Distributed Event Streams Composition

présentée le 20 février 2016
Université de Grenoble Alpes

pour l'obtention du grade de Docteur en Informatique
par

Orleant EPAL NJAMEN

acceptée sur proposition du jury:

Rien n’est plus insondable
que le système de motivations
derrière nos actions…
— Georg Christoph Lichtenberg

À vous que j’aime si fort…

Remerciements

Merci!

Abstract

The abstract...

Keywords: event stream processing, complex event processing, distributed systems, smart grids, Quality of service .

Résumé

Mots clefs : event stream processing, complex event processing, distributed systems, smart grids, Quality of service.

Contents

List of Figures

List of Tables

1 Introduction

Event-based systems have gained importance in many application domains, such as management and control systems, large-scale data dissemination, monitoring applications, autonomic computing, etc. The components of an event-based system communicate by producing and consuming events, where an event is the notification that a happening of interest has occurred [VC02]. An event service mediates producers and consumers enabling loosely coupled communication among them. Producers publish events to the service, and consumers express their interest in receiving certain types of events by issuing event subscriptions. A subscription is seen as a continuous query that allows consumers to obtain event notifications [DGP07, BKK07]. The service is then responsible for matching received events with subscriptions and conveying event notifications to all interested consumers [MFP06]. Several academic research and industrial systems have tackled the problem of event composition. Techniques such as complex pattern detection [GJS92, GD94, CC96, PSB04], event correlation [Duo96, YB05], event aggregation [Luc02], event mining [AS95, GTB02] and stream processing [WDR06, DGP07, BKK07], have been used for composing events. In some cases event composition is done on event histories (e.g. event mining) and in other cases it is done dynamically as events are produced (e.g. event aggregation and stream processing).

The extension of event models towards more flexible and Quality of Services (QoS) oriented event models requires an analysis of the semantics that should be given to the events, and of their associated processing strategies. This requires dissociating the modeling of events and the application design and, the proposal of methods that allow to define event types independently of the management issues (detection, production, notification). In addition, event composition varies according to the different application requirements, in other words, the semantics of composition operators and in the different production policies. Therefore, we need to adapt the event models to smart grid characteristics and requirements. We must define event types, composition operators with their associated semantics, and composition algorithms to produce QoS oriented complex events.

2 Background and related works

3 Event Streams Composition Model

This chapter presents the building blocks to model an event composition system. It defines events, which are the entities manipulated by such a system. Then, it introduces event streams and stream based operations. Then it defines how to represent an event composition, with the related QoS dimensions.

Contents

3.1 Introduction

A vast number of event management models and systems have been, and continues to be proposed. Several standardization efforts are being made to specify how entities can export the structure and data transported by events. Models proposed in the literature range from simple models which describe the notification of signals, to evolved models which take into account various policies to manage and compose events. Existing models have been defined in an ad hoc way, notably linked to the application context (active DBMS event models), or in a very general way in middleware (Java event service, MOMs). Of course, customizing solutions prevents systems to be affected with the overhead of an event model way too sophisticated for their needs. However, they are not adapted when the systems evolve, cooperate and scale, leading to a lack of adaptability and flexibility. This chapter introduces the concepts of event, event type and event streams. It introduces stream based operations and presents how to define an event composition expression. It ends with the specification of QoS related to an event composition expression.

3.2 Event, event type, event streams

3.2.1 Event

An event is something that happens at a particular place and time and that is particularly significant, interesting or unusual. In computing systems the notion of event has a major importance since it provides a powerful abstraction to model dynamic aspects of applications. For instance, events can represent state changes in databases; signals in message systems; changes of existing objects or the creation of new objects in object-oriented systems; or "real-world" events such as the departure or arrival of vehicles. In event-driven programming an event is a message that indicates a situation that happened, such as a keystroke or a mouse click. In process control an event is an occurrence that happened and that has been registered. Examples are a purchase order, an email confirmation of an airline reservation, a stock tick message that reports a stock trade.

The literature proposes different definitions of an event. For example, in [MsSS97] an event is a happening of interest, which occurs instantaneously at a specific time. Another definition given by [RW97] characterizes an event as the instantaneous effect of the termination of an invocation of an operation on an object. According to the first definition, an event exists because some entity is interested in it; the second one defines events independently of any interested party. The second definition also subsumes an object model while the first one is neutral with respect to the model adopted for entities.

In this document we define an event in terms of a source named producer in which the event occurs, and a consumer for which the event is significant (figure 2.1). Thus, an event describes a fact, a situation, observable in a producer and significant for a consumer.

Producer
Event
Consumer

Figure 3.1 – Event

3.2.2 Event type

An event type is an expression that characterizes a class of significant facts (events) and the context under which they occur. Facts of the same nature are denoted by events that have the same type. Event types are characterized by an event model. An event model gives concepts and general structures used to represent event types.

According to the complexity of the event model, the event types are represented as sequences of strings [YBMM94], regular expressions [BGP$^+$94] or as expressions of an event algebra [CM94, GD94, CC96]. Other models represent an event type as a collection of parameters or attributes, allowing the type itself to contains implicitly the content of the message. This model is useful when we need to reason about events content. For example, *MeterAlarm(idMeter:string, voltage:real, current:real)* is an event type that represents a smart meter reading, where the current and voltage values observed are represented by attributes *voltage* and *current*.
Event composition, or specifically event streams composition supports the idea of performing operations on events. Some of those operations needs to access the events content (see Section 3.3). For that reason, we represent an event type as a collection of attributes:
$EventTypeE := Set < Attribute >$.
$Attribute :=< Name, Value >$.

Our definition of an event type includes the attributes presented in Table 3.1. The *typeName* attribute refers to the name of the event type. The *producerID* attribute refers to the id of the entity that produced the event occurence. The *detectionTime* attribute refers to the time at which the event occurence has been detected by a source. The *productionTime* attribute refers to the time at which the event has been produced (as a result of processings on others events) by an event-processing unit. The *notificationTime* attribute refers to the time at which the event is notified to interested consumers. The *receptionTime* attribute refers to the time at which an interested consumer receives the event. The context (*context* attribute) of an event type defines all the attributes that are particular to this event type. They represent the others data manipulated by the producer, which are relevant to this event type. An event instance (or simply event) is an occurrence of an event type.

3.2.3 Simple event type, complex event type

Event types can be classified as simple event types that describe elementary facts, and complex event types that describe complex situations by event combinations (see Figure 3.2).

Name	Type
typeName	String
producerID	String
detectionTime	Number
productionTime	Number
notificationTime	Number
receptionTime	Number
context	Set<Attribute>

Table 3.1 – Event types attributes

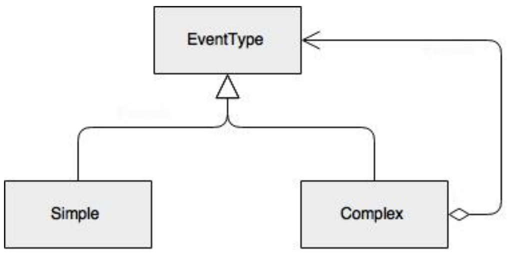

Figure 3.2 – Classification of event types

Simple event type

Event producers produce instances of a simple event type. Simple event instances are not generated as a result of processing others events.

Complex event type

Complex event types represent situations (relevant or critical) that can be inferred from the occurrences of others events. Complex events are produced by processing other events. Event processing operators are defined in Section 3.3.

3.2.4 Event streams

An event stream is a continuous, append-only sequence of events. We note $Stream(s, T)$ the stream of events of type T generated by the source s. If S is a set of sources, then $\{\bigcup stream(s, T), s \in S\}$ defines a stream of events of type T, denoted $Stream(T)$.

3.3 Stream based operators

Complex events are produced by composing event streams. Event stream composition operators specify the types of operations that can be performed on event streams. An event stream composition operator op takes one or many input event streams of a given type, and produces an output stream of a given type (see Figure 3.3).

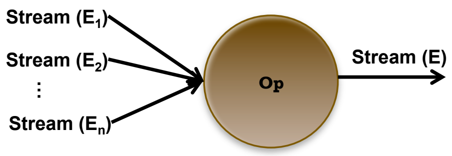

Figure 3.3 – Operator synopsis

Event stream composition operators considered here are : filter, logic operators, sequence, window operators, aggregation operators, selection operator and flatten.

3.3.1 Filter

The *filter* operator is denoted $filter(ES_i, P)$. It receives one input stream ES_i and a predicate P. The filter operator produces an event e in the output event stream each time an event e_i in the input stream satisfies the predicate P, that is if $P(e_i) = true$. The context of the output event e is the same as the context of the input occurrence e_i. That is, $e.context = e_i.context$.

3.3.2 Logic operators

Logic operators produce an event stream based on the occurrence of events in input streams. They are used to test specific pattern of occurences of event types.

Disjunction

The disjunction operator is denoted $OR(ES_1, ES_2, \dots, ES_n)$. It produces events in the output stream at the occurrence of events in any of the input streams $ES_i, i \in \{1, 2, \dots, n\}$. An event e is produced in the output stream if an instance e_i, of any input stream ES_i occurs. The context of the complex event e contains all occurrences e_i that occur.

Conjunction

The conjunction operator is denoted $AND(ES_1, ES_2, \dots, ES_n)$. It produces events in the output stream at the occurrence of events in all of the input streams $ES_i, i \in \{1, 2, \dots, n\}$. An event e is produced in the output stream if events $e_1, e_2, \dots, e_n$ occur respectively in input streams

$ES_1, ES_2, \ldots, ES_n$, regardless their occurrence order. The context of the complex event e contains all events $e_1, e_2, \ldots, e_n$.

3.3.3 Sequence

The sequence operator is denoted $SEQ(ES_1, ES_2, \ldots, ES_n)$. It receives two or more input event streams $ES_1, ES_2, \ldots ES_n$. The sequence operator produces a complex event e in output stream each time instances $e1$ in ES_1, e_2 in ES_2,..., e_n in ES_n are received in the specified order. Then, sequence denotes that $\forall i$, occurrence e_i "is received before" occurrence e_{i+1}. This implies that the reception time of event $e_1.receptionTime < e_2.receptionTime < \ldots < e_n.receptionTime$. The context of the complex ec contains all events e_1 and $e_2, \ldots, e_n$.

3.3.4 Window operators

Window operators partition an event stream into finite parts of the original event stream. Let's consider an input stream $Stream(E)$. We denote by $Stream_f(E)$, a finite part of $Stream(E)$. A window operator applied on $Stream(E)$ results in a stream of finite streams, which we denote $Stream(Stream_f(E)) = \{ES_{f,1}, ES_{f,2}, ES_{f,3}, ...\}$. The way each finite stream is constructed depends on the window specification, which can be *time-based* or *tuple-based*.

Time based windows

Time based windows define windows using time intervals.

- **Fixed windows**: $win : within(t_b, t_e, ES)$. Defines a fixed time interval $[t_b, t_e]$. The output stream contains a single finite event stream $ES_{f,1}$ such that $\forall e \in ES, e \in ES_{f,1}$ iff $t_b \le e_i.receptionTime \le t_e$.

- **Landmark windows**: $win : since(t_b, ES)$. Defines a fixed lower bound time tb. The output stream is a sequence of finite event streams $ES_{f,i}, i \in \{1, 2, \ldots, n\}$, such that each $ES_{f,i}$ contains events e_i received since the time lower bound t_b. That is, $\forall i, e \in ES_{f,i}$ iff $t_b \le e_i.receptionTime$.

- **Sliding windows**: $win : sliding(t_w, t_s, ES)$. Defines a time duration t_w and a time span t_s. The output stream is a sequence of finite event streams $ES_{f,i}, i \in \{1, 2, \ldots, n\}$, such that each $ES_{f,i}$ contains events from stream ES produced during last t_w time units. The finite event streams in the sequence are produced each t_s time unit. That is, if $ES_{f,i}$ is produced at time t, then $ES_{f,i+1}$ will be produced at time $t + t_s$.

Size bounded windows

A size bounded window defines the number of events for each window. We distinguish between size fixed windows and sliding size fixed windows.

- **Size fixed windows**: $win : batch(nb, ES)$. Specifies a fixed size *nb* of each finite stream. The output stream is a sequence of finite event streams $ES_{f,i}, i \in \{1,2,\ldots,n\}$, such that each finite event stream $ES_{f,i}$ contains nb most recent events from ES, and are non-overlapping. If we consider the event stream $ES = \{e_1, e_2, e_3, e_4, e_5, e_6, \ldots\}$, $win : batch(3, ES)$ will result in finite event streams $\{ES_{f,1}, ES_{f,2}, ...\}$ such that $ES_{f,1} = \{e_1, e_2, e_3\}$, $ES_{f,2} = \{e_4, e_5, e_6\}$, and so on.
- **Moving fixed size windows**: $win : mbatch(nb, m, ES)$. Defines a fixed size nb of each finite stream, and a number of events m after which the window moves. The output stream is a sequence of finite event streams $ES_{f,i}, i \in \{1,2,\ldots,n\}$, such that each $ES_{f,i}$ contains nb most recent events from ES. $ES_{f,i+1}$ is started after m events are received in $ES_{f,i}$ (moving windows). As result, an event instance may be part of many finite event streams. This is the case if $m \leq nb$. For example, if we consider windows of size nb=3 moving after each m = 2 events, that is $win : mbatch(3, 2, ES)$, the event stream $ES = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, \ldots\}$ will be partitioned into finite event streams $\{ES_{f,1}, ES_{f,2}, ES_{f,3}, \ldots\}$ such that $ES_{f,1} = \{e_1, e_2, e_3\}$, $ES_{f,2} = \{e_3, e_4, e_5\}$, $ES_{f,3} = \{e_5, e_6, e_7\}$, and so on.

3.3.5 Aggregation operators

Applied on a stream of finite streams $Stream(Stream_f(E)) = \{ES_{f,1}, ES_{f,2}, ES_{f,3}, ...\}$, an aggregator operator $aggregate(attr, Stream(Stream_f(E_i)))$ computes for each finite stream $ES_{f,i}, i \in \{1,2,\ldots,n\}$, a complex event *e* which carry (within its context) the aggregated value of the attribute *attr* over events occurrences in $ES_{f,i}$. The output is a stream of such complex events. Aggregates operators include *max, min, count, avg* and *sum.*

- The max operator compute the maximum value of the attribute over the event instances in $ES_{f,i}$.
- The min operator compute the minimum value of the attribute over the event instances in $ES_{f,i}$.
- The $count$ operator compute the number of event instances in $ES_{f,i}$ with the specified attribute.
- The avg operator compute the average value of the attribute over the event instances in $ES_{f,i}$.
- The sum operator compute the sum of the attribute's values over the event instances in $ES_{f,i}$.

3.3.6 Selection operators

A selection operator takes as input a stream of finite streams $\{ES_{f,1}, ES_{f,2}, ...\}$ and produce as output an event stream $e_1, e_2, \ldots$ such that for all i, each e_i is a selection of a particular event from $ES_{f,i}$. The choice of that particular occurrence depends on the selection operator:

- **First occurrence**: $first(Stream(Stream_f(E)))$.
  For each finite stream $ES_{f,i}, i \in \{1, 2, \ldots, n\}$, selects the first event occurrence. For example, if we apply the first operator to the stream of finite streams $\{\{e_1, e_2\}$, $\{e_3, e_4, e_5, e_6\}$, $\{e_7, e_8\}, \ldots\}$, we obtain as output the stream $\{e_1, e_3, e_7, \ldots\}$.

- **Last occurrence**: $last(Stream(Stream_f(E)))$.
  For each finite stream $ES_{f,i}, i \in \{1, 2, \ldots, n\}$, selects the last event occurrence. For example, if we apply the last operator to the stream of finite streams $\{\{e_1, e_2\}$, $\{e_3, e_4, e_5, e_6\}$, $\{e_7, e_8\}, \ldots\}$, we obtain as output the stream $\{e_2, e_6, e_8, \ldots\}$.

3.3.7 Flatten

Flatten operator takes as input a stream of finite event streams $\{ES_{f,1}, ES_{f,2}, ES_{f,3}, ...\}$ and produce as output a stream of events constructed by concatenating events in $ES_{f,1}, ES_{f,2}, ES_{f,3}$, and so on. For example, if we consider the stream of finite streams $\{\{e_1\}, \{e_2, e_3\}, \{e_4, e_5, e_6\}, \ldots\}$, the flatten operator produces the output stream $\{e_1, e, e_2, e_3, e, e_4, e_5, e_6, e, \ldots\}$ where e is a special "empty" event that indicates the end of each finite stream in the output stream.

3.4 Selection policy

There are ambiguous situations where an operator has to select between many event instances in input streams in order to produce a complex event. For example, consider the situation depicted in Figure 3.4 where the *and* operator is performed on two event streams A and B. Suppose that two event occurrences a_1 and a_2 are received from input stream A at time t_1 and $t_2 > t_1$ respectively. According to the operator definition, a complex event has to be produced in the output stream when occurrences from input streams A and B are received. So, at time t_2, there is nothing produced in the output. Now let's suppose that an occurrence b_1 is received from input stream B at time $t_3 > t_2$. Now, the *and* operator should produce a composite event in the output. For that, it must be specified which occurrence between a_1 and a_2 should be considered for the construction of the complex event. It is the goal of selection policies, to specify the events to be selected in such situations. The event selection policies are classified in recent, chronologic, continuous and cumulative.

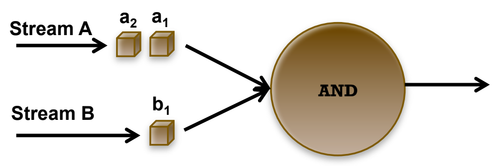

Figure 3.4 – Example situation where a selection policy should be applied

Recent Only the newest event occurence is selected. In the example depicted in Figure 3.4, the event a_2 is selected between a_1 and a_2.

Chronologic Only the oldest event occurence is selected. In the example depicted in Figure 3.4, the event a_1 is selected between a_1 and a_2.

Continuous All the event occurences are selected. In the example depicted in Figure 3.4, the event a_1 and a_2 are selected. As result, two complex events are produced in the output stream, with event parameters $\{a_1, b_1\}$ and $\{a_1, b_2\}$ respectively.

Cumulative All the event occurrences are accumulate in order to produce the complex event. In the example depicted in Figure 3.4, the event b_1 is combined with both events a_1 and a_2.

3.5 Event composition graph

A complex event is specified as an *event composition graph* (see Figure 3.5). An event composition graph identifies the event streams to be processed with the operation chain to be performed on them. In a more formal way, an event composition graph is a direct acyclic graph (DAG) $G =< \mathscr{N}, \mathscr{S} >$, where $\mathscr{N} = \{P \cup O\}$ consists in a set of terminal nodes P and a set of non-terminal nodes O, and a set of edges $\mathscr{S}$. Terminal nodes P represent input stream producers and non-terminal nodes O represent stream based operators. The edges $\mathscr{S}$ represent event streams. From an end user point of view, an event composition graph can be specified using either a graphical user interface, or programmaticaly.

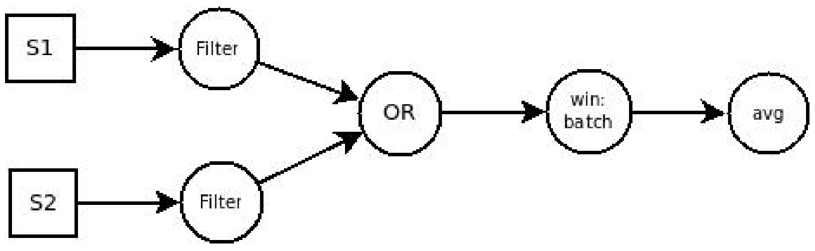

Figure 3.5 – Event composition graph

3.6 Quality of service

Event streams composition in smart grids face many challenges that are due to QoS that have to be guaranteed. This section first discusses such QoS requirements and then, presents how they are modeled as QoS expressions.

3.6.1 QoS dimensions

The quality of serice dimensions that we identify as relevant for smart grids are:

- *Latency.* Once detected, events may have to be notified as fast as possible to consumers. Such a timing constraint is even critical in some applications. For example, in the common practice for power device protection, the circuit breaker must be opened immediately if the voltage or current on a power device exceeds the normal values. The notification latency of an event is the time elapsed between its production and its notification to interested consumers.
- *Event priority.* Event priority defines a priority order between events. In some contexts, there may exist priorities between events that have to be captured by the event processing runtime. For example in a smart grid, alarm events are generally higher priority than events that report energy consumption. Events that have a higher priority have to be processed and notified earlier than less priority events.
- *Memory occupation.* Different devices may have different available memory capacities. To adapt the event processing to the memory capacity of each device, it must be a way to specify the maximum memory occupation incurred by an event processing unit at the execution time. The memory occupation constraint gives an upperbound on the number of events that an event processing unit can maintain in its main memory at execution time.

3.6.2 QoS expression

Let's consider three domains $D_{latency}$, $D_{priority}$, D_{memory}, corresponding to the QoS dimensions latency, event prioity and memory occupation respectively. Those domains are described below:

- $D_{latency} \subseteq R+$, as the latency corresponds to a time delay (an expected value for measuring time belongs to the set of positive real numbers). The domain inherits the properties of R+ (e.g. the operators of comparison). Best is shortest for latencies, the latency is bounded with the comparison operator "less than" (<) and "less or equal to" (≤).
- $D_{priority} \subseteq N^*$. An event type is associated to a priority level. The lower is the priority level, the higher is the priority associated to the event. The priority level varies according to the event type. The domain inherits the properties of N (e.g. the operators of

comparison). The priority is heavily restricted bounded, thus “equal to” (=) is the only accepted comparison operator.

- $D_{memory} \subseteq N$, as the memory corresponds to a quantity. The domain inherits the properties of N. The memory occupation is associated to an upper-bound that indicates the maximum memory available on a processing device. This is denoted by the comparison operator “less or equal to” ($\leq$).

Let us assume D the set of the QoS domains that we considered, that is $D = D_{latency} \bigcup D_{priority} \bigcup D_{memory}$. Given a domain D_Q, we assume a function *name(D_Q)* that returns its name, a function *operator(D_Q)* that returns the set of related operators, and a function *value(D_Q)* that returns the set of included values. For instance, let us consider the domain $D_{priority}$, thus:

- *name($D_{piority}$)* = event priority,
- *operator($D_{priority}$)* = equal to (=) ,
- *value($D_{latency}$)* = N^*, this is the set of all positive integer numbers.

Let also consider that $\mathscr{L}$ is the set of available processing devices.

Atomic QoS expression

An atomic QoS expression is of the form (d, θ, v) or , (d, θ, v, l) where

- d denotes a domain $D_Q, D_Q \in D$,
- $\theta \in$ *operator(D_Q)*,
- $v \in$ *value(D_Q)*,
- $l \in \mathscr{L}$ is a processing device.

For instance, the atomic QoS (latency, ≤, 2000) specifies that the latency for notifying an event must be less or equal to 2000 ms, assuming that the time unit is the millisecond. In the same way, the atomic QoS (memory, ≤, 1000, device1) specifies that the memory occupation on the device “device1” must be less or equal to 1000 Mo, assuming that the unit for the memory is the megabyte.

Complex QoS expression

A complex QoS expression specifies multiple QoS criteria. It is defined inductively as follows.

1. An atomic QoS is a complex QoS expression.

2. If QC_1 and QC_2 are complex QoS expressions then $QC_1 \wedge QC_2$ is a complex QoS expression.

The QoS expression (latency, ≤, 2000) ∧ (event priority, =, 1) specifies that the latency must be less than or equal to 2000 ms, and in addition, the highest priority level (i.e. priority = 1) is required.

3.7 Conclusion

In this chapter, we presented a model for event streams composition. We focused on the definition of concepts that are manipulated by an event stream composition system. First, we presented the concepts of event, simple and complex event types, and event streams. Then, we presented operators applicable to event streams in order to produce complex events, with their associate selection policies. After that, we presented how stream based operators can be combined together to define the event composition logic associate to a complex event. And finally, we presented some QoS dimensions applicable to an event composition, and how they can be specified using QoS expressions.

In the next chapter, we will go one step forward, by presenting how the presented model can be leveraged to define an approach for distributed event streams composition that deals with QoS.

4 Event Processing Network

French Summary

Contents

A.1 Introduction

[BGP+94] M L Bailey, Burra Gopal, Michael Pagels, L L Peterson, and Prasenjit Sarkar. PATHFINDER: A Pattern-Based Packet Classifier. (JANUARY 1994):115–123, 1994.

[CC96] Christine Collet and T. Coupaye. Primitive and Composite Events in O2 Rules. In *Actes des 12ièmes Journées Bases de Données Avancées*, 1996.

[CM94] S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases, 1994.

[GD94] S. Gatziu and K.R. Dittrich. Detecting composite events in active database systems using Petri nets. *Proceedings of IEEE International Workshop on Research Issues in Data Engineering: Active Databases Systems*, 1994.

[MsSS97] Masoud Mansouri-samani, Morris Sloman, and Morris Sloman. Gem - a generalised event monitoring language for distributed systems. *IEE/IOP/BCS Distributed Systems Engineering Journal*, 4, 1997.

[RW97] David S Rosenblum and Alexander L Wolf. A design framework for Internet-scale event observation and notification. *ACM SIGSOFT Software Engineering Notes*, 22(6):344–360, 1997.

[YBMM94] Masanobu Yuhara, Brian N Bershad, Chris Maeda, and J Eliot B Moss. Efficient Packet Demultiplexing for Multiple Endpoints and LargeMessages. In *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, page 13, 1994.

Bibliography