RACED: an Adaptive Middleware for Complex Event Detection

Gianpaolo Cugola, Alessandro Margara
Dip. di Elettronica e Informazione
Politecnico di Milano, Italy
[cugola, margara]@elet.polimi.it

ABSTRACT

While several event notification systems are built around a publish-subscribe communication infrastructure, the latter only supports detection of simple events. Complex events, involving several, related events, cannot be detected. To overcome this limitation, we designed RACED, an adaptive middleware, which extends the content-based publish-subscribe paradigm to provide a complex event detection service for large scale scenarios. In this paper we describe its main aspects: the event definition language; the protocol enabling efficient and distributed detection of complex events through a network of service brokers; the mechanism that enables RACED to dynamically adapt to network traffic. A preliminary evaluation shows the benefits of RACED w.r.t. more traditional publish-subscribe infrastructures.

Categories and Subject Descriptors

C.2.2 [**Computer Communication Networks**]: Network Protocols—*Routing protocols*; C.2.4 [**Computer Communication Networks**]: Distributed Systems—*Distributed applications*

General Terms

Design, Languages, Measurement

Keywords

Publish-Subscribe, Complex Event Detection

1. INTRODUCTION

In the last years, the publish-subscribe communication paradigm [12], and in particular its content-based incarnation [7], has shown its effectiveness in a wide range of scenarios, by providing a strong decoupling among communication parties, which simplifies the design of loosely coupled systems. For this reason publish-subscribe has been widely adopted as a natural substrate to build event notification systems; in this field, however, single events are often useless on their own, while they become relevant when their mutual relations are considered as well.

The traditional publish-subscribe paradigm lacks the expressive power to express and detect *complex events* [10], i.e. events defined as patterns involving relations between other events. To overcome this limitation, different systems have been proposed recently, which extend publish-subscribe with the ability to cope with complex event detection [6, 14, 15]. However, these works are mainly focused on the definition of a rich language for event specification and only few of them address the problem of efficiently distributing events in large scale networks.

In this paper we describe a novel approach for the design of a complex event detection middleware, called RACED (Rate-Adaptive Complex Event Detection). The contribution of our work is threefold: *i.* we propose a simple language for complex event definition which is suitable for distributed processing; *ii.* in order to support large scale scenarios involving thousands of nodes, we developed a protocol to let a set of service brokers, connected in an overlay network, cooperate to efficiently provide the complex event detection service to their clients; *iii.* we augment this protocol with a mechanism that enables RACED to dynamically adapt to network traffic, and in particular to event generation rates.

The rest of the paper is organized as follows: in Section 2 we present the general architecture and the API of our system; in Section 3 we describe our complex event definition language; in Section 4 we present our protocol for distributed event detection, focusing on its capability to dynamically adapt to event generation rates; in Section 5 we evaluate such protocol, using the Omnet++ network simulator [16]. Finally in Section 6 we survey related work, providing some concluding remarks in Section 7.

2. SYSTEM ARCHITECTURE

The architecture of RACED is shown in Figure 1.

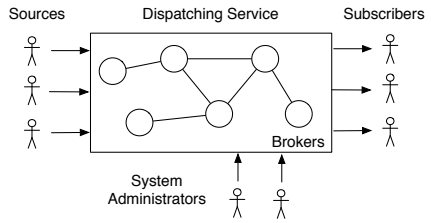


Figure 1: System architecture

ARM 2009, December 1, 2009, Urbana Champaign, Illinois, USA.

Sources notify the dispatching service about events. Examples of sources are temperature and smoke sensors sending notifications about some location, or a traffic monitoring system, notifying about congestioned routes. We call such event notifications *messages*. *System administrators* define complex events: for example they can define that an event of type `fire` is detected when the system receives both a message about high temperature and a message about smoke coming from the same room. Finally, *subscribers* ask the system to be notified about the detection of certain (complex) events. Internally, the *dispatching service* is built around different brokers, connected in an overlay network, which cooperate to detect and route events from sources to subscribers. Table 1 presents the API of RACED.

`Advertise(MessageType t)`
`Publish(Message m)`
`Subscribe(EventType t)`
`DefineEvent(EventType t, Pattern p)`

Table 1: The RACED API

Before sending messages, sources have to invoke the `advertise` operation, to declare the type of messages they will publish. This build a contract between sources and the dispatching service: only messages whose type has been advertised can be published; such contract is exploited by the dispatching service to enable distributed detection of complex events, as explained in Section 4. The `Publish` operation is used to inject new messages into the dispatching system, while the `Subscribe` operation is invoked by subscribers to express the type of events they are interested in. Finally, complex events are defined by system administrators using the `DefineEvent` operation, which specifies the type t of the new event and a `Pattern` that expresses when an event of type t occurs. Next section introduces the syntax and semantics of such patterns.

3. EVENT DEFINITION LANGUAGE

Several languages have been recently proposed to define complex events [6, 14, 15]. In most cases they privilege expressiveness over simplicity. Being interested in providing a complex event detection system tailored to large scale scenarios, we took the opposite approach and designed a low level language with few operators, optimized to allow detection of complex events in a distributed way.

The RACED language provides just five constructs: *i.* message filters, *ii.* composition operators, *iii.* parameters, *iv.* windows, and *v.* event definitions.

Higher level constructs can be easily built on top of it (e.g. sequences and repetitions can be represented by combining message filters and composition operators).

Message filters. Filtering is the capability, offered by content-based publish-subscribe systems, to select or discard single messages according to their content. Our language offers the same functionality. Messages have a type and a set of *attributes* represented as key-value pairs as in 1. Message filters define the type of the matched messages and the possible values for attributes through a predicate (a conjunction of constraints on single attributes). As an example, the filter given in 2 *matches* the message given in 1.

(1) `Temp[Value:20, Location:"office 1"]`

(2) `Temp[Value>10 AND Location="office 1"]`

Composition operators. Message filters can be combined through the operators `AND` and `NOT` to define events whose occurrence requires different messages to be published. As an example, in 3 we combine two messages that have to occur and a third that have not to occur for a certain event to happen.

(3) `Temp[Value>30] AND Alert[Type="smoke"] AND NOT Weather[State="rain"]`

Parameters. Consider Example 3: in many scenarios the fact that a message containing a high temperature and a smoke alert have been published in a non raining condition may be meaningless. However, it becomes useful if the three notifications are all related to the same area, as they may indicate a possible fire. To express patterns in which different messages are selected only if they satisfy mutual relations, our language enables the definition of parameters and constraints on them. Example 4 shows a pattern that satisfies the aforementioned requirements using a parameter `$X`.

(4) `Temp[Value>30 AND Location=$X] AND Alert[Type="smoke" AND Location=$X] AND NOT Weather[State="rain" AND Location=$X]`

Windows. Sometimes the composition of multiple messages is meaningful only if they are published in a limited amount of time. For example high temperature and smoke notifications from the same location are not relevant if they are generated in different days while they become significant if generated within 5 minutes. Additionally there exist patterns that cannot be processed without time constraints. Consider again Example 3: how long does the system have to wait until it can decide that no rain messages have been received? The `NOT` operator cannot be evaluated without explicit timing constraints[1].

For these reasons our language includes windows defined using the `WITHIN` operator as in Example 5. When the `WITHIN` clause is not specified we assume that a default value is used.

(5) `Temp[Value>30] AND Alert[Type="smoke"] WITHIN 5 min`

It is worth mentioning that the exact semantics of the windowing mechanism depends on the time model provided by the underlying system. In particular, in our system published messages are time-stamped by the first broker receiving them, while brokers' clocks are kept in sync with an error that we assume being not significant for the kind of applications we focus on.

Event definitions. System administrators define events invoking the `DefineEvent` operation (part of the system API). It includes a pattern defined using the operators above together with the definition of the attributes valid for the new event. As an example, in 6 we define the new event `Fire` with two attributes: `Temp` and `Location`.

(6)
```
DefineEvent(Fire[Temp:$X, Location:$Y],
  Temp[Value=$X>30 AND Location=$Y] AND
  Alert[Type="smoke" AND Location=$Y]
  WITHIN 5 min)
```

[1] `NOT` is known as a *blocking* operator [8].

4. EVENT DETECTION PROTOCOL

To support large scale scenarios, we designed a protocol that defines how multiple service brokers cooperate in RACED. Such protocol delivers subscriptions exploiting the shortest path tree rooted at the subscriber[2]; during delivery, subscriptions are partitioned at each hop, letting each source receive only those parts that are relevant for the message types it advertised. Messages follow the opposite route ascending the tree up toward the relevant subscribers, being filtered and combined along the route.

In particular, each broker runs a link-state protocol to collect information about the topology of the dispatching network. It exploits such information to compute its *shortest path tree* (SPT) using Dijkstra. The computed SPT is forwarded to all other brokers in the network, so that everyone could store its position (i.e., children and parent) in the defined tree. The SPT is initially used to propagate new event types defined using the `DefineEvent` primitive so that every broker could store them.

Next sections detail how advertisements, subscriptions and messages are forwarded. For simplicity, we consider a single subscriber and its SPT.

4.1 Forwarding of Advertisements

Advertisements are forwarded from sources up to the subscriber. Each broker saves all the message types contained in the advertisements coming from its descendants in an *advertisement table*. In Figure 2 we show the advertisement table of broker 2 after it has been filled (we denote the set of message types advertised by broker x as $types(x)$). Broker 1 is the subscriber.

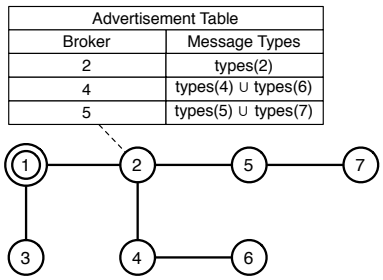


Figure 2: Forwarding of advertisements

4.2 Forwarding of Subscriptions

When a client connected to a broker B calls the `Subscribe` operation for an event type t, the broker B looks at the `Pattern` p defining t and creates a packet we call *subscription* that contains the following fields:

- *Positive Filters* (PF): it is the set of all non-negated message filters that appear in p.
- *Negated Filters* (NF): it is the set of all negated (preceded by the `NOT` clause) filters that appear in p.
- *Window* (W): is the timing window expressed in the `WITHIN` clause of p.
- *Type* (T): is the type of the packet, which can be either *push* or *pull* (more on this later).

This packet is sent to all other brokers along B's SPT by partitioning it while it travels. To understand how such partitioning works we have to introduce the concept of *sending set*. The sending set of a filter f for a broker B is the set of all brokers in B's advertisement table that advertised the type of f.

When a broker B' (including B) have to forward a subscription s it partitions the filters in s (i.e., in PF and NF) according to their sending sets (filters with the same sending set belong to the same partition); then it creates a new subscription s' for each partition, including the filters in that partition; finally it sends each subscription s' to all children in the sending set of the filters in s'.

Using this mechanism the detection of a complex event is recursively decomposed into the detection of its parts. When a broker B' receives a subscription s, it becomes responsible for the detection of messages matching the positive and negated filters of s and for the transmission of collected information up to its parent. By partitioning s, B' delegates part of this detection to its children.

Figure 3 provides a concrete example of partitioning. Types are represented using capital letters and filters are represented through their type. Assuming that broker 2 receives from its parent (broker 1) the subscription s shown in figure, which contains four positive filters and one negated, we show the sending sets calculated by 2 and, for each child of 2, the subscriptions it receives.

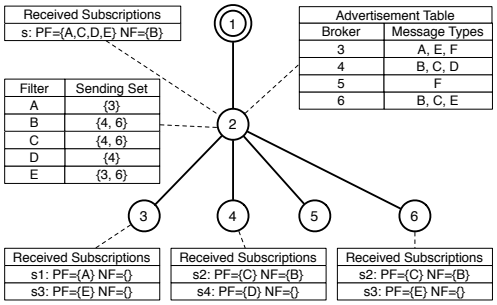


Figure 3: Processing of subscriptions: an example

4.3 Forwarding of Notifications

Each broker stores the data contained in all subscriptions it receives and uses them to filter and combine messages. In particular, when a broker B receives a message m from one of its clients, it first checks, for each stored subscription s, whether m matches one of the filters contained in s. If this happens, then m is stored inside a data structure called *History*, which B uses to detect the *matching sets* for s, i.e. the sets of messages satisfying all the filters in s.

An example of message processing is shown in Figure 4. Consider the subscription s shown there; it requires the detection of a message matching filter A and one matching filter B in a window of size 3. At time T=1 the broker receives the message A1 that matches filter A; at time T=2 it receives the message B1 that matches filter B. Messages A1 and B1 together form a matching set for s. When, at T=3, the broker receives the message B2, it forms a new matching set with A1 and B2. Finally, at T=4, A1 is deleted from the

[2]For simplicity we forget about subscribers and sources to focus on the overlay network of brokers; for this reason in the following we use the term *subscriber* (resp. *source*) to indicate the broker to which a subscriber (resp. source) is connected.

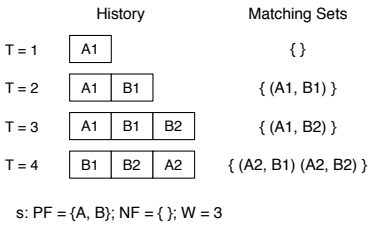


Figure 4: Processing of messages: an example

History, as it is too old for the detection window considered, while A2 arrives, resulting in two new matching sets, one with A2 and B1 and one with A2 and B2.

Notice how in the case of negated filters inside the subscription, matching sets are detected only if there are no messages matching negated filters in the window.

Detected matching sets are delivered to the parent node inside packets called *notifications*. Each notification may include multiple matching sets, each including multiple messages. At the parent node, messages contained in the received notifications are processed again using the same procedure described above. Whether message sets matching a subscription s are delivered immediately when detected, or stored and delivered later, is determined by the *type* of s as described below.

4.4 Push vs. Pull-Based Forwarding

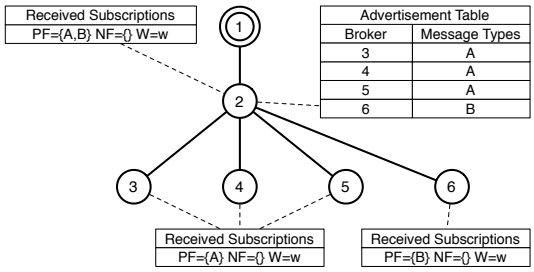


Figure 5: Pull-based Forwarding: an example

Consider now the situation shown in Figure 5, in which broker 2 receives a subscription containing two positive filters, one for messages of type A and one for messages of type B. According to its advertisement table, it delegates the detection of messages of type A to brokers 3, 4, and 5, and the detection of messages of type B to broker 6. In this case broker 2 can satisfy all the constraints of its subscription only if it receives *both* a message matching the first filter *and* a message matching the second filter generated within w. This means that, if broker 6 never sends notifications, all messages received from brokers 3, 4, and 5 are useless and only waste network resources.

Starting from these considerations we introduced in our protocol the concept of *pull-based forwarding* as opposed to the more common push-based approach. In particular, every subscription has an associated *type*(T) which can be either *push* or *pull*. A push subscription requires the broker receiving it to promptly send all matching sets of messages up to its parent; on the contrary a pull subscription requires the broker to store matching sets of messages until the parent explicitly asks for them. So, in the example of Figure 5, broker 2 could decide to send the subscription for messages of type A as a pull subscription and to ask for the delivery of stored messages only after receiving messages of type B from broker 6.

More specifically, the mechanism to decide the type of subscriptions to send to children and to ask for stored messages (in case of pull subscriptions) works as follows. When a broker B receives a subscription s, it processes and partitions it as explained above. For each newly generated subscription s' it looks at the sets PF and NF. If PF is empty then the subscription is processed in a special way: its type is set to push and the receiving children is asked to promptly send up messages matching the filters in NF. Among all the subscriptions (if any) having a non empty PF only one is selected as the *master* subscription, while all the others are considered *slave* subscriptions. The master is sent as a push subscription while the slaves are sent as pull. When broker B detects a matching set of messages for the master subscription, it sends an *open* packet to all children processing slave subscriptions. The open packet causes children to send all message sets they had detected so far (if any) and to continue to send new sets using a push approach for a period long w, where w is the window of s.

It is worth mentioning that in case of shared parameters between filters of the master subscription and filters of the slave subscriptions, the open packet asks only for messages having the right values for the shared parameters (the values that appear in the messages that matched the master). This reduces the number of message sets that have to be sent in reply to an open packet.

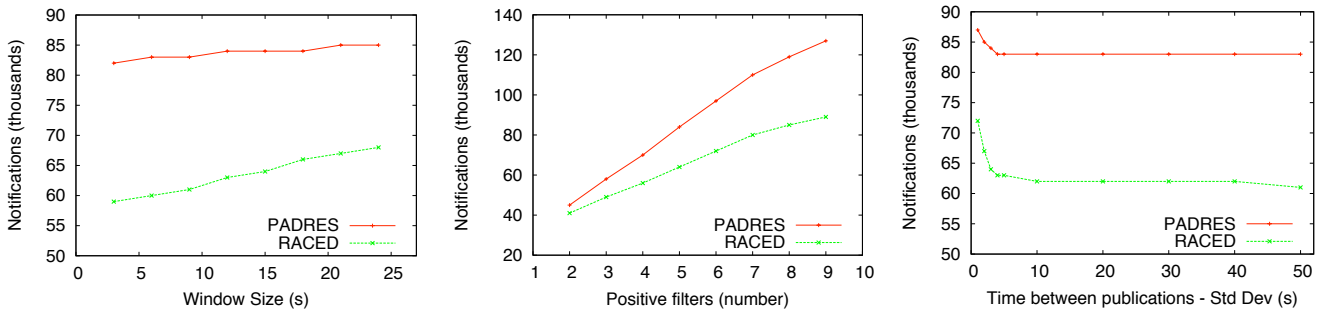
4.5 Adaptive Selection of Masters

The right choice for the master vs. slave subscription may strongly influence the performance of our protocol. In fact, if messages satisfying the master subscription are received sporadically, then fewer requests are sent to children holding slave subscriptions, which may drop several packets locally (i.e., those exiting the window) resulting in less network traffic. On the contrary a master subscription that continuously receives notifications eliminates the benefit of the pull-based approach. To address this issue, our protocol monitors traffic flowing in the network and let each broker adapt its choice of master subscriptions to the traffic monitored in the previous time frame. More specifically each broker B stores, for each subscription s it has received, the number n of matching sets it has detected in a given amount of time t and computes the *generation rate* of s, $gr(s) = n/t$. At the same time, periodically B decides the master selection for s by asking to its children the generation rates of all the partitions of s that it sent them. The part having the lowest generation rate is chosen as the new master.

In summary, the mechanism combining push and pull-based forwarding, coupled with this adaptive mechanism in the choice of which part of a subscription to manage as push and which to treat as pull, results in the ability for our protocol to optimize complex event detection to the actual traffic, minimizing the route followed by messages to be matched and combined together.

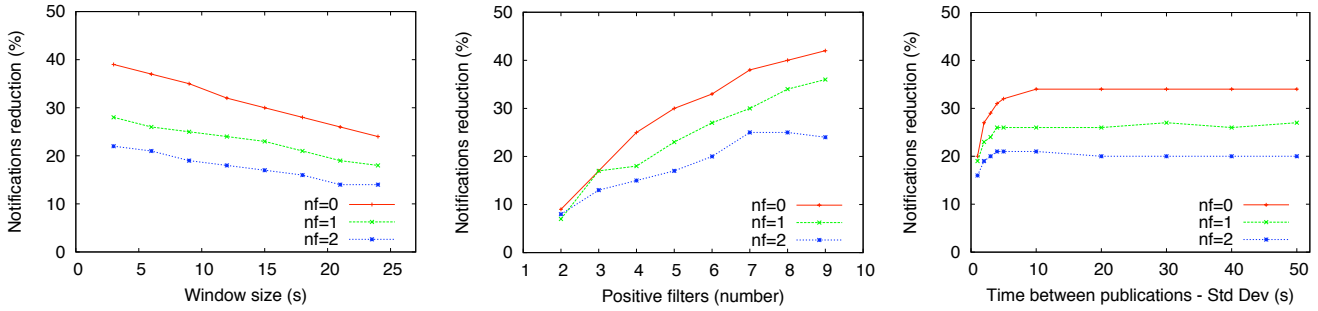
5. EVALUATION

To evaluate the benefits of the distributed detection protocol of RACED, we compared it with PADRES [9]. As explained in Section 1, only few works have addressed the prob-



(a) Window Size (b) Number of Filters (c) Publications Rate Distribution

Figure 6: Number of notifications generated



(a) Window Size (b) Number of Filters (c) Publications Rate Distribution

Figure 7: Notifications savings

lem of distributing detection of complex events; PADRES represents probably the most promising effort in this direction. Similarly to RACED, PADRES defines an advertisement mechanism and exploits it to partition subscriptions in a network of brokers. However, it delivers all notifications using a push-based approach; as a consequence, brokers cannot prevent the delivery of useless notifications, as it happens with our push/pull mechanism. Additionally, PADRES does not provide any mechanism to adapt to network traffic, as RACED does.

To compare the two protocols, we implemented both in Omnet++ [16] and tested them in a scenario of 120 brokers and 600 clients, each one publishing messages of a single type chosen among 100 different types, with different contents. We analyzed the cost of forwarding by counting the total number of notifications generated in the network after all clients have published 1000 messages.

In particular, we analyzed the impact of three parameters: the size of the detection window, the number of filters composing the event to be detected and the distribution of publications rates. In Figure 6 we show the number of notifications generated by the two protocols when the complex event to match does not contain any negated filter. In Figure 7 we present the improvement of RACED over PADRES in the same scenarios showing how it changes when the number of negated filters in the event definition (`nf`) varies from 0 to 2.

Figure 6(a) shows how the number of notifications increases in both protocols when the size of the detection window grows. Looking at Figure 7(a) we notice how small windows favor RACED more, as notifications coming from slave subscriptions are stored for less time, increasing the chance that an open packet is not followed by any reply.

Figure 6(b) shows how the number of notifications increases with the number of positive filters in the event. On one hand adding positive filters decreases the chance of capturing the event, but this trend is dominated by the huge number of matching sets detected when many single messages have to be combined in the matching event. As shown in Figure 7(b), large numbers of filters increase the gain of RACED, as they promote the recursive decomposition of the subscription while it moves from subscriber to sources, thus maximizing the advantages of our push/pull and adaptive mechanisms.

Figure 6(c) shows how the traffic decreases when we increase the variance of publication rates. Increasing the variance, in fact, also increases the possibility of having messages generated so rarely that they are hardly ever captured inside a detection window. Figure 7(c) shows the benefits of our adaptive mechanism with the advantage of RACED over PADRES increasing from 20% up to more than 35%.

If we look at Figure 7 we may also observe how the number of negated filters `nf` reduces the advantage of RACED over PADRES. There are two reasons for this: first, negated filters reduce the propagation of useless notification, problem that affects PADRES more than RACED; at the same time, negated filters limit the possibility to define pull-based subscriptions. Finally, it is worth mentioning how our simulations did not take into account parameterization; considering it would bring even better results, as it would enable finer grain selection of notifications from pull-based subscriptions.

6. RELATED WORK

In the last years a large number of content-based publish-subscribe systems have been developed [12, 7, 11, 5]; proposed solutions were, at the beginning, based on a centralized dispatcher, but soon they moved to distributed solutions to improve scalability.

All these systems share the same communication model, in which messages bring data and subscriptions filter single messages according to their content. Recently, a few works have been proposed that extend the expressive power of traditional content-based publish-subscribe to take into account information contained in multiple messages [6, 14, 15]. However, these works are mainly focused on the definition of rich languages and usually don't address the problem of event matching in large scale systems. Among the few exceptions, the PADRES system [9] adopts a distribution protocol similar to ours; in particular it exploits a tree-based topology to distribute information and advertisements to filter subscriptions. However, all notifications flow using a push-based approach and the dynamics of network traffic is never taken into account.

The problem of combining information coming from multiple sources and to distribute it to users has been addressed also in the field of so called DSMSs (Data Stream Management Systems) [3]. These systems define highly expressive languages, usually derived from SQL [2, 4], which are suitable for generic data manipulation. Even if some of these systems [1] address the problem of increasing scalability by distributing processing, the proposed solutions focus on clustering scenario, in which a set of co-located machines shares the load of processing, while we take the network cost into account and focus on processing messages as close as possible to the sources.

Finally, in [13], authors propose an algorithm to distribute operators in a network of service brokers trying to minimize a cost function, which involves data generation rates. This proposal, however, does not contain any mechanism to inhibit useless data to be propagated in the network, like our master-slave algorithm.

7. CONCLUSIONS

In this paper we presented the design of RACED, an adaptive middleware that extends the content-based publish-subscribe paradigm to provide a complex event detection service for large scale scenarios. In particular, we introduced an event definition language and we described a protocol enabling efficient and distributed detection of complex events inside a network of service brokers. To increase performance, our protocol includes an adaptive mechanism that allows brokers to dynamically adapt their behavior to network traffic. Our tests show that it provides evident benefits over more traditional solutions.

We plan to extend our work in two directions: on one side we are investigating optimization techniques for multiple subscriptions; at the same time we are extending our event definition language to provide not only event detection, but also event processing, for example to compute aggregate values.

Acknowledgment

This work was partially supported by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom; and by the Italian Government under the projects FIRB INSYEME and PRIN D-ASAP.

8. REFERENCES

[1] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik. Aurora: a data stream management system. In *SIGMOD*. ACM, 2003.

[2] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2), 2006.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*. ACM, 2002.

[4] Y. Bai, H. Thakkar, H. Wang, C. Luo, and C. Zaniolo. A data stream language and system designed for power and extensibility. In *CIKM*. ACM, 2006.

[5] R. Baldoni and A. Virgillito. Distributed event routing in publish/subscribe communication systems: a survey. Technical report, DIS, Università di Roma "La Sapienza", 2005.

[6] S. Courtenage. Specifying and detecting composite events in content-based publish/subscribe systems. In *ICDCS Workshops*. IEEE Computer Society, 2002.

[7] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surveys*, 2(35), 2003.

[8] Y.-N. Law, H. Wang, and C. Zaniolo. Query languages and data models for database sequences and data streams. In *VLDB*. VLDB Endowment, 2004.

[9] G. Li and H.-A. Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In *Middleware*. Springer-Verlag New York, Inc., 2005.

[10] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[11] G. Mühl, L. Fiege, F. Gartner, and A. Buchmann. Evaluating advanced routing algorithms for content-based publish/subscribe systems. In *MASCOTS*, 2002.

[12] G. Mühl, L. Fiege, and P. Pietzuch. *Distributed Event-Based Systems*. Springer, 2006.

[13] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE*. IEEE Computer Society, 2006.

[14] P. R. Pietzuch, B. Sh, and J. Bacon. Composite event detection as a generic middleware extension. *IEEE Network*, 18, 2004.

[15] A. Ulbrich, G. Mühl, T. Weis, and K. Geihs. Programming abstractions for content-based publish/subscribe in object-oriented languages. *CoopIS, DOA, and ODBASE*, 3291, 2004.

[16] A. Varga. The omnet++ discrete event simulation system. *ESM*, 2001.