Tradeoffs in Event Systems*

P. Th. Eugster[†1], P. Felber[2], R. Guerraoui[3], S. B. Handurukande[‡3],
[1]Distributed Computing and Systems Research Group,
Chalmers University of Technology, Göteborg, S-412 96 Sweden.
[2] Institut EURECOM, 2229 route des Crêtes, BP 193
06904 Sophia Antipolis, France.
[3] Distributed Programming Laboratory,
Swiss Federal Institute of Technology in Lausanne.

Abstract

This paper addresses fundamental tradeoffs in event systems between scalability *(in terms of event filtering, routing, and delivery),* expressiveness *(when describing interests in events), and* event safety *(ensuring encapsulation of polymorphic events and type-safe handling of these). We point out some ramifications underlying these tradeoffs and we propose a pragmatic approach to handle them. We achieve scalability using a multi-stage filtering strategy that combines approximate and perfect matching techniques for the purpose of event routing and filtering. We achieve expressiveness and event safety by representing events as instances of application-defined abstract types and defining subscriptions in terms of predicates expressed on these types.*

1 Introduction

The design and implementation of event systems has been an active field of research over the last few years. These systems have evolved from simple multicast-oriented topic-based systems (e.g., [Cor99, Ske98, TIB99, AEM99]) to elaborate, content-based, systems that filter and disseminate data events according to their content (e.g., [SAB+00, CRW00, SBCea98, CNF98, MÖ1, SCG01]). Content-based dissemination techniques permit accurate addressing of events to selected subscribers according to their interests.

Event systems do however face fundamental tradeoffs while attempting to satisfy several demands made on them. First, these systems must *scale* to a potentially large number of subscribers (hundreds of thousands), subscriptions (millions), and events (hundreds per second). Second, they should provide *expressive* mechanisms to precisely specify the interests of subscribers, in order to avoid receiving irrelevant events. This is especially important when subscribers have low bandwidths or expensive connections, as in the case of wireless phones and pagers. Third, the event model must be *safe* enough to permit exchange of arbitrary information encapsulated within application-defined types, i.e., without revealing implementation details (preserving encapsulation) or requiring explicit marshaling and unmarshaling (enforcing type safety) of this information.

Until now, event systems have been focusing only on parts of the equation, such as scalability and expressiveness [CRW00], and we are not aware of any system that provides (even partial) support for all three aspects, namely *filtering scalability*, *subscription expressiveness*, and *event safety*. In fact, the tradeoffs are a consequence of an underlying conflict that prevents filtering techniques from scaling without reducing subscription expressiveness or violating encapsulation.

This paper proposes a way to pragmatically combine the benefits of (1) a highly-scalable filtering technique, (2) an expressive subscription language and (3) a generic, yet safe, event representation. Event safety is enforced in the sense that events are defined as objects which are instances of application-defined types, and subscription expressiveness is obtained by supporting subscriptions based on any public member of these types. Scalability is achieved using a multiple stage filtering approach, where events are pre-filtered using elaborate information retrieval techniques. While the use of these techniques generally has the undesirable consequence of breaking event encapsulation, we circumvent this problem by performing

*This paper is an extended and revised version of our paper with the title "Event Systems: How to Have Your Cake and Eat It Too" in the proceedings of the IEEE International Workshop on Distributed Event-Based Sytems, 2002.

†The work of this author is sponsered by the Swiss National Science Foundation.

‡The work of this author is sponsered by Microsoft Research, Cambridge.

approximate filtering on the intermediate stages and preserving subscription expressiveness and type safety on an end-to-end basis.

In short, the contribution of this paper is twofold. First, we explicitly pose the inherent tradeoff in event systems. Second, we propose a pragmatic way to handle this tradeoff. We also provide a quantitative evaluation of our pragmatic approach by comparing the filtering complexity of our approach with a simple existing scheme.

The rest of the paper is organized as follows. Section 2 describes the tradeoffs involving event safety, expressiveness and scalability. Section 3 introduces the idea underlying our multi-stage filtering approach and Section 4 presents an architecture for putting our approach to work. Section 5 gives some simulation results for a large scale setting.

2 Tradeoffs

In this section, we first discuss in more detail the three desirable properties of events systems, namely *filtering scalability*, *subscription expressiveness*, and *event safety*, before pointing out inherent tradeoffs between these properties.

2.1 Desirable Properties

We start by the property of event safety, in order to underline the impact of this often neglected constraint on the two other closely studied properties, filtering scalability and subscription expressiveness.

Event Safety. Most event systems make few or no assumption on the nature of the events or their format because of the loose coupling between the publishers and subscribers. As a consequence, these systems usually provide low-level message-passing abstractions.

As discussed in [EGD01], working with typed events (and objects in particular) in event systems offers a number of advantages over unstructured and untyped events. When events are objects (encompassing implicit type inclusion information, a "value", and some behaviour), type hierarchies additionally permit the filtering of these events according to their polymorphic nature. In other terms, events can be filtered according to their conformance to types, including "content-based" queries expressed on any public members of these event types. Subscribers can register their interest to some event type (including all its subtypes), and if encapsulation and type safety are guaranteed, publishers can easily extend the hierarchy and create new event (sub)types without requiring subscribers to update their subscriptions.

Filtering Scalability. Most content-based event systems use an indirect form of addressing where selectivity is controlled by subscribers. The content of the message acts as a destination address that must be matched against the subscriptions ("group" addresses) of subscribers.

One can imagine different alternative architectures for content-based event systems.

- The first one relies on a centralized server (e.g., Elvin3 [SAB+00]) for filtering events and forwarding those of interest to the appropriate subscribers. The major drawback of the "centralized" approach is that the server is a bottleneck both in terms of processing power and network bandwidth, in addition to being a single point of failure.

- The second architecture of interest, used by event systems based on group communication, consists in broadcasting events to all subscribers and letting the each filter out events that do not match its local subscriptions at runtime. This "broadcast" approach, attractive because of its fully distributed nature, but does not scale well when the number of publishers and the message frequency increase.

- The last and most scalable approach relies on a set of networked nodes (e.g., [CRW00], also known as *overlay network*) for content distribution. Publishers and subscribers are connected to a local node (or a server) that is responsible for forwarding events in the system and delivering it to the local subscribers. Nodes are also in charge of storing events for temporarily disconnected subscribers with durable subscriptions.

Overlay networks are a key for scalability in content-based event systems because the resource-consuming tasks can be split among all the nodes of the network. Each node is responsible for only a subset of all subscribers and filtering can be performed in a distributed manner. An example is shown in Figure 1 The architecture of the network nodes (e.g., hierarchical, peer-to-peer) and the techniques employed to filter and route events are also key factors in scaling to a large number of subscribers, subscriptions, and event types and instances.

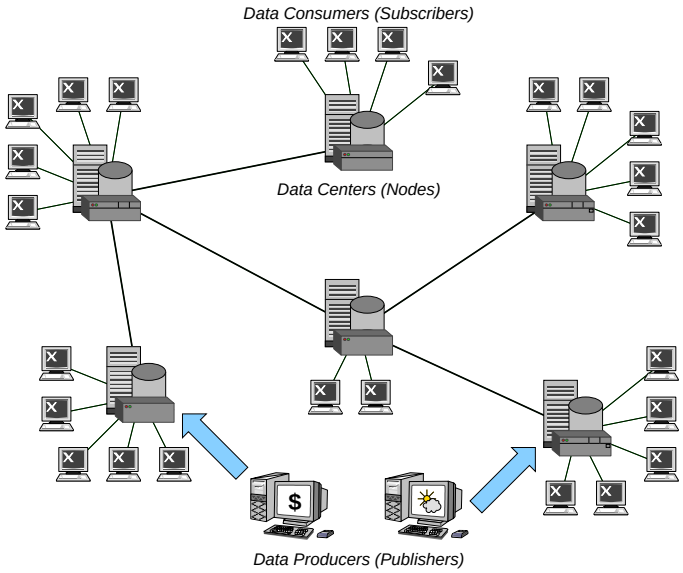

Figure 1: An overlay network: Publishers and subscribers are connected using a set of brokers.

Subscription Expressiveness. The expressiveness of subscriptions defines how accurately subscriptions can represent the interests of the subscribers. One can imagine different "levels" of expressiveness:

- The simplest form of subscription languages only permit string matching.
- More advanced subscription languages use filters [SAB+00, Car98, MÖ1] in the form of a set of attributes and constraints on the values of those attributes, where constraints are specified using common equality and ordering relations ($=$, $\neq$, $<$, $>$, etc.), as well as regular expressions.
- A further step in subscription expressiveness is to allow events to be filtered according to their type [EGD01]. Type-based filtering adds a new dimension to content-based event systems, by letting subscribers register their interests both to the nature and the value of published events.

Advanced subscription languages are highly desirable, because subscribers can more accurately express their interests. As expressiveness increases, so does selectivity and less irrelevant events have to be delivered to subscribers.

2.2 The Conflicts

Event safety is a property of event representation, scalability depends on the system architecture, and expressiveness relates to the subscription language. These three aspects, despite what might appear in Figure 2, are not orthogonal: desirable characteristics of one aspect may have unwanted effect on the other aspects. These conflicts are highlighted in the rest of this section.

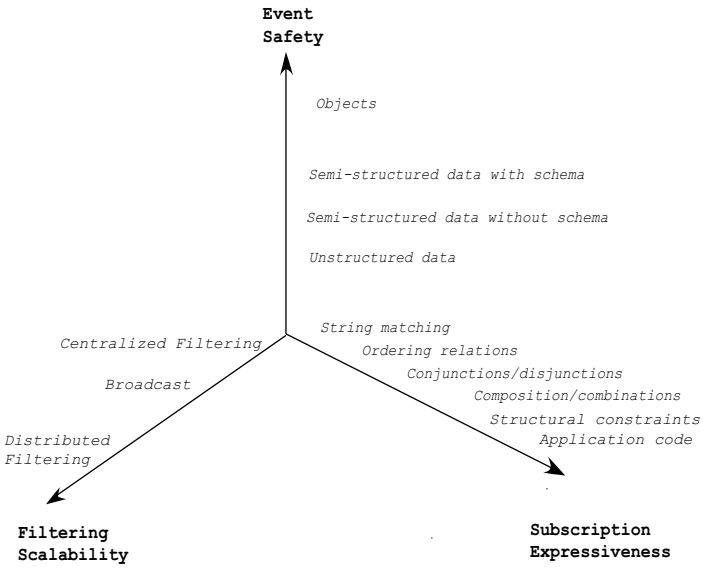

Figure 2: Three Aspects of Event Systems.

Event Safety vs. Filtering Scalability. The use of typed information adds some overhead to event filtering. This overhead is generally small when events are not objects, since it only includes the cost of type verification and polymorphic data handling. When events are objects (with their own behavior) and data is in principle accessible only through the object access methods, each object might have to be unmarshaled and instantiated in the runtime execution environment before filtering. When such a scheme is naively applied at each filtering step, scalability and performance decrease strongly.

In general, as event representation becomes more powerful and type-safe (becoming even part of the application design), the subscription language should also become more expressive, which might however jeopardize scalability.

Expressiveness vs. Filtering Scalability. Highly-expressive subscription languages allow subscribers to accurately specify their interest. In some respect, this can increase scalability by limiting the

number of irrelevant events delivered to subscribers. On the other hand, expressiveness increases filtering complexity and processing time. When dealing with real-time events, large numbers of subscriptions and high emission frequencies, filtering time must be kept as small as possible.

Event Safety vs. Expressiveness. Giving the application the possibility of defining own event types makes it difficult to ensure type safety and encapsulation of these events when describing subscriptions. Indeed, to ensure encapsulation and for expressiveness, a subscription should be able to involve methods defined by the type subscribed to, which is difficult to achieve "reasonably" in a programming language in a way ensuring static type safety. Given the fact that, for a reasonable performance in filtering and routing of events, the publish/subscribe engine has to be given an insight into subscriptions (e.g., for collapsing subscriptions), describing subscriptions by implementing typed filter objects is clearly unappealing. Language extensions [HMN+00, Eug01], or at least powerful language mechanisms, such as advanced reflection and genericity mechanisms are required [Eug01] to achieve satisfactory event safety and expressiveness.

3 Multi-Stage Filtering

In this section, we present a pragmatic approach for the provision of scalability with a type-safe event representation and an expressive subscription language. Our approach is based on multi-stage filtering. We first define some notions related to filtering before describing our approach.

3.1 Definitions

Publishers and subscribers communicate by exchanging events. A subscriber subscribes to specific events by registering a filter that is applied to incoming events: A subscriber only receives the events that matches its filter(s).

Definition 1 (Filter). *Consider a language $\mathcal{L}_E$ for representing events, and a language $\mathcal{L}_F$ for specifying filters. A filter is a function $f \in \mathcal{L}_F : \mathcal{L}_E \rightarrow \{true, false\}$ such that $f(e) = true$ if and only if event e matches the filter f.*

A filter corresponds to a subscription of a subscriber. An event is forwarded to a subscriber when at least one of its subscriptions returns *true*, and discarded otherwise. The filter f_T defined by "$\forall e \in \mathcal{L}_E \;\; f_T(e) = true$" expresses interest in all events, while the filter f_F defined by "$\forall e \in \mathcal{L}_E \;\; f_F(e) = false$" discards all events.

Example 1. Consider the following events describing stock quotes (events are represented by name-value tuples):

e_1 = (symbol,"Foo") (price, 10.0) (volume, 32300)
e_2 = (symbol, "Bar") (price, 15.0) (volume, 25600)

A filter selecting only the stock quotes for symbol "Foo" with price higher than \$5 can be defined as follows (filters are represented by name-value-operator tuples):

f = (symbol, "Foo", =) (price, 5.0, >)

Applying filter f to events e_1 and e_2 yields the following results:

$$f(e_1) = true$$
$$f(e_2) = false$$

We now introduce a covering relation for filters.

Definition 2 (Filter Covering). *A filter f covers another filter f' ($f \sqsupseteq f'$) if and only if the following property holds:*

$$f \sqsupseteq f' \quad \Leftrightarrow \quad \forall e \in \mathcal{L}_E \;\; f'(e) = true \Rightarrow f(e) = true$$

Informally, this means that f' is a more restrictive (or *stronger*) filter than f. The f_T filter covers all filters and the f_F filter is covered by all filters.

Example 2. The following filters cover filter f of Example 1:

f' = (symbol, "Foo", =)
f'' = ($price$, 5.0, >)
f''' = (symbol, "Foo", =) (price, 4.5, >=)

We also define a covering relation on event.

Definition 3 (Event Covering). *An event e covers another event e' for filter f ($e \overset{f}{\sqsupseteq} e'$) if and only if the following property holds:*

$$e \overset{f}{\sqsupseteq} e' \quad \Leftrightarrow \quad f(e') = true \Rightarrow f(e) = true$$

Informally, this means that e can be filtered more accurately than (or as well as) e' by filter f, and e is therefore a more accurate representation of the event.

Example 3. The following event covers event e_1 of Example 1 for filter f:

e'_1 = (symbol, "Foo") (price, 10.0)

Note that the event covering relation is bound to a filter. This is necessary because filters can define complex expressions, such as predicates on the presence of some data in an event. For instance, with filter "$(volume, \exists)$" that checks for the existence of an attribute named "volume", event e'_1 of Example 3 does not cover event e_1 of Example 1.

3.2 Scalability through Pre-filtering

An important factor for scaling to a large number of subscriptions and events is to limit the amount of events that transit between the nodes of the network. In particular, if a node has no subscriber interested in a given event, the event should not be forwarded to that node. It is therefore desirable to filter events as early as possible in the overlay network on the path between the publisher and the subscribers. If the event traverses multiple nodes, it may be filtered multiple times, but the amount of events filtered by each node — in particular nodes close to subscribers — will be smaller. This is especially important when matching is time-expensive, e.g., when using objects for events and filters. Without pre-filtering, each node or subscriber would need to filter the sum of all events published by all publishers. The main goal of pre-filtering is thus to scale to a large number of publishers (i.e., events), while efficient indexing and matching techniques aim at scaling to a large number of subscribers (i.e., subscriptions).

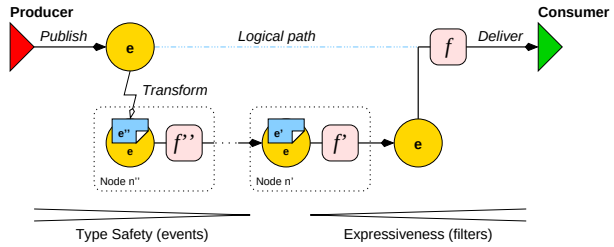

Figure 3: Multi-stage filtering increases scalability by weakening the event representation and the subscription language, and by filtering events on intermediary nodes.

The intuition behind our multi-stage filtering approach is illustrated in Figure 3. Consumer subscriptions are transformed into covering subscriptions that are simpler to evaluate and can be easily indexed for efficient matching. Similar ideas are used in XML document filtering [CFF+02, CFGR02] based on filter aggregation and indexing schemes. Producer events are transformed into covering events adequate for matching against the weakened subscriber filters; with event objects, transformation typically leads to augmenting the event with some meta-data that describes the relevant attributes of the object's state. Filtering is performed on intermediate nodes using the weakened events and filters. Only the events that match all intermediary filtering stages need to be matched against the original filters by the subscriber runtime.

3.3 Transformations

Imperfect filtering is performed through the intermediate nodes by applying transformed filters on transformed events. We now describe how these transformations are performed to guarantee that filtering will be consistent with the original events and subscriptions.

Filter Transformation. A filter f can be transformed into a *weaker* filter f' for the purpose of pre-filtering. Using f' at strategic locations in the infrastructure can reduce the network traffic and the amount of events to be filtered by the subscribers.

Proposition 1 (Filter Transformation). *Given an original filter f, a filter f' can be used for event pre-filtering if and only if $f' \sqsupseteq f$.*

It follows that, if $f' \sqsupseteq f$, then f' can be applied before f to the events without loss of consistency.

Event Transformation. Since the event covering relation depends on a filter, events and filters must be weakened in a coordinated manner. For that purpose, one should use transformation functions that generate covering filters in such a way that weakened events cover original events for all covering filters.

Proposition 2 (Event Transformation). *Given two subscription languages $\mathcal{L}_F$ and $\mathcal{L}_{F'}$, a transformation function $t : \mathcal{L}_F \rightarrow \mathcal{L}_{F'}$, and an original event e, an event e' can be used for pre-filtering if* $\forall f \in \mathcal{L}_F \;\; e' \overset{t(f)}{\sqsupseteq} e$.

3.4 High-Level Abstractions

So far, we have represented events as sets of properties (name-value pairs), and filters as constraints expressed on these properties. When providing high-level abstractions as interface to an event system, i.e., events as encapsulated instances of application-defined object types, and filters possibly expressed with arbitrary language constructs based on the types of these instances, the picture becomes more complex.

Ensuring Event Encapsulation on an End-to-End Base. While expressing event-based distributed interaction through such high-level abstractions is more convenient for programmers, it makes efficient implementations hard. Filtering performance can only be poor if at each filtering stage events have to be deserialized and filtered by performing high-level code. Viewing events as sets of properties is on the other hand not adequate for ensuring encapsulation of events, but leads to more performant implementations. To get the best of both worlds, the event system has to be able to infer a low-level event representation for filtering from a high-level view offered to developers. This low-level representation can be viewed as *meta-data* attached to event objects, which is used for filtering of these objects, and a similar representation is constructed from filters expressed in a high-level programming language.

To support the event system in inferring meta-data, event types have to be designed by following a simple convention: for each attribute (used for filtering), the type offers an access method (used for expressing filters), whose name corresponds to the attribute's name prefixed with `get`. Reflection techniques of modern object-oriented languages are then used to extract information from objects and types.

Arbitrary methods defined in event types, e.g., with parameters, are harder to consider in filtering without actually performing them. These are hence only applied locally, which is sufficient in most cases, since event types are usually fine-grained types encompassing a limited set of attributes, and filters are in most cases expressed on these attributes, i.e., their access methods in our case.

Example 4. A simple stock event can be represented as an instance of the stock class shown below, expressed with a Java-like syntax for illustration. The attributes `symbol` and `price` are declared as private, and the corresponding access methods `getSymbol` and `getPrice` as public. The event system automatically deduces the effective attributes from this. Even if no such attributes were defined in the "type", but only access methods, the event system would look for these attributes at run-time.

```
public class Stock {
  private String symbol;
  private float price;

  public String getSymbol  { return symbol; }
  public float  getPrice   { return price;  }

  Stock(String symbol, float price) {
    this.symbol = symbol;
    this.price = price;
  }
}
```

Filter Interpretation. Similarly, filters can be expressed as closures in a programming language, and are interpreted by the event system. For the sake of simplicity, we continue this discussion through an example building on the `Stock` class introduced above. Consider a filter expressed through a class `BuyFilter` with a single method `match()` (by the absence of closures in Java):

```
class BuyFilter {
  private float last = 0;
  private String symbol;
  private float max;
  private float threshold;

  BuyFilter(String symbol, float max,
            float threshold)
  {
    this.symbol = symbol;
    this.max = max;
    this.threshold = threshold;
  }

  public boolean match(Stock stock) {
    float price = s.getPrice();
    if(price >= max) return false;
    boolean match = (price <= last * threshold);
    last = price;
    return match;
  }
}
```

The filter expresses interest in stock events, cheaper than a given price, such that the value of those stock events is smaller than a percentage of the value of the previous matching stock event.

Consider the following data event d and filters f and g:

```
Stock d = new Stock(``Foo'', 9.0);
BuyFilter f = new BuyFilter(``Foo'', 10.0, 0.95);
BuyFilter g = new BuyFilter(``Foo'', 11.0, 0.97);
```

We can generate the following data event d_1, f_1 and g_1 such that $f_1 \sqsupseteq f$, $g_1 \sqsupseteq g$ and $d_1 \overset{f_1,g_1}{\sqsupseteq} d$:

$$
\begin{aligned}
d_1 &= (class,'' Stock'')\ (symbol,'' Foo'')\ (price, 9.0) \\
f_1 &= (class,'' Stock'', =)\ (symbol,'' Foo'', =)\ (price, 10.0, <) \\
g_1 &= (class,'' Stock'', =)\ (symbol,'' Foo'', =)\ (price, 11.0, <)
\end{aligned}
$$

Filters f_1 and g_1 filter stock events according to their type, symbol, and current price, but not according to the price difference with respect to the previous event. Filtering using f_1 and g_1 is therefore not accurate.

An interesting consequence is that, by weakening filters f and g, g_1 is now covering f_1 ($g_1 \sqsupseteq f_1$). On the common path between publishers and the subscribers that registered f and g, we can now ignore filter f_1 (and its derivative) and keep only filter g_1.

Filter g_1 can be further weakened by generating g_2 such that $g_2 \sqsupseteq g_1$ and $d_1 \overset{g_2}{\sqsupseteq} d$ (no significant benefit will be obtained from weakening d_1 further):

$$g_2 \;=\; (class,'' Stock'',=)\ (symbol,'' Foo'',=)$$

Finally, one can weaken g_2 again and generate g_3 such that $g_3 \sqsupseteq g_2$. Since g_3 only compares a single attribute for equality, one can use the same efficient mechanisms than with topic-based publish/subscribe, e.g., group communication, and define one topic per attribute value. This illustrates the actual fact that topic-based addressing is a degenerated form of content-based addressing. In our case, g_3 filter data according to its type:

$$g_3 \;=\; (class,'' Stock'',=)$$

4 The Architecture

For the implementation of multi-staged filtering, we arrange a set of intermediate nodes in an arbitrarily-deep hierarchy.[1] An example of this arrangement, with four stages, is shown in Figure 4. The "user-level" stage, where subscribers are located, is the lowest stage. The other stages consist of intermediate nodes, i.e., nodes of the overlay network. Events are filtered and forwarded to the subscribers by these nodes. Published events are first forwarded to the top most stage. Then, events traverse down the hierarchy from higher to lower stages. Filters are kept at each node. Weaker filters are applied at higher-stages; the strongest filters are at the lowest stage. As shown in Figure 4, each node has one or more child nodes (or subscribers at the stage-0). Filters associated with children nodes are weakened and stored in the parent node with the corresponding child identity. This procedure is elaborated later in this section. Hence a node has a limited set of filters and an associated set of children nodes which is also limited. That is, nodes do not contain any global knowledge which facilitates scalability. When a node receives an event, it is checked against each filter and if the event matches the filter, it is forwarded to the child node/s associated with the filter.

Example 5. Filters are arranged in a four-stages hierarchy as follows.

[1] Non-hierarchical configurations can also be used, but they have a higher complexity and are not described in this paper.

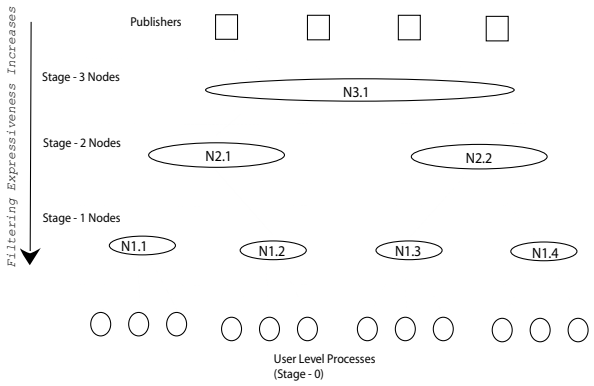

Figure 4: Multi-stage filtering using a hierarchical configuration.

- **Stage-0.** At this stage the received events are evaluated against the exact filters specified by the subscribers. Hence perfect filtering is achieved at this level. It is expected that only a small portion of the overall traffic reaches each of these nodes, and that the match ratio is high (i.e., few irrelevant messages ever reach a user-level node). The subscribers have specified the following filters:

f_1 = (class, "Stock", =) (symbol, "DEF", =) (price, 10.0, <)
f_2 = (class, "Stock", =) (symbol, "DEF", =) (price, 11.0, <)
f_3 = (class, "Stock", =) (symbol, "GHI", =) (price, 8.0, <)
f_4 = (class, "Auction", =) (Product, "Vehicle", =) (Kind, "Car", =) (Capacity, 2K, <) (price, 10K, <)

- **Stage-1.** Filters at this stage are constructed by weakening the subscriber filters. The weakening is done such that the weakened filters cover one or more user-level filters. In general, as a result there will be less filters at this stage than the user-level stage. The weakening of filters can be done by transforming the least general[2] set of attributes as described in Section 4.1. The most general set of attributes are kept unchanged when weakening the filters at this stage.

[2] In a filter such as f_1 = (class, "Stock", =) (symbol, "DEF", =) (price, 10.0, <) we choose "class" as the most general attribute and "Price" as the least general attribute. The process of classifying attributes from most general to least general is described in Section 4.1.

The user-level filters (f_1, f_2, f_3, f_4,) are weakened to obtain the following filters (g_1, g_2, g_3) which will be used at first stage.

g_1 = (class, "Stock", =) (symbol, "DEF", =) (price, 11.0, <)
g_2 = (class, "Stock", =) (symbol, "GHI", =) (price, 8.0, <)
g_3 = (class, "Auction", =) (Product, "Vehicle", =) (Kind, "Car", =) (Capacity, 2K, <)

- **Stage-2.** Filters at this stage are constructed by weakening the filters at first stage. When weakening, the least general set of attributes which were already weakened are removed. The next set of least general attributes are weakened as appropriate to form the filter at the second stage. Note that, by removing attributes from the filters, filtering speed is increased even if the number of filters does not change.

  The filters h_1, h_2, and h_3 are constructed by transforming g_1, g_2, and g_3.

h_1 = (class, "Stock", =) (symbol, "DEF", =)
h_2 = (class, "Stock", =) (symbol, "GHI", =)
h_3 = (class, "Auction", =) (Product, "Vehicle", =) (Kind, "Car", =)

- **Stage-3.** At this stage filtering is done only on the type of events. That is, the filters are constructed by weakening second stage filters such that the newly formed filters contain only type information. Finally h_1, h_2, and h_3 are transformed to obtain i_1 and i_2.

i_1 = (class, "Stock", =)
i_2 = (class, "Auction", =)

The above shown weakening process can be automated such that the system creates all the weakened filters once the subscription filters are available for an advertised set of events. A description of the automation process is given next.

4.1 Automating the Filter Weakening

One approach to obtain weakened filters is to make use of < or > relations, as shown in Example 5 where the weakened filter g_1 is obtained to cover f_1 and f_2. This approach does not apply, however, to other relations such as "=". We therefore introduce a more elaborate scheme to automate filter weakening in a generic manner.

Informally, our automation process works as follow. At each stage, a subset of attributes is removed from the filters to form the weakened filters. For removing attributes, we first categorize attributes into groups (process to be described shortly). Each group is associated with a stage and consists of one or more attributes, which are used at that stage to weaken the filter. When generating an event, the publisher specifies the groups and the attributes they contain. This information is disseminated together with event advertisements. When a node subsequently receives a filter to be weakened, the node can use the group information from the publisher and create a weakened filter in an automated fashion, according to the node's stage. Once both the advertisements and subscriber filters are available, all the intermediate filters can be generated automatically. This process is presented more formally in the rest of this section.

Attribute-Stage Association. Let G_c be a set, which specifies the association of attributes of the weakened filters with each stage for the event class c in a multi-stage filtering scheme with $n+1$ stages. We represent G_c as follows:

G_c = $\{s_0, s_1, \ldots s_n\}$
s_i = < Stage-i: A_i >

Where A_i is the set of attributes used in the weakened filters at Stage-i. G_k is sent by producers together with advertisements of event class k.

Example 6. Consider the event class "auction" and filter f_4 of Example 5. The following information can be propagated to all the nodes at the advertisement phase.

$G_{Auction}$ = $\{s_0, s_1, s_2, s_3\}$
s_0 = <Stage-0: 1, 2, 3, 4, 5>
s_1 = <Stage-1: 1, 2, 3, 4>
s_2 = <Stage-2: 1, 2, 3>
s_3 = <Stage-3: 1>

Tuple s_1, for instance, is interpreted as follows: at Stage-1 the first four attributes of the standard subscription filter[3] are used to form the weakened filter. Therefore, g_3 is obtained from f_4 by keeping only the first four attributes at Stage-1.

Grouping the attributes. Among all attributes, the attribute which divides the event space into a small set of large sub-categories (i.e., containing many events) is called the most general attribute. Conversely, the least general attribute divides the event

[3]We denote as "standard subscription filter" the filter that contains all the attributes (see Section 4.4).

space into many small sub-categories. Attributes are arranged from the most to the least general, and groups are created accordingly. In the filtering process, the most general attributes are used at higher stage nodes (while less general attributes are ignored). In our notation, we order filters from left to right starting by the most general.

4.2 Subscription

A subscriber wanting to subscribe to an event should connect to a particular node in the hiearchy so that the events will be forwarded to the subscriber. One possibility is connecting to a node depending on the locality in terms of the network. That is, the subscriber connects to a node which has a low latency and high bandwidth link. Even though low latency message delivery is possible with this scheme, "similar" subscriptions might not be "closer" to each other in the hierarchy. For example, though f_1 and f_2 are similar subscriptions which differ only by one specific attribute filter they could be connected to node *N1.1* and *N1.4* respectively. As a result there would be 2 similar covering filters covering f_1 and f_2 stored in the Stage-1 (in *N1.1* and *N1.4*). Also the events which match f_1 and f_2 would be forwarded twice; i.e., along two paths via *N2.1* and *N2.2*. This is not efficient - espcialy when there are more than two similar filters- as will be seen next.

On the other hand, subscribers can be connected to nodes by arranging similar subscriptions together. That is, if 2 subscriptions have number of equal attribute filters as with f_1 and f_2, they can be arranged closer to each other in the hierarchy. For example, they both can be connected to the node *N1.1*. In this configuration there will be only one covering filter at Stage-1 (at *N1.1*). Also the events that match f_1 and f_2 only need to be forwarded along a single path; i.e., via *N2.1*. This increases the filtering efficiency and bandwidth usage. Filtering efficiency is increased since in parent nodes, a single weakened filter covers many children/subscription filters. As a result there are less number of filters to be evaluated in the system. Bandwidth usage should increase since an event is forwarded along a less number of paths in the network. The gain achieved by this configuration is quite significant when there are many similar subscriptions.

We propose an algorithm that arranges "similar" subscriptions together at the phase of subscription. When a subscriber joins the hiearchy, with its subscription filter a "search" is done to find the node in which similar subscription filters are stored. This is acheived by searching the strongest filter which covers the subscription filter and which is already stored in

Send *Subscription(f_{sub})* to the root node
while Joined $\neq$ True **do**
  if receives *join-At(id_{node})* **then**
    send *Subscription(f_{sub})* to id_{node}
  if receives *accepted-At($node_i$)* **then**
    Parent $\leftarrow node_i$
    Joined=True
□
task RENEW THE SUBSCRIPTIONS {*Before the Expiry of each Time To Live(TTL)*}
  send *Renewal-of-subcription* to *Parent*

(a) At Subscriber :

Upon Receiving *Subscription(f_{sub})*
if $node_i$ is not in the Stage-1 **then**
  if $\exists$ weakened filter f_w stored in $node_i$ such that $f_w \sqsupseteq f_{sub}$ **then**
    $id_n \leftarrow$ id of node associated with f_w
    Send *join-At(id_n)* to the subscriber
  else
    if $\exists$ a set C of wild-card attribute filters in f_{sub} **then**
      do HANDLE-WILDCARD-SUBS()
    else
      $id_{child} \leftarrow$ Randomly selected child of $node_i$
      send *join-At(id_{child})*
else
  do INSERT-SUBSCRIBER()
□
Upon Receiving *req-Insert(f_c, id_c)* from a child
Add $<f_c, id_c>$ pair to the filtering table
if Not root **then**
  send the weakened f_c to the parent
□
task EXTEND THE VALITIDY OF FILTERS {*Before the Expiry of each Time To Live(TTL)*}
  send renewal messages to all the filters submitted to the *Parent*
□
task REMOVE INVALID FILTERS {*At the end of each 3x(TTL) periods*}
  Remove the filters/ids to which renewal messages are not received
□
INSERT-SUBSCRIBER()
$f'_{sub} \leftarrow$ weakened form of f_{sub}
store $<f'_{sub}, id_{sub}>$ in $node_i$
send *accepted-At(i)*
$f''_{sub} \leftarrow$ weakened form of f'_{sub}
send *req-Insert(f''_{sub}, i)* to parent
□
HANDLE-WILDCARD-SUBS()
find the most general attribute $Attr_{mg}$ from C
using G find the top most Stage j at which $Attr_{mg}$ is used
if $Node_i$ is at Stage (j +1) **then**
  do INSERT-SUBSCRIBER()
else
  $id_{child} \leftarrow$ Randomly selected child of $node_i$
  send *join-At(id_{child})*

(b) At $Node_i$:

Figure 5: Algorithm for Insertion and Deletion of Subscription

an intermediate node. The algorithm which performs this is shown in the Figure 5(a) and 5(b). It is efficient since it looks for the covering filter in stage by stage basis and does not look for the covering filter in each and every node.

4.3 Unsubscription

When a subscriber joins a node in the hierarchy, a filter is stored in this node. As a result the node submits a weaker filter to its parent and the process continues along all the stages until there is a corresponding weaker filter in the root node. In our scheme the subscriber is expected to renew its subscription before the end of a predefined Time-To-Live (TTL). Also each node renews the validity of the filters which they submitted to their parents. When a subscriber or a node doesn't send renewal messages for their filters, it is assumed that these subscribers or nodes do not exist or are not interested in the subscriptions. As a result after the expiry of TTL those filters are removed from corresponding nodes. The unsubscriptions are handled according to this scheme. The scheme is better suited than explicit unsubscription in some sense as it handles process failure and network partitions well, in which case explicit unsubscribe messages cannot be sent. If necessary, this scheme can be combined with explicit unsubscription for efficiency.

4.4 Handling Missing Attributes in Subscriptions

In some subscription filters it is possible that some of the attributes are not specified or specified for all possible values. We denote them as "wildcard" subscription. Some examples are shown below.

f_x = (class, "Stock", =) (symbol, "DEF", =)
f_y = (class, "Stock", =) (symbol, "ALL", =) (price, 100, <)
f_z = (class, "Stock", =) (price, 100, <)

In f_x the attribute "price" is not specified (unlike in f_1), hence the subscriber should receive all the events which match f_x irrespective of price values. If an attribute is not specified in a subscription filter, it means that the subscriber is interested about the events irrespective of the values of the unspecified attribute. As a result, in the above example, f_y and f_z are equal. In our scheme all the subscription filters are converted to filters having all the attributes specified; for example in the form of f_y. That is, if the subscriber doesn't specify an attribute *Attr* -which exists in events- in its filter, (Attr, "ALL",=) is automatically added to the filter. We denote this form as standard subscription filter format and attribute filters in the form (Attr, "ALL",=) is denoted as "wildcard" attribute filters.

With this kind of filters, a question arises about the node to which these subscribers should be connected. If they are connected to a stage-1 node naively, the efficiency of the system could be significantly reduced. One basic idea of our pre-filtering scheme is to limit the number of events that pass through nodes. As a result, the number of evaluations that need to be carried out at each node is decreased. But if a subscription filter like f_y or f_z is linked to Stage-1 node, say N1.1, then that node will receive all the events in the class stock. There can be a huge number of stock events. This overloads the node N1.1 as it will have to evaluate all these events. The same scenario applies for the node N2.1 which forwards events from the root node to the subscriber via N1.1.

To avoid this inefficiency, we devise a scheme in which such subscription filters are connected to higher stage nodes directly instead of connecting them to stage-1 nodes. The algorithm in Figure 5(b) presents this scheme more formally and a description is given in Section 4.5

4.5 The Algorithm

The algorithm shown in Figure 5(a) and 5(b) handles subscription and unsubscription (including the ones with wild-card attribute filters). The basic idea of this algorithm is as follows.1) The subscriber sends a subscription request with the corresponding filter f_{sub} to the root node. 2) Any node $node_i$(including the root node) unless it is at the stage-1, upon receiving a subscription request checks if there exists a filter f_w which covers f_{sub} stored in the $node_i$. If there exists a f_w, the subscriber is asked to send the subscription request again to the node id_n which is associated with f_w. It is possible that the node id_n has a stronger covering filter for f_{sub}. 3) If there is no such covering filter for f_{sub} in $node_i$, then it checks if any wild-card attribute filters exist in f_{sub}. If there exist, the HANDLE-WILDCARD-SUBS() function (described next) is executed. If no wild-card attribute filters exist in f_{sub}, the $node_i$ selects one of its children node randomly and asks the subscriber to join at the selected child node. 4) If a subscription request message is received by any node at the stage-1 the subscriber is joined at this node. The subscriber is notified about this by using "accepted-At(*node*)" message. Then the node sends a corresponding weakened filters to the parent node so that events which match f_{sub} will be forwarded properly along the paths from the root to the node at stage-1.

As a result once this is performed, if there exists a stronger filter in a node which covers f_{sub}, the subscriber joins this node. Hence "similar" subscription filters are grouped together.

Handle-wildcard-subs. 1)First find the most general attribute $Attr_{mg}$ from the set C, which contain all the wildcard attribute filters of the f_{sub}. (In other words, scan the f_{sub} from left to right and find the first occurrence of the wildcard attribute. This is the most general wildcard attribute since attribute filters are ordered according to the generality.) 2) Using G (which associate stages with attributes) find the top most stage j at which $Attr_{mg}$ is used. If $Node_i$ is at stage (j+1) then insert the subscribers at this node. 3) Otherwise select a child node of $node_i$ randomly and asks the subscriber to join at the selected child node.

4.6 Event Filtering and Forwarding

After the execution of the algorithm shown in Figure 5(a) and 5(b) each node will have a set of weakened filters and corresponding child node/subscriber Ids in the form of $< f_w, id>$. When a node receives an event e and if $f_w(e)$ = true, then e should be forwarded to the child node identified by id. This is the event filtering and forwarding scheme. For each event received by a node, the event should be evaluated and matched against all the weakened filters in the node. For this, efficient indexing and matching techniques can be used. Those techniques are beyond the scope of our paper. For clarity and to present the global picture of our scheme to the reader, we present the simplest possible techniques in terms of an algorithm in Figure 6.

The basic idea behind this naive algorithm is as follows. 1)Each node maintains a table T with entities in the form of $< f'_{sub}, id_1[,id_2,...]>$ where f'_{sub} is a weakened filter and $id_1[,id_2,...]$ is/are Ids of node/s associated with f'_{sub}. 2)Whenever a node stores $< f'_{sub}, id_1 >$ (Figure 5(b)) insert the $< f'_{sub}, id_1>$ pair in the table T. If f'_{sub} already exists in T add id_1 to the list of Ids associated with f'_{sub}. 3) For each event e evaluate all the filters in the T with e.4) Whenever the result is true for the filter f_w, forward the e to the corresponding node/s.

5 Performance Evaluation

To have a type safe, expressive (these two properties add overhead to filtering) and a scalable event system, the computational power requirement at each node should be as low as possible for the proposed filtering scheme. To measure the power requirement for filtering, we introduce the notion of Load Complexity (LC). We also introduce the Relative Load Complexity (RLC) notion for the purpose of comparing the multi-stage event system with the centralized approach (Section 2.1). For evaluating efficiency of pre-filtering, we introduce the Matching Rate (MR) concept. The results clearly show that our multi-stage filtering scheme scales better in terms of the number of nodes and the number of events.

```
Upon receiving a <filter,ID> pair
for all < f'sub, idsub > received do
  if f'sub is already in table T then
    add idsub to the list of IDs associated with f'sub
  else
    insert < f'sub, idsub > to T
□
Upon receiving an event e
for all events e received do
  for all filters f in T do
    if f(e)=true then
      forward e to node/s associated with f
```

Figure 6: Algorithm for Filtering and Forwarding Events

5.1 Metrics

Load Complexity (LC). This is defined as follows; For any time unit, at any node which performs filtering,

$$LC = (\#\ of event\ received) \times (\#\ of filter)$$

Where "# of event received" is the number of messages received for filtering and "# of filter" is the number of filters in the node.

It is clear by keeping a small amount of filters in a node we can reduce the computational power requirement even if there are many number of events in a time period.

The "Relative Load Complexity (RLC)" can be used for comparing a content based event system with many intermediate filtering nodes, with respect to a centralized server architecture (first architecture discussed in Section 2.1) that has a complete set of subscriptions in a centralized server. It is defined as follows; for any time unit, at any node which performs filtering,

$$RLC = \frac{LC}{(Total\ \#\ of Events) \times (Total\ \#\ of Subs)}$$

Where "Total # of Events" is the total number of events received for filtering and "Total # of Subs" is the total number of subscriptions in the event system. This metrix represents how a system balances and delegates the work load of filtering among nodes. For a content based event system with a centralized server, with all subscription filters at one server, the RLC is equal to one. It is desirable to have a smaller value for the RLC.

Matching Rate (MR). This is defined as follows. In a time unit, at any node which performs filtering,

$$MR = \frac{Number\ of\ matched\ events}{Total\ number\ of\ received\ events}$$

A higher value in this metric indicates that a higher number of events are evaluated to "true". This means a node receives a higher number of events that it is interested in. Hence nodes spend less time evaluating unnecessary events against filters and the network bandwidth is used for forwarding more "relevant" events. As a result of pre-filtering, lower stage nodes and subscribers should optimally have a MR value close to 1.

5.2 Simulation Environment

In our simulations, a dummy set of events and a dummy set of subscriptions are generated initially. The events generated represent a simple form of bibliographic data. The attributes of an event are: author, conference, year and title.

We create a hierarchy of nodes with four levels[4]. Then the weakened filters for each stage are created.

The filters at each stage have the following formats:
At Stage-0 (subscription filters)

$f_1 =$ (Year, "year$_i$", =) (Conference, "conference$_i$", =)
(Author, "author$_i$",=) (Title, "title$_i$", =)

At Stage 1:

$f_1 =$ (Year, "year$_i$", =) (Conference, "conference$_i$", =)
(Author, "author$_i$",=)

At Stage 2:

$f_2 =$ (Year, "year$_i$", =) (Conference, "conference$_i$", =)

At Stage 3:

$f_3 =$ (Year, "year$_i$", =)

Each weakened filter is associated with one or more child nodes. Each node has a table which consists of entries with the following format;

<Filter *filter$_i$*>, <IDs of child nodes *id-list$_i$*>

The *filter$_i$* is a subscription filter or weakened form of it. The *id-list$_i$* consists of child nodes that are associated with *filter$_i$*. Once an event is received by a node, it evaluates the event against all the filters. If the evaluation is "true" for a given *filter$_i$* then the event is forwarded to all the child nodes in the *id-list$_i$* of *filter$_i$*. The sizes of the tables varries according to the number of subscriptions and their nature. Similar to Figure 4, our simulation tool consists of 4 levels and we simulate 100 nodes at level-1, 10 nodes at level-2 and 1 node at level-3.

5.3 Results

In our simulations, we passed the pseudo randomly generated events to the node of level 3. This node evaluates the event against all the filters it has and if the event evaluates to "true" for any of the filter, the node forwards the event to respective stage-2 node/s; the event is considered as a "matched event". If the event evaluates to "false" for all entries in the table, then it is discarded by the node and is not be forwarded any further. As a result, child nodes are not receiving any events that they are not interested in. This forwarding process continues until the event is received by subscribers or be discarded by nodes. To evaluate the performance figures, at each node, the number of filters, the number of received events and the number of matched events are counted.

RLC. We calculate the relative load complexity (RLC) using the data gathered from the simulations. The RLC can be used for comparing the processing power requirements at each node.

The following table summarizes the results. The first column shows the respective level. The node average of RLCs of a given level is shown in the second column. The third column shows the total of averages of RLC at each level.

[4]Note that it is not necessary to have equal number of attributes and levels in the hierarchy.

Stage	Node avg. of RLC	Total node avg. of RLC
0	2×10^{-7}	2×10^{-4}
1	2×10^{-4}	2×10^{-1}
2	0.1	1
3	0.02	0.02

According to the RLC values shown above, it is clear that the processing power requirement at each node is much less than the requirement of the centralized server (Section 2.1). Since the computational power requirement for filtering is very much less at any node, the event system hence scales in terms of message rate. As a result more expressive filters can also be used without much performance degradation. The system scales better also with the number of subscriptions since by adding a few number of intermediate nodes, the number of subscribers can be increased significantly without increasing the required computational power at any node. In other words, due to the delegation of work among intermediate nodes, the addition of more subscribers does not overload the existing nodes. The global total of RLCs in the system (addition of all values of the column 3 of the table) is around 1. This shows that, in multi-stage filtering though there are many intermediate nodes in the system, there is no greater computational power requirement in global sense, comparatively to the centralized server approach.

MR. The matching rate (MR) was also calculated for all the nodes in the system. Figure 7 shows the matching rate for 150 level-0 nodes, 100 level-1 nodes and 10 level-2 nodes. Calculations show that the average matching rate is 0.87 for the above subscribers.

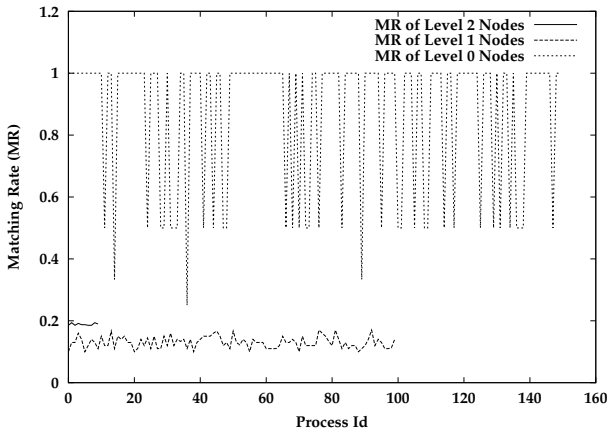

Figure 7: Matching rate of the nodes.

6 Conclusion

In this paper we present the desirable but conflictiong properties of event systems. These properties are event safety, filtering scalability and expressiveness. But as shown in the paper, acheiving them together is a difficult task since they are conflicting to each other. We acheive expressiveness and event safety by representing events as objects. Use of objects increases the complexity of the filtering since -for example- the objects need to be instantiated at run time during the filtering. To circumvent this complexity we use a pragmatic filtering technique based on filter weakening and data weakening. The events are filtered multiple times on the path between the publisher and the subscriber by a set of intermediate nodes; each node performs partial filtering of the events but nodes taken together perform complete filtering of events according to the interests of subscribers. Filter weakening is used to acheive partial filtering at each node while data weakening is used to make the filtering scheme efficient while the preserving the encapsulation. As a result, it is possible to acheive filtering scalability together with event safety and expressiveness.

References

[AEM99] M. Altherr, M. Erzberger, and S. Maffeis. iBus - a software bus middleware for the Java platform. In *International Workshop on Reliable Middleware Systems of the 13th IEEE Symposium On Reliable Distributed Systems (SRDS '99)*, pages 43–53, October 1999.

[Car98] A. Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-area Networks*. PhD thesis, Politecnico di Milano, December 1998.

[CFF+02] C.Y. Chan, W. Fan, P. Felber, M.N. Garofalakis, and R. Rastogi. Tree pattern aggregation for scalable XML data dissemination. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, 2002.

[CFGR02] C.Y. Chan, P. Felber, M.N. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, 2002.

[CNF98] G. Cugola, E. Di Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *Proceedings of the 10th International Conference on Software Engineering (ICSE '98)*, pages 261–270, april 1998.

[Cor99] Talarian Corporation. *Everything You need to know about Middleware: Mission-Critical Interprocess Communication (White Paper).* http://www.talarian.com/, 1999.

[CRW00] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the Principles of Distributed Computing 2000*, pages 219–227, July 2000.

[EGD01] P.Th. Eugster, R. Guerraoui, and Ch.Heide Damm. On objects and events. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA) 2001*, October 2001.

[Eug01] P.Th. Eugster. *Type-Based Publish/Subscribe.* PhD thesis, Swiss Federal Institute of Technology, Lausanne, December 2001.

[HMN+00] M. Haahr, R. Meier, P. Nixon, V. Cahill, and E. Jul. Filtering and Scalability in the ECO Distributed Event Model. In *Proceedings of the 5th IEEE International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 2000)*, pages 83–92, June 2000.

[MÖ1] G. Müehl. Generic constraints for content-based publish/subscribe systems. In *Proceedings of the 6th International Conference on Cooperative Information Systems (CoopIS)*, 2001.

[SAB+00] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content based routing with elvin4. In *Proceedings of the Australian UNIX and Open Systems User Group Conference ((AUUG2K)*, June 2000.

[SBCea98] R. Strom, G. Banavar, T. Chandra, and M. Kaplan et al. Gryphon: An information flow based approach to message brokering. In *International Symposium on Software Reliability Engineering (ISSRE '98)*, November 1998.

[SCG01] A. C. Snoeren, K. Conley, and D. K. Gifford. Mesh-based content routing using xml. In *18th ACM Symposium on Operating System Principles, Banff, Canada*, October 2001.

[Ske98] D. Skeen. *Vitria's Publish-Subscribe Architecture: Publish-Subscribe Overview.* http://www.vitria.com, 1998.

[TIB99] TIBCO. *TIB/Rendezvous White Paper.* http://www.rv.tibco.com/, 1999.