processes up at the same time. In other words, TIB/Rendezvous does not face the non-determinism due to point (1) (see Introduction) and reduces non-determinism due to point (2) by using an additional caching mechanism in which the last n messages for each subject are temporarily stored.

The SIENA project faces the design and implementation of a scalable general-purpose content-based system. The research efforts in SIENA has been focused in maximizing expressiveness in the selection mechanism offered to the end user without sacrificing scalability in the delivery mechanism. The result is a formalized structure of information and subscriptions. Moreover, an efficient matching algorithm and efficient mechanism for disseminating subscriptions within the distributed architecture have been proposed. Concerning the problem of non-determinism, SIENA does not address it nor it provides delivery guarantees stronger than “best effort”.

Similarly to SIENA, also in the Gryphon project efficient matching dissemination have been investigated. As far as the problem of non-determinism is concerned, Gryphon provides a persistent form of availability of information based on stable storage. Therefore, Gryphon actually eliminates the source of non-determinism proper of a publish/subscribe system.

3 A Framework for Publish/Subscribe

We consider a distributed system composed by a set of processes $\Pi = \{p_1, \ldots, p_n\}$ that communicate by exchanging information items. An information item belongs to a set denoted by Inf. To simplify the presentation, we assume the existence of a discrete global clock. This is a fictional device: the processes do not have access to it. We take the range $\mathcal{T}$ of the clock's ticks to be the set of natural numbers. In the following we present the behaviour of processes by detailing the operations that they can perform. By specifying the semantic of operations, we analyze the overall behaviour of the whole system.

3.1 Process Operations

Each process p_i can execute the following operations: $\texttt{publish}_i(x)$, $\texttt{notify}_i(x)$, $\texttt{subscribe}_i(\mathcal{C})$, $\texttt{unsubscribe}_i(\mathcal{C})$, where x is an *information item* and a $\mathcal{C}$ is a *subscription*. We drop the index of the process whenever not necessary. In our model an information item x is a conjunction of pairs (attribute, value). A subscription $\mathcal{C}$ is a predicate: if the information item x matches the subscription, then $\mathcal{C}(x) \equiv \top$, otherwise $\mathcal{C}(x) \equiv \bot$.

Each process p_i submits an information item x to other processes by executing the $\texttt{publish}_i(x)$ operation. p_i receives an information item x submitted by other processes by executing an upcall to $\texttt{notify}_i(x)$. A process p_j is notified only for information items that match its current subscription $\mathcal{C}_j$ ($\mathcal{C}_j(x) \equiv \top$). A precise specification of information and subscriptions syntax and matching semantics is out of the scope of this paper. Subscriptions are respectively installed and removed at each process by executing the `subscribe` and `unsubscribe` operations. Finally, for sake of simplicity, we assume that two `publish` operations never publish the same information.

3.2 Local Computation

Each process executes a sequence of operations. Each operation execution produces an event. Then, each process p_i produces a $sequence$ of events. This sequence is called *local computation* of process p_i and is denoted by h_i. We denote as e_i^s the event produced by process p_i at time $s \in \mathcal{T}$. In particular we denote the events produced by the execution of $\texttt{publish}_i(x)$, $\texttt{notify}_i(x)$, $\texttt{subscribe}_i(\mathcal{C})$, $\texttt{unsubscribe}_i(\mathcal{C})$ at process p_i as *$pub_i(x)$, $nfy_i(x)$, $sub_i(\mathcal{C})$, $usub_i(\mathcal{C})$* respectively. Not all the possible event patterns can belong to a local computation of a publish/subscribe system. In the following we give some local property that has to be satisfied by a publish/subscribe local computation.

When a process activates its subscription $\mathcal{C}$, producing a sub event at global time s, then it may or may not notify the information item that matches the subscription. When it wants to delete the subscription it produces a $usub$ event. For sake of simplicity we assume that a process producing a $usub$ event, has previously produced a sub event. Formally:

$\forall\ e_i^t = usub(\mathcal{C}) \Rightarrow \exists!\ e_i^s = sub(\mathcal{C})$ s.t. $s < t$ (LP1)

We also assume that after a subscription event sub, eventually there will be the corresponding unsubscription event $usub$ in h_i. Formally:

$\forall\ e_i^s = sub(\mathcal{C}) \in h_i \Rightarrow \exists!\ e_i^t = usub(\mathcal{C})$ s.t. $s < t$ (LP2).

Any two successive events $e_i^s, e_i^u \in h_i$ $(s < u)$, such that $e_i^s = sub(\mathcal{C})$ and $e_i^u = usub(\mathcal{C})$ define a *subscription interval* of p_i for a subscription $\mathcal{C}$, denoted by $S_i(\mathcal{C})$. Then it includes all events e_i^t s.t. $s \le t \le u$.

Without loss of generality we assume that each process can have only one active subscription at a time. This implies that there cannot be overlapped subscription intervals at each process. Formally:

$$\forall e_i^t \in S_i(\mathcal{C}) = [e_i^s, \ldots, e_i^u] \text{ s.t. } t \neq s \Rightarrow e_i^t \neq sub(\mathcal{C}') \text{ (LP3)}$$

Finally, in order to preserve the correct semantics of the `notify` operation, a notification event can be generated only after a subscription event that matches the notified information and before the corresponding unsubscribe event. (i.e. a notify event falls only in matching subscription intervals). Formally:

$$\forall e = nfy(x) \in h_i \Rightarrow nfy(x) \in S_i(\mathcal{C}) \text{ s.t. } \mathcal{C}(x) = \top \text{ (LP4)}$$

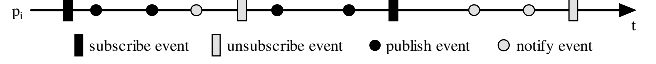

Figure 1. A local computation

Let us remark that a publish event can fall at any place of a local computation regardless of the subscription intervals.

Finally, let us introduce the notion of legal local computation. A local computation is legal if it satisfies (LP1), (LP2), (LP3), and (LP4). Figure 1 shows an example of a legal local computation.

4 Basic Properties of a Publish/Subscribe Communication System

In this section we provide the notion of information availability of an information item which is central in the context of a publish/subscribe communication system. Then we provide two global properties on a publish/subscribe distributed computation that express the guarantees on delivery of available published information to all matching subscribers. These properties could be used as basic building blocks for defining classes of QoS in a publish/subscribe system.

4.1 Information Availability

A process p_i publishes an information item x, producing a $pub_i(x)$ event, at a given time. We consider an available information item, one that can be delivered to matching subscribers. Obviously, an information item is *available* for interested subscribers only *after* it has been published. However, in a distributed system not all processes may have the same view of published information items, i.e. x is not available to all processes at the same time.

We model the availability of an information item by defining a boolean predicate $\text{AVAIL}_i(x)$ that is equal to $\top$ when the information item x is available for the process p_i. In this case, for the assumptions in Section 3.2, any subscription interval that matches the information item x has to contain the $nfy(x)$ event.

Delivery guarantees of a publish/subscribe system are strictly related to information *availability*. Ideally as soon as an information item is published it should be available for interested subscribers. The reactivity of the system to make an information item available is related to the underlying communication subsystem. To clarify this point, consider a trivial publish/subscribe system in which a publisher sends an information item to all other processes and then each process notifies the received item only if a current subscription matches it. In this case the information item is available to a subscriber when the subscriber receives it.

We define an *availability interval* for a process p_i, denoted $\mathcal{A}_i(x)$, as the time interval in which the published information item x is available at p_i. Before the publication of information item x, $\text{AVAIL}_i(x)$ is equal to $\perp$ at each process p_i. The availability interval starts at the time in which AVAIL_i switches from $\perp$ to $\top$. Once an information item x is available at a process p_i, and there is a subscription $\mathcal{C}$ such that $\mathcal{C}(x) \equiv \top$, a notify event can be produced at p_i. Let us remark that the same information x could be notified to a process p_i more than once. An availability interval can be upper bounded or unbounded. In the first case, the interval ends at process p_i when AVAIL_i switches from $\top$ to $\perp$ (see Figure 2). In the second case, all the published information has to be stored in a persistent way.

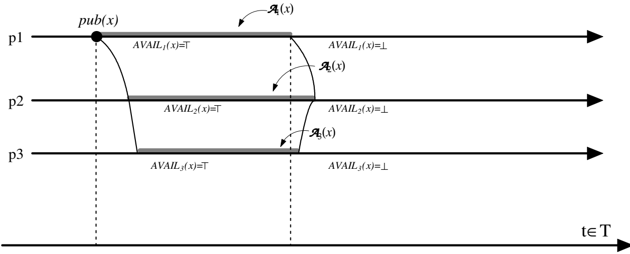

Figure 2. Availability intervals

4.2 Global Properties of a Publish/Subscribe Computation

In this Section we define two properties on a publish/subscribe computation that express reliability as guaranteed delivery of available published information to all matching subscribers. A publish/subscribe computation is the union of all local computations executed by processes that belongs to Π. Considering a single subscriber p_i, different levels of guarantees corresponds to how many times a notify event for a published information x appears in its local computation h_i within its subscription intervals matching x. Then, if *at least one* of such intervals contains $nfy(x)$, then we will say that the local computation of p_i is *complete*, as at least one subscription has been satisfied. If *at most one* of such intervals contains $nfy(x)$, then we will say that the local computation is *minimal*, as there is no more than one notification event for the same information item. Formally, we can define the following properties:

Definition 1 *A local computation h_i is* complete

iff $\forall x \; \exists nfy(x) \in (\mathcal{A}_i(x) \bigcap (S_i(\mathcal{C}) : \mathcal{C}(x) = \top))$

Definition 2 *A local computation h_i is* minimal

if $\forall x$ *s.t.* $\exists \; e_i^s = nfy(x), e_i^t = nfy'(x)$, ***then*** $nfy(x) = nfy'(x)$.

In the following figures we shows three examples of local computations. Figure 3 shows a local computation that is complete but non-minimal. Figure 4 shows a local computation that is minimal but non-complete. Figure 5 shows a local computation that is both minimal and complete.

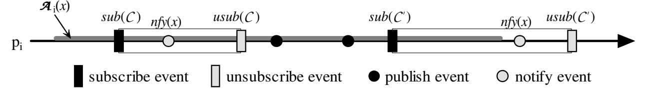

Figure 3. A complete and non-minimal local computation

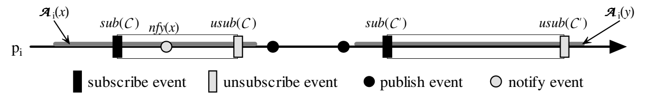

Figure 4. A minimal and non-complete local computation

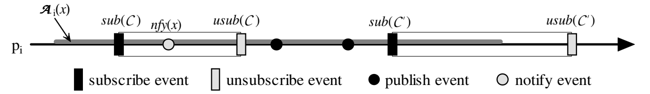

Figure 5. A minimal and complete local computation

We now extend these two properties to publish/subcribe computations, defining the following global properties:

Property 1 *A publish/subscribe computation is* complete *if h_i is complete at each process p_i.*

Property 2 *The publish/subscribe computation is* minimal *if h_i is minimal at each process p_i.*

A complete publish/subscribe computation ensures that each available published information item is surely notified to all the interested subscribers. However, it may be notified more than once. On the other hand, a minimal publish/subscribe computation ensures that each available published information item may be notified only to a subset of subscribers but each notified subscriber will be notified for a same information item only once. If both completeness

and minimality are satisfied, the computation ensures that each available information item will surely be notified to each interested subscriber exactly once.

4.3 Availability Classes

The implementation of a pub/sub system strongly depends on the criteria used for implementing the availability predicate. Obviously, the more an information item is available, the longer it is maintained in the system, i.e. the more memory is needed for storing information. On the other hand when the information item is no longer available, the pub/sub system can garbage-collect it. The different methods to "switch off" the AVAIL predicate drive different behaviors of the system that may be used to fit diverse types of applications. We now present three different classes of information, depending on "when" the AVAIL predicate is switched off. In particular, we consider an expiration time Δ for an information item x: x is no more available (and can be garbage-collected by the system) after a time Δ is elapsed from the time it is considered available. Then the three information classes, corresponding to three approaches in setting Δ, are:

0-availability : each published information item x expires as soon as it becomes available. Only processes whose subscription matches x at the very moment of its publication are guaranteed for notification. This implementation scheme is very low demanding since it does not require storing items. However, it is subjected to runs between concurrent pub and sub events, i.e. a subscriber may miss an information item x if its subscription is even slightly delayed with respect to the publication.

Δ-availability : each published information item x expires after $\Delta > 0$ from the instant it becomes available. Garbage collection is performed by the pub/sub system on all expired information items. This is more resilient to runs between the publisher and subscribers: a subscriber whose subscription "is seen" in the system after Δ since x is available, can be still satisfied. On the other hand high values for Δ may provoke undesirable out-of-date notifications.

∞-availability : each published information item x remains available in the pub/sub system for an indefinitely long time. When a subscriber installs a new subscription $\mathcal{C}$ it will receive all the previously published and already available information items that match $\mathcal{C}$. This implementation style obviously requires an ideal infinite memory to store all the information, since no information item is ever garbage-collected.

This classification can be related to the non-deterministic behavior deriving from the time decoupling between participants, that is one peculiar features of the pub/sub paradigm. In general, the more an information item remains available in the system, the less non-determinism is experienced (for example, the effect of runs between publications and subscriptions is limited). Reduction of non-determinism increases the probability that an intended receiver gets the information. If the information is stored in a persistent way, non-determinism is completely removed and this probability goes to one [TvS02]. Of course, this impacts on the size of the memory necessary within the system. Then, we can say that the three information classes feature decreasing levels of non-determinism on the other hand requiring increasing memory. However, we point out that this is not the case where a single class can be identified in absolute as better than the others, whereas each class is suited to meet different application requirements. Hence we give examples of applications that may belong to each class.

An example of an application based on 0-available information is a stock exchange system. Information items represent instant values of stock quotes, expiring very quickly. Since the publication rate is high, missing notifications due to runs poses no problem, since subscribers can get a new information after a short time. An example of an application based on Δ-available information is a daily news diffusion system. Each information item represents the daily issue of the news, having a lifetime of 1 day. Suppose issues are published every morning at 3 A.M., an issue will be notified also to processes submitting their subscription during the day. An example of an application based on ∞-available information is a digital library where catalog updates are published as information items, kept available for future subscribers. A new subscriber will receive all the previous notifications in order to build its local copy of the catalog.

5 Implementation Issues

The framework presented in previous sections abstracts the execution of a general distributed system following the pub/sub paradigm. The typical implementation model considers a logically centralized engine which is responsible for gathering the publications from publishers and forward them to interested subscribers. We present two different implementations of the engine, a centralized one and a sketch of a distributed one, comparing how availability is managed and the requirements for the completeness and minimality properties to be satisfied.

We first extend the model given in Section 3. The centralized implementation model includes a central entity $\mathcal{I}$ which we will refer to as the intermediary. The intermediary is composed by a set of processes $S_1, \ldots, S_n$, which

we assume not to crash. This simplifying assumption will allow us in the context of this paper to concentrate on the basic behavior of the system. All processes interacts by exchanging messages over an asynchronous distributed system. Communication is reliable and FIFO: messages sent by a process to a destination are eventually delivered to the destination (message transfer delay is unpredictable but finite) in their sending order. We assume processes do not fail and each process takes a finite time to execute a step.

5.1 The Intermediary

The role of the intermediary $\mathcal{I}$ is to propagate asynchronously the information sent by publishers only to the interested subscribers. The intermediary is accessed by processes through operations defined in its interface: `subscribe(`$\mathcal{C}$`)`, `unsubscribe(`$\mathcal{C}$`)` and `publish(`x, Δ`)`. A process invokes the operation `subscribe(`$\mathcal{C}$`)` (`unsubscribe(`$\mathcal{C}$`)`) to express (revoke) its interests in receiving information items matching the subscription $\mathcal{C}$. A process invokes the intermediary's `publish(`x, Δ`)` operation to transmit the information item x, with availability interval Δ. Δ is provided by the publishing process as a parameter to the `publish` operation to allow an application to define the desired availability for published information. $\mathcal{I}$ transmits the information by invoking the `notify(`x`)` operation that must be defined in the interface of the processes (Fig. 6).

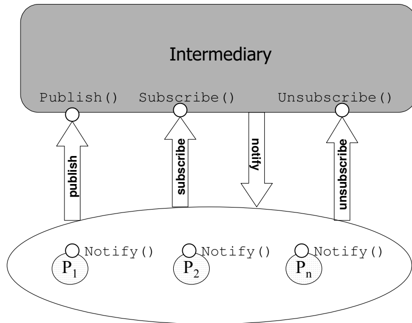

Figure 6. Publish/Subscribe Centralized Implementation Model

5.2 Intermediary as a single process

In the following we consider a basic implementation of the intermediary as a single process S_1. Let us remark that, despite its simplicity, this centralized implementation has been exploited by some publish/subscribe system (e.g. [SAB$^+$00]). Figures 7 and 8 show the pseudo-code of operations performed at a process and at the intermediary. In

```
INITIALIZATION
1  boolean subscribed = ⊥;

PROCEDURE SUBSCRIBE(subscription C)
1  send message SUB(C) to I;
2  wait until receive ACK;
3  subscribed = ⊤; %event sub(C)%

PROCEDURE UNSUBSCRIBE(subscription C)
1  send message UNSUB(C) to I;
2  wait until receive ACK;
3  subscribed = ⊥; %event usub(C)%

UPON RECEIVING NFY(x)
1  if (subscribed)
2    then NOTIFY(x); %event nfy(x)%

PROCEDURE PUBLISH(information x, Δ)
1  send message PUB(x) to I;    %event pub(x)%
```

Figure 7. A generic process

the following we give an explanation of the pseudo-code, showing that it produces only complete and minimal computations.

Generic Process The pseudo-code in Figure 7 highlights the points when an event is generated in the local computation of a process p subsequently to an operation invocation in p. The code produces only legal computations at each process. In particular, the SUB and UNSUB messages are acknowledged by the intermediary in order to avoid the possibility of a notify event occurring out of a matching subscription interval. PUB operations does not require acknowledgement because legality properties does not imposes any constraint on *pub* events within a computation.

Intermediary The intermediary maintains the following data structures:

- $\mathcal{M}$: a set of couples $(\mathcal{C}, p_i)$ where $\mathcal{C}$ represents the current subscription for p_i.
- Inf: a set of couples (x, t) where x is an information item expiring at time t (local time of the intermediary).

The pseudo-code is structured in four clauses activated by the receipt of a message or when a predicate is verified. The clauses are executed within a same program, then their execution is mutually exclusive. The criteria for managing the AVAIL$_i$ predicate is the one presented in previous section: an information item is kept available for all processes and stored in Inf for a time Δ since it has been received. Referring to the model, this can be formally expressed saying that AVAIL$_i(x)$ is $\top$ for all i if and only if $\exists (x,t) \in Inf$. (x, t) is stored in Inf when the publication message is received by $\mathcal{I}$ (Line 14) and removed when the predicate stating its expiration is verified (Line 20).

INITIALIZATION
1 $Set\,of\ (\mathcal{C}, p_i)\ \mathcal{M} := \emptyset$;
2 $Set\,of\ (x, t)\ Inf := \emptyset$;

BEGIN
1 **while** $true$ **do**
2 **when** $(receiving$ SUB$(\mathcal{C}) from\ p_i)$ **do**
3 $\mathcal{M} := \mathcal{M} \cup \{(\mathcal{C}, p_i)\}$;
4 $send$ ACK $to\ p_i$;
5 **for each** $(x, t) \in Inf$
6 **if** $(\mathcal{C}(x) = \top\ \wedge x\ never\ been\ notified\ to\ p_i)$
7 **then** $send$ NFY(x) $to\ p_i$;
8
9 **when** $(receiving$ UNSUB$(\mathcal{C}) from\ p_i)$ **do**
10 $\mathcal{M} := \mathcal{M}\ - \{(\mathcal{C}, p_i)\}$;
11 $send$ ACK $to\ p_i$;
12
13 **when** $(receiving$ PUB$(x, \Delta))$ **do**
14 $Inf\ = Inf\ \cup \{(x, curr_time + \Delta)\}$; $\%AVAIL_i(x){=}\top\ \forall p_i\%$
15 **for each** $(\mathcal{C}, p) \in \mathcal{M}$
16 **if** $(\mathcal{C}(x) = \top)$
17 **then** $send$ NFY(x) $to\ p$;
18
19 **when** $((\exists (x, t) \in Inf : t < curr_time))$ **do**
20 $Inf\ = Inf\ - \{(x, t)\}$ $\%AVAIL_i(x){=}\bot\ \forall p_i\%$

Figure 8. Intermediary

Computations produced at each process are complete: when $\mathcal{I}$ receives a publication, it checks for all the processes having a matching subscription in $\mathcal{M}$ and sends them a notification (Line 15). Moreover, when $\mathcal{I}$ receives a new subscription from a process p, it checks for all matching available information items in Inf and notifies them to the new subscriber (Lines 5-7). Also the minimality property is satisfied by local computations: $\mathcal{I}$ also checks for each information item in Inf if it has been previously notified to p, to avoid a process to receive a same information item more than once (Line 6). This can be implemented by a process-side filtering of information items or by a list maintaining for each available information item the set of processes it has already been sent to.

5.3 Intermediary as a Network of Servers

In general the intermediary can be implemented as a set of distributed server processes $S_1, \ldots, S_n$, where each server S_i has the role of accepting subscriptions and notifications and forwarding them within the system. This approach is inherently more scalable and robust than the centralized one, as we will discuss later in this this section, thus it is adopted by most pub/sub systems. A process p_i requests to publish an information item or to subscribe for a topic only to a single server. Each server then exposes the same interface of the single-process intermediary presented in previous section and maintains a state related only to subscriptions it manages.

The main issue to be addressed in this case is information diffusion. This can be performed by exploiting network level facilities such as IP Multicast [OAA+00] or by constructing a network of application levels connections among servers (overlay network [ZZJ+01, ZKJ01]). The latter solution provides higher flexibility and is then considered more feasible for wide area environments: each server can maintain connections only with a subset of the other servers in the system, obtaining a lowest resource consumption and higher adaptability to dynamic conditions. In the context of this paper we consider an architecture based on an overlay network. Each server has to maintain all the active subscriptions of processes registered with it and all the connections with other neighbors severs. Connected servers form a network whose topology does not follow any particular structure (e.g. a hierarchy such as in TIBCO).

In the following we give a sketch of an algorithm for information item diffusion, based on the construction of a spanning tree over the server network rooted at the server that has received the notification request. Tree construction is performed by flooding information items from the source node throughout the network. When a server receives a notification request for an information item x from a client or from another server it adds x to the set of available information items and forwards it to all its neighbors. If a server receives a same request for x more than once it discards it. x is then notified to all the processes that have a matching subscription. When x expires, the server removes it from the set of available information items.

This algorithm can be subject to various optimizations aimed at reducing the number of messages sent. For example, in topic-based systems the tree can be built in advance [RKCD01], because once a process is subscribed to a topic it will receive all the messages related to that topic. This is not possible in content-based systems, where receivers are calculated on a per-message basis. One possible optimization in such systems is the one proposed in SIENA, where subscriptions are diffused in order to prune messages before they reach parts of network with no interested subscribers. For what concerns availability, in this case an information item x is available at a process p_i when it is received by the server in which p_i is registered. Obviously in this case, a same information item will be available at different times at each process and it is possible for a server to receive an information item long after its publication. Exploiting semantic dependence between information items allows to identify out-of-date items and to prune them out in order to save bandwidth [ORO00].

The completeness property is easily satisfied by our algorithm, since, under the simplifying conditions we assumed, it is sufficient that each information item eventually reaches all the servers. On the contrary, in presence of faults, an acknowledgement mechanism is required that highly increases algorithm complexity and network traffic. In this scenario, the minimality property assumes a partic-

ular relevance in reducing the network traffic, by avoiding as much as possible sending an information item more than once. However, even in our basic distributed implementation, ensuring minimality is not as trivial as in the single-process case: let us assume, as in the single-process intermediary, that a Δ-available information item x is garbage-collected by a server S_i after a time Δ from its reception. Following the presented diffusion algorithm, x can be received again by S_i after it has expired but cannot be recognized as "old" information. Then, in this distributed implementation,the garbage collection mechanism must maintain also expired information items, but in an asynchronous environment it is not possible to decide for how long.

If channels are FIFO, it is sufficient to maintain at each server a data structure that stores the sequence number of the last information item received from each server. All obsolete information items can be recognized and pruned. In the general case when messages can be received out of order this solution can lead to incorrect pruning of information items that have not yet become available, violating the completeness property. Then more sophisticated, and resource-demanding, algorithms are required.

6 Conclusions and Future Work

In this paper we defined a formal computational model for publish/subscribe systems that considers communication between processes at an abstract level. In our approach, the formalization of the concept of information availability plays a central role in defining the exact behaviour in terms of information items actually delivered to subscribers and in the representation of non-deterministic aspects of the paradigm. Two sample implementations have been presented, discussing the protocol-level requirements for managing availability and providing basic QoS properties, under simplified conditions. In future work we will set the model in more realistic environments, by designing scalable and fault-tolerant protocols for message diffusion in large-scale asynchronous systems.

References

[ASS+99] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching Events in a Content-Based Subscription System. In *Proceedings of The Symposium on Principles of Distributed Computing*, pages 53–61, 1999.

[BCM+99] 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 International Conference on Distributed Computing Systems*, 1999.

[CCC+01] A. Campailla, S. Chaki, E. M. Clarke, S. Jha, and H. Veith. Efficient filtering in publish-subscribe systems using binary decision. In *Proceedings of The International Conference on Software Engineering*, pages 443–452, 2001.

[CRW01] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and Evaluation of a Wide-Area Notification Service. *ACM Transactions on Computer Systems*, 3(19):332–383, Aug 2001.

[EFGK01] P.Th. Eugster, P. Felber, R. Guerraoui, and A.M. Kermarrec. The Many Faces of Publish/Subscribe. Technical Report ID:2000104, EPFL, DSC, Jan 2001.

[FLPS00] F. Fabret, F. Llirbat, J. Pereira, and D. Shasha. Efficient matching for content-based publish/subscribe systems. Technical report, INRIA, 2000.

[GKP99] R. E. Gruber, B. Krishnamurthy, and E. Panagosf. The architecture of the READY event notification service. In *Proceedings of The International Conference on Distributed Computing Systems, Workshop on Middleware*, Austin, Texas, 1999.

[OAA+00] L. Opyrchal, M. Astley, J. S. Auerbach, G. Banavar, R. E. Strom, and D. C. Sturman. Exploiting IP multicast in content-based publish-subscribe systems. In *Middleware*, pages 185–207, 2000.

[OPSS93] B. Oki, M. Pfluegel, A. Siegel, and D. Skeen. The Information Bus - An Architecture for Extensive Distributed Systems. In *Proceedings of the 1993 ACM Symposium on Operating Systems Principles*, December 1993.

[ORO00] J. Orlando, L. Rodrigues, and R. Oliveira. Semantically reliable multicast protocols. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS 2000)*, Oct. 2000.

[PPPL+00] R. Preotiuc-Pietro, J. Pereira, F. Llirbat, F. Fabret, K. Ross, and D. Shasha. Publish/Subscribe on the Web at Extreme Speed. In *Proc. of ACM SIGMOD Conf. on Management of Data*, Cairo, Egypt, 2000.

[RKCD01] A. I. T. Rowstron, A. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Networked Group Communication*, pages 30–43, 2001.

[RST91] M. Raynal, A. Schiper, and S. Toueg. On the Causal Ordering Abstraction and a Simple Way to Implement it. *Information Processing Letters*, 12(39):343–350, 1991.

[SAB+00] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content Based Routing with Elvin4. In *Proceedings of AUUG2K, Canberra, Australia*, June 2000.

[TvS02] A. Tanenbaum and S. van Steen. *Distributed Systems: Principles and Paradigms*. Prentice-Hall, 2002.

[ZKJ01] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, 2001.

[ZZJ+01] S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant widearea data dissemination, 2001.