Noname manuscript No.
(will be inserted by the editor)

Deployment Strategies for Distributed Complex Event Processing

Gianpaolo Cugola · Alessandro Margara

Received: July 19th, 2012 / Accepted: Spetember 4th, 2012

Abstract Several *Complex Event Processing* (*CEP*) *middleware* solutions have been proposed in the past. They act by processing primitive events generated by sources, extracting new knowledge in the form of composite events, and delivering them to interested sinks.

Event-based applications often involve a large number of sources and sinks, possibly dispersed over a wide geographical area. To better support these scenarios, the CEP middleware can be internally built around several, distributed *processors*, which cooperate to provide the processing and routing service.

This paper introduces and compares different *deployment strategies* for a CEP middleware, which define (*i*) how the processing load is distributed over different processors and (*ii*) how these processors interact to produce the required results and to deliver them to sinks.

Our evaluation compares the presented solutions and shows their benefits with respect to a centralized deployment, both in terms of network traffic and in terms of forwarding delay.

Keywords Complex Event Processing · Distributed Processing · Event Based Systems · Routing

Gianpaolo Cugola
Dipartimento di Elettronica e Informazione (DEI), Politecnico di Milano,
Piazza Leonardo Da Vinci, 32 - 20133 Milan, Italy
E-mail: cugola@elet.polimi.it

Alessandro Margara
Dipartimento di Elettronica e Informazione (DEI), Politecnico di Milano,
Piazza Leonardo Da Vinci, 32 - 20133 Milan, Italy
E-mail: margara@elet.polimi.it

1 Introduction

Several complex systems operate by observing a set of *primitive events* that happen in the external environment, interpreting and combining them to identify higher level *composite events*, and finally sending the notifications about these events to the components in charge of reacting to them, thus determining the overall system's behavior. The general architecture of such event-based applications is shown in Figure 1. At the peripheral of the system are the *sources* and the *sinks*. The former observe primitive events and report them, while the latter receive composite event notifications and react to them. At the center of the system is the *Complex Event Processing* (*CEP*) *middleware* (or *system*), which is responsible for processing and routing events from sources to interested sinks. It operates by interpreting a set of *event definition rules*, which describe how composite events are defined from primitive ones.

Event-based applications usually involve a large number of sources and sinks, possibly dispersed over a wide geographical area. Typical examples are sensor networks for environmental monitoring [9] and financial applications, requiring a continuous analysis of stocks to detect trends [20]. To better support these scenarios, the CEP middleware can be internally built around several, distributed *processors*, connected together to form an *overlay network*, and cooperating to provide the processing and routing service.

This paper introduces and compares different *deployment strategies* for CEP. A deployment strategy defines (*i*) how the processing load is distributed over processors and (*ii*) how these processors interact to produce the required results and to deliver them to sinks.

The first aspect is often called *operator placement*: given a network of processors and a set of rules, it finds the best mapping of the operators defined in rules on available processors. Operator placement may pursue different goals, e.g., reduce the latency required to deliver notifications to interested parties, or minimize the usage of network resources. In the last few years, different solutions have been proposed for operator placement [26]. The problem is known to be extremely complex to solve, even for small instances with a reduced number of processors and rules. Accordingly, existing approaches are often based on approximated optimization algorithms or heuristics; moreover, they usually rely on a centralized decider, which collects all the relevant information about the network status and locally computes a solution to the problem. Only a few proposals have considered a decentralized algorithm for solving the operator placement [32].

On the other hand, operator placement is only part of the problem: when the processing effort is split among different processors, it also becomes necessary to precisely define the protocols that govern the interaction among them, specifying how rules and subscriptions are deployed, how primitive events are forwarded from the sources to the processors, and how composite events are finally delivered to sinks. These issues are usually not considered by existing CEP systems: most of them are based on a centralized deployment, in which all the processing is performed on a single machine (e.g. [8,2]). Even when

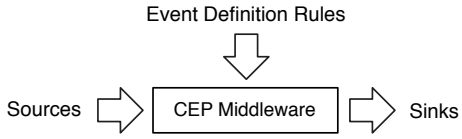


Fig. 1 The architecture of a CEP application

distributed processing is allowed, the communication among processors often requires manual configuration [5].

The solutions presented in this paper address all these problems and are explicitly tailored to large scale distributed scenarios: they take into account the topology of the processing network as well as the location of event sources and their generation rates. Moreover, they do not rely on a centralized decider: each processor autonomously decides which parts of the processing to execute locally and which parts can and should be delegated to other processors.

These strategies have been fully implemented as part of our T-Rex CEP middleware [16,17], including the protocols needed to organize processors into an overlay network, to deploy rules and subscriptions, and to deliver notifications from sources to sinks. We provide an evaluation and comparison of the strategies implemented: we measured significant advantages with respect to a centralized solution, both in terms of network traffic and forwarding delay. We also show how these two metrics are often conflicting, making it difficult to optimize both of them into a single strategy.

The rest of the paper is organized as follows: Section 2 introduces the event, subscriptions, and rule models we consider in this paper; Section 3 and Section 4 present our deployment strategies in details and evaluates their properties in an emulated network. Finally, Section 5 surveys related work and Section 6 provides some conclusive remarks.

2 Background

Events and subscriptions models. We assume events, i.e., things of interest, to occur instantaneously at some points in time. In order to be understood and processed, events are observed by sources and encoded in *event notifications* (or simply *events*). We assume that each event notification has an associated *type*, which defines the number, order, names, and types of the *attributes* that build the notification. Notifications have also a timestamp, which represents the occurrence time of the event they encode. In the following we assume that processors receive events in timestamp order: mechanisms to cope with out-of-order arrivals of events have been discussed in the past and can be adopted to ensure this property [39]. As an example, the air temperature in a given area at a specific time can be encoded into the following event notification:

```
Temp@10(area="A1", value=24.5)
```

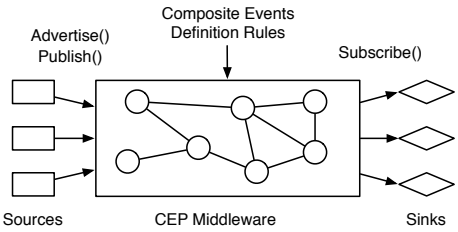


Fig. 2 The reference CEP middleware architecture used in this paper

Figure 2 summarizes the key architectural components and we consider in this paper and their interactions.

Throughput the paper, we represent sources as square boxes. As we will better show in Section 3, our deployment strategies rely on the knowledge of the type of events produced at each source. Accordingly, we ask sources to *advertise* the type of the primitive events they will publish. This builds a contract between the sources and the T-Rex system: only events whose type has been advertised will be processed[1].

The interests of sinks (represented as diamond boxes) are modeled through *subscriptions*, including a *type* and a set of *constraints*. A subscription s *matches* an event notification e if s has the same type as e and all the constraints expressed in s are satisfied by the attributes in e. As an example, the following subscription matches the previous temperature notification.

```
Temp(area="A1", value>=12)
```

Sinks can subscribe directly to primitive events published by sources, or to composite events. The latter are specified by a set of event definition rules, which describe how composite events are generated from primitive ones.

Rule definition language. In this paper, we consider rules written using the TESLA language [15]; since it includes all the typical operators used for CEP, we believe this choice will not impact the generality of our results.

We present the main features of TESLA through an example. Consider an environment monitoring application that processes information coming from a sensor network. Sensors notify their position, the air temperature they measure, and the presence of smoke and rain. Now, suppose a user wants to detect the presence of fire. She has to teach the system to recognize such critical situation starting from the raw data measured by sensors. Depending on the environment, the application requirements, and the user preferences, the presence of fire can be detected in many different ways. Here we present three possible definitions of the fire event and we use them to illustrate the operators supported by TESLA.

[1] The use of advertisements has been first introduced (and widely adopted) by publish-subscribe systems [11].

D1. There is fire when there is smoke and a temperature higher than 45 degrees in the same area of smoke (detected within 5 min from smoke). The fire notification has to embed the temperature actually measured.
D2. There is fire in presence of a temperature higher than 45 degrees and in absence of rain (in the last hour).
D3. There is fire when there is smoke and the average temperature in the last 5 min. in the same area is higher than 45 degrees.

Each TESLA rule has the following general structure:

```
Rule R
define    CE(att_1: Type_1, .., att_n: Type_n)
from      Pattern
where     att_1 = f_1, .., att_n = f_n
```

Intuitively the first two lines define a composite event from its constituents, specifying its structure, i.e., its type and the name and type of its attributes, and the pattern of events that lead to the composite one. Finally, the `where` clause defines the actual values of the composite event attributes using a set of functions `f_1,..,f_n`, which may depend on the arguments defined in `Pattern`.

If we carefully look at the definition of fire *D1* given above, we may notice that it is ambiguous. Indeed, it is not clear what happens if the `Smoke` event is preceded by more than one `Temp` event. In such cases we say that the *selection policy* of the rule is ambiguous [18,13]. TESLA is both very rigorous and very expressive about this point, allowing users to precisely define the selection policy they have in mind. By using the `each-within` operator, Rule R1 below:

```
Rule R1
define    Fire(area: string, measuredTemp: double)
from      Smoke(area=$a) and
          each Temp(area=$a and value>45)
          within 5 min. from Smoke
where     area=Smoke.area and measuredTemp=Temp.value
```

adopts a *multiple* selection policy: when a `Smoke` event is detected it notifies as many `Fire` events as they are the `Temp` events observed in the last 5 min. Other policies can be encoded by substituting the `each-within` operator with the `last-within` or the `first-within` operators, or with their generalized versions `n-last-within` and `n-first-within`. As an example, in presence of three temperature readings greater than 45 degrees followed by a `Smoke` event, Rule R1 would notify three `Fire` events, while adoption of the `last-within` would result in a single `Fire` notification, with the last temperature read. Besides these aspects, Rule R1 shows how in TESLA the occurrence of a composite event is always bound to the occurrence of a simpler event (`Smoke` in our example), which implicitly determines the time at which the new event is detected. This anchor point is coupled with other events (`Temp` in our example) through ad-hoc operators, like the `each-within`, which capture the

temporal relationships that join together events in sequences. Finally rule R1 also provides an example of parameterization, which is modeled in TESLA by using the variable `$a` to bind the values of attribute `area` in different events.

The second definition of `Fire` introduced above shows how time-based negations can be expressed in TESLA:

```
Rule R2
define    Fire(area: string, measuredTemp: double)
from      Temp(area=$a and value>45) and
          not Rain(area=$a) within 5 min. from Temp
where     area=Temp.area and measuredTemp=Temp.value
```

Beside time-based negations, TESLA allows interval-based negations, by requiring a specific event not to occur between other two events.

Finally, Rule R3 shows the use of aggregates (i.e., the function `Avg`) to express the `Fire` definition *D3*. Also aggregates can be time-based (as in this example) or interval-based.

```
Rule R3
define    Fire(area: string, measuredTemp: double)
from      Smoke(area=$a) and
          45 < $t=Avg(Temp(area=$a).value
          within 5 min. from Smoke)
where     area=Smoke.area and measuredTemp=$t
```

3 Deployment Strategies

This section presents the deployment strategies considered in this paper. For ease of exposition, we separately describe: (*i*) how processors organize themselves into one or more *processing trees*; (*ii*) how advertisements and subscriptions are forwarded among processors; (*iii*) how TESLA rules are recursively partitioned and distributed among available processors; (*iv*) how notifications are forwarded; (*v*) how traffic information can be used to limit event transmission.

For simplicity, the following discussion assumes that processors and links cannot fail (i.e., the communication is reliable). However, all the mechanisms described below can be extended to detect and react to failures.

Differently from some previously proposed solutions, which build a distributed complex event processing service on top of a publish-subscribe messaging network [34,33], our strategies strictly integrate the task of routing and forwarding events and the task of process and combine primitive events to detect and generate composite ones.

3.1 Building Processing Trees

We consider a set of processors P connected with each others at two levels. (*i*) On top of the physical network, is the overlay network. We do not impose any

condition on its structure: processors can be connected in any way, forming a generic graph. (*ii*) To simplify routing, our deployment strategies organize processors into one or more *processing trees* on top of the overlay network. More precisely, they use a processing tree to collect primitive events from sources (the leaves) and to filter and (partially) process them as they move toward the root of the tree, where composite events are finally detected. This enables incremental evaluation of rules at intermediate processors, which reduces the amount of information flowing along the tree, as well as the processing load at the root processor. Moreover, the same tree adopted for detecting a composite event *ce* is also used to distribute *ce* to all the interested sinks[2].

Since we want to minimize latency in collecting information from sources and delivering results to sinks, we build *Shortest Path Trees* using the link delay as a cost metric. In particular, to build the tree T_p rooted at processor p, p sends a special message $\texttt{CreateTree}_\texttt{p}$ to all its neighbors. When a processor p' receives such a message it behaves as follows.

- If p' receives the message for the first time, it marks the sender s as its father in T_p, sends an `ACK` message to s, and forwards the message to all its neighbors except s.
- If p' already received the message, it sends a `NACK` message to the sender s.

When a processor $p' \in P$ receives an `ACK`, it marks the sender as its child in T_p. p' obtains a complete knowledge about its children in the tree as soon as it receives an `ACK` or `NACK` message from all its neighbors. This protocol allows all processors to obtain local knowledge about T_p, i.e., their father and the set of their children.

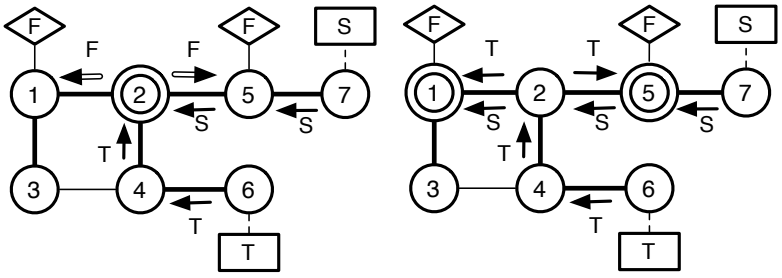
3.2 Single Tree vs. Multiple Trees

We consider two classes of deployment strategies. The first one organizes all processors into a single processing tree: in these strategies one processor ℓ is elected as the network leader. All events move from sources to ℓ going up along T_ℓ, and they are incrementally evaluated according to the TESLA rules deployed in the system. When they arrive to ℓ, the processing is complete, the corresponding composite events are generated and delivered to sinks by flowing back down T_ℓ.

The second class of strategies creates a tree for each sink and primitive events flow from the sources along these different trees. More precisely, if a sink s is interested in a composite event *ce*, all primitive events that contribute to *ce* move from their sources up along T_s. Processing is performed incrementally on each tree. When a composite event reaches the root of a processing tree there is no need to further distribute it, since the root of the tree coincides

[2] For simplicity, in the following discussion we assume sinks to be interested only in composite events. Primitive events can be easily handled using well known protocols for content-based routing (see for example [10,19]).

with the interested sink, and a different tree is created for each sink. On the one hand, this removes the need for spreading composite events once they have been detected. On the other hand, this approach duplicates primitive events by forwarding them over multiple trees.



(a) Single Tree (b) Multiple Trees

Fig. 3 Single Tree vs. Multiple Trees

To clarify the differences between the two classes of strategies consider Figure 3. It represents a sample overlay network with 7 processors. Now assume a single TESLA rule has been deployed in the system, which processes `Smoke` (`S`) and `Temp` (`T`) events to detect possible occurrences of `Fire` (`F`). There is a single source of `Smoke`, connected to processor 7, and a single source of `Temp`, connected to processor 6, while there are two sinks interested in `Fire`, connected to processor 1 and 5, respectively.

An example of single tree strategy is shown in Figure 3(a), where processor 2 is chosen as the network leader (links of T_ℓ are identified with thick lines). `Temp` and `Smoke` events flow from their sources up along T_2 (following single arrows in figure). Here they are combined to detect `Fire`; finally, `Fire` notifications are delivered from processor 2 to sinks 1 and 5 (following double arrows in figure).

An example of a multiple trees strategy is shown in Figure 3(b). One tree is built for each sink interested in `Fire` (i.e., 1 and 5). Sources forward events along both T_1 and T_5: for example 6 sends `Temp` events to 5 using the path 6-4-2-5, while 1 uses the path 6-4-2-1. Since the first three processors are in common, a single copy of each event notification is actually delivered along this sub-path. Similarly, source 7 delivers `Smoke` notifications both to 1 and 5. In this case, `Fire` events need not be forwarded, since they are autonomously detected at each sink.

3.3 Advertisement and Subscription Tables

As we said in Section 2, we require sources to notify the middleware about the set of event types they will publish, using `Advertisement` messages. We

show how advertisements are forwarded on a single processing tree: the same procedure is applied for each tree in the case of multiple trees strategies.

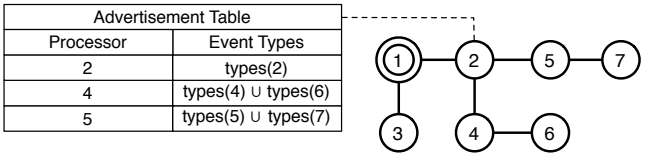


Fig. 4 Forwarding of advertisements

Figure 4 considers the tree T_1 rooted at processor 1: advertisements are forwarded from the sources up along T_1. Each processor saves all the event types contained in the advertisements coming from its descendants in an *advertisement table* (one for each defined tree). In Figure 4 we show the advertisement table of processor 2 by denoting the set of message types advertised by processor x as $types(x)$.

As in the case of advertisements, subscriptions move from sinks up to the root of the processing tree[3]; they are combined at each step and stored in a local *subscription table* at each processor. Subscriptions are then used to forward (composite) event notifications along the opposite path, from the root down to the sinks. When receiving an event notification e, each processor computes the set of subscriptions matching e to determine which children have to receive e.

3.4 Rule Deployment

To enable incremental evaluation of primitive events as they flow from the sources to the root of a processing tree, rules must be recursively partitioned into *partial rules*, moving in the opposite direction. The partitioning algorithm exploits the information stored in the advertisement tables of each processor.

In the case of a single tree strategy, rules are forwarded to the root of the tree, which is responsible for partitioning them and distributing the resulting "partial rules". In the case of multiple trees strategies, rules are accessible by every processor $p \in P$: they are actually stored at one or more processors, but p can download them on demand. When a sink s connected to processor p issues a subscription for composite events of type c, p retrieves all the rules that generate events of type c, and partitions and distributes them along its own processing tree T_p.

Partitioning TESLA rules. For the sake of clarity, we present the partitioning algorithm incrementally, through examples that progressively include all the features offered by TESLA. As a first example, consider Rule R4

[3] Forwarding of subscriptions is required only when a single tree strategy is adopted. Multiple trees strategies do not require subscriptions to be forwarded, since each sink is already at the root of its own tree, where composite events are detected.

Rule R4

```
define    CompEvent()
from      A() and last B() within 5 min. from A
          and last C() within 5 min. from B
          and last D() within 5 min. from C
          and last E() within 5 min. from D
```

and the processing tree T_1 shown in Figure 5. 1 is the root of the tree and there are three sources: 3 produces primitive events of type `A` and `B`; 4 produces events of type `C` and `E`; 5 produces events of type `D`. This information is stored in the advertisement table of processor 2; since advertisements are combined at each level of the tree, 1 has a single entry in its advertisement table, stating that all types of events (`A`, `B`, `C`, `D`, and `E`) come from 2.

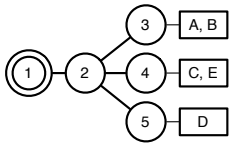


Fig. 5 Rule Deployment: an Example

In this scenario, Rule R4 is partitioned as follows: by looking at its advertisement table, 1 observes that 2 has all the information needed to correctly process the entire rule R4. Accordingly, it sends the complete rule to 2, delegating the entire processing (including the generation of composite events) to it. 2 looks at the advertisements coming from its children: none of them produces all the events required to process Rule R4. Accordingly, 2 remains the responsible for producing composite events. It splits Rule R4 into partial rules, and forwards them to processors 3, 4, and 5.

Partial rules include a pattern but do not generate composite events: they are used to limit as much as possible the number of event notifications that are forwarded up along the processing tree. When a processor p, responsible for processing a partial rule R', receives a set of primitive events PE that satisfy the pattern in R', it forwards all events in PE to its father.

Consider for example processor 3: its clients are the only sources for events of type `A` and `B`. To correctly process Rule R4, processor 2 does not need to receive all events of type `A` and `B`, but only those notifications of events `A` that are preceded by an event `B` in the previous 5 minutes; moreover, since the `last-within` operator is used, only the last `B` event before each `A` is relevant. Accordingly, 2 creates the following partial rule for 3.

```
A() and last B() within 5 min. from A
```

Similarly, 2 does not need to receive all events of type `C`, but only those preceeded by an event of type `E`. Accordingly, it creates and sends the following partial rule to 4.

```
C() and each E() within 10 min. from C
```

`C` and `E` are not contiguous elements in the sequence defined by Rule R4, but they are separated by event `D`, which is not produced by the sources of processor 4. Accordingly, the partial rule considers a timing constraints that sums the time limits between `C` and `D` together with the time limit between `D` and `E`. Similarly, the local knowledge of processor 4 is not sufficient to evaluate the single selection constraint on `E`; for this reason, the partial rule adopts the `each-within` operator, capturing all notifications of `E` followed by a `C` event within 10 minutes. Finally, processor 5 receives the following partial rule, asking for all events of type `D`[4].

```
D()
```

The partitioning algorithm described above is applied recursively: partial rules are split into other partial rules, until all sources have been reached.

Handling Events from Multiple Sources. We now describe how the partitioning algorithm changes when events are produced at multiple sources. Consider again Rule R4 and the processing tree T_1 represented in Figure 6.

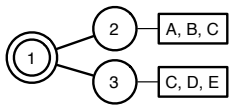


Fig. 6 Handling Events from Multiple Sources

At a first sight, it may be tempting to split Rule R4 into two partial rules, one involving `A`, `B`, and `C` (for processor 2), and one involving `C`, `D`, and `E` (for processor 3). However, neither 2, nor 3 receive all events of type `C`, so they may produce wrong results if they consider `C` during processing. It is processor 1 that is responsible for combining events of type `C` with the others. More in general, the detection of a certain type t of events may be delegated to a child c in the processing tree only when c is *the only one* processor that advertises type t. Accordingly, in the situation shown in Figure 6, Rule R4 is split into three partial rules. The first one, involving events `A` and `B`, is forwarded to 2:

```
A() and last B() within 5 min. from D
```

The second one, involving events `D` and `E`, is forwarded to 3:

```
D() and last E() within 5 min. from D
```

The last one, involving events of type `C`, is forwarded to both 2 and 3: `C()`.

Handling Parameters. TESLA rules may include parameters that bind the content of different primitive events. While partitioning a rule, if a parameter

[4] Notice that, while the example above only considers the types of events, our strategies also consider the constraints on the content of events expressed in rules during partitioning.

par involves only events that are captured by a partial rule R', than *par* is added to R'. Otherwise, if *par* involves events from different rules, it cannot be attached to any of them; in this case *par* is checked at a processor higher in the tree, where all involved primitive events are received.

Consider for example Rule R5 below and the two processing trees in Figure 7.

```
Rule R5
define    CompEvent()
from      A(v=$x) and last B(v=$x) within 5 min. from A
          and last C() within 5 min. from B
```

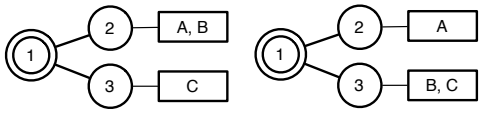


Fig. 7 Handling Parameters

In Figure 7(a), both events of type `A` and `B` come from the same processor 2. In this case the parameter can be added to the partial rule sent to 2, which becomes:

```
A(v=$x) and last B(v=$x) within 5 min. from A
```

On the contrary, in Figure 7(b) events of type `A` come from 2, while events of type `B` come from 3. Accordingly, the partial rule sent to 2 (i.e., `A()`) cannot refer to the parameter, and the same applies to the partial rule sent to 3:

```
B() and last C() within 5 min. from B
```

Processor 1 remains responsible for detecting Rule R5 and for checking the values of attribute `v` in events `A` and `B`.

Handling Negations. Similarly to parameters, negations can be attached to partial rules only if they include all the primitive events used to specify their time bound. Consider for example Rule R6 and the processing trees in Figure 8.

```
Rule R6
define    CompEvent()
from      A() and last B() within 5 min. from A
          and last C() within 5 min. from B
          and not D() between C and B
```

In Figure 8(a) both events `B` and `C` come from the same processor as the negated event `D`. Accordingly, we can include the negation inside the partial rule delivered to processor 3, which becomes:

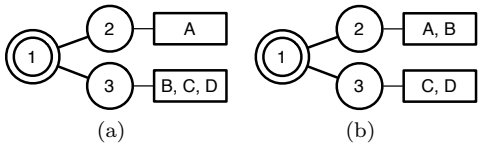


Fig. 8 Handling Negations

```
B() and last C() within 5 min. from B
and not D() between C and B
```

On the contrary, in Figure 8(b), events of type `B` and `C` are detected by two different processors. In this case, the negation cannot be included as part of the partial rule for 3. All events of type `D` have to be delivered to 1, which is responsible for processing the negation. Accordingly, 2 receives the following partial rule:

```
A() and last B() within 5 min. from B
```

while 3 receives two different partial rules, one for events of type `C` (i.e., `C()`), and one for events of type `D` (i.e., `D()`).

Computing Aggregates. In the deployment strategies described in this work, the aggregates included in a rule R are always computed by the processor responsible for generating composite events for R. In the case of rules involving complex aggregates over large volumes of data, it would be possible to modify this behavior by including special messages between processors that deliver the (partial) results of aggregates. Several solutions have been proposed in the past to optimize aggregates computation in distributed systems [22, 23], and some of them are explicitly tailored for event-based systesm [21].

We plan to explore this aspect in the near future. What we can anticipate is that in our experience processing does not introduce a significant delay [16, 17], especially if compared with the time required to forward information on a wide area network. Accordingly, we do not expect to get relevant benefits from incremental evaluation of aggregates. On the other hand, calculating aggregates in-network may contribute to limit the traffic.

3.5 Forwarding of Events

Primitive events are forwarded from the sources up along the processing trees. In the case of multiple trees strategy, event notifications are labeled with the set S_T of trees they are relevant for. When a processor p receives a primitive event e, it reads the set S_T; for each tree T in S_T, p extracts the set of rules (and partial rules) deployed, and uses them to process e. All produced results (either composite events, in the case of complete rules, or primitive events, in the case of partial rules) are delivered to the father of p in the tree T.

Notice that a single processor p may host multiple partial rules, all regarding the same tree T. In this case, it becomes possible for a primitive event e to satisfy the patterns of different partial rules at different time instants. To avoid duplicate transmissions, each processor keeps a history of already forwarded events for each tree it participates in. The size of this history is dynamically computed depending from the timing constraints expressed in rules, which in turn determine the maximum period of time in which events are stored for processing.

Pull-Based Forwarding. Consider the processing tree T_1 shown in Figure 9. 1 is responsible for detecting `Fire` starting from `Temp` (`T`) and `Smoke` (`S`), using Rule R1, defined in Section 2 and reported here for simplicity.

```
Rule R1
define    Fire(area: string, measuredTemp: double)
from      Smoke(area=$a) and
          each Temp(area=$a and value>45)
          within 5 min. from Smoke
where     area=Smoke.area and measuredTemp=Temp.value
```

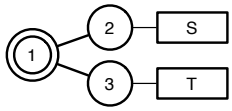


Fig. 9 Limitations of the Push-Based Forwarding

`Smoke` events are produced by 2, while `Temp` event are produced by 3. In certain cases it may happen to receive a large number of event notifications about high temperature, while `Smoke` notifications are much less frequent. In these cases, since processor 1 can produce `Fire` only when it receives both `Smoke` and `Temp`, the vast majority of events delivered by processor 3 would be discarded; forwarding them to 1 only wastes network resources.

Starting from these considerations, we introduced the concept of *pull-based forwarding* as opposed to the more common push-based approach we considered so far. In particular, every partial rule has an associated *mode* which can be either *push* or *pull*. A push partial rule requires the processor receiving it to promptly send all matching primitive events to its parent; on the contrary, a pull partial rule requires the processor to store matching events until the parent explicitly asks for them. In the example of Figure 9, processor 1 can decide to send the partial rule for events of type `T` in pull mode, and to ask for the delivery of stored events only after receiving events of type `S` from 2.

More specifically, the protocols used to decide the mode associated to partial rules and to ask for stored events (in case of pull rules) works as follows. When a processor p receives a rule R, it processes and partitions it as explained in the previous sections. Among the partial rules generated starting from R,

one is selected as the *master*, while the others are considered *slave*. The mode associated to the master is push, while slave partial rules have a pull mode. When processor p receives a set of events that satisfy the pattern expressed in the master, it sends an `Awakening` message to all children processing slave rules. Upon receiving this message, a child c, processing the slave partial rule R', starts sending events matching R' to p.

Consider again our example: assume that 1 chooses the partial rule about `Smoke` as master for Rule R3. When it receives a `Smoke` notification, it sends an `Awakening` message to 3. Processor 3 has a pull partial rule regarding `Temp` notifications: when it detects a high temperature, it stores the corresponding notification in a buffer, where it remains for the next 5 minutes (the timing constraints expressed in Rule R3). Upon receives an `Awakening` message, 3 sends all the event notifications stored (if any) to 1. After receiving these messages, 1 starts processing. In this case we say that the `Awakening` opens a *past window* of 5 minutes for the slave rule.

The way slave partial rules are processed depends from the position of the primitive events they have to detect inside the pattern specified in the original rule. As an example, consider again Figure 9, but now assume that the partial rule about `Temp` is elected as a master. Processor 2 does not have to store any `Smoke` notification. When it receives an `Awakening` message from 1 (meaning that a `Temp` notification has been detected) it starts to forward all the `Smoke` events it detects in the *following* 5 minutes. In this case we say that the `Awakening` opens a *future window* of 5 minutes for the slave rule. In the most general case, with TESLA rules that define more than one sequence, an `Awakening` can open both a past window and a future window.

When there are parameters shared by the master rule and one or more slave rules, the `Awakening` message can include the desired values for the shared attributes to further reduce the network traffic. In the previous example, if the partial rule about `Smoke` is selected as master, and a `Smoke` event is detected from an area a, this area is stored inside the `Awakening` message sent to 3, so that only stored `Temp` events coming from area a are forwarded.

Adaptive selection of masters. The right choice for the master vs. slave partial rules may strongly influence the performance of our protocol. To support this choice, we provide each processor with monitoring capabilities, to continuously analyze the network traffic. More specifically, a processor p stores, for each rule R it is responsible for, the number n of events it received in a given amount of time t from each partial rule R' originating from R. Periodically, p computes the *generation rate* of each partial rule R', $gr(R') = n/t$, and uses it to update its decision about the master, by choosing the rule with the lowest generation rate.

In presence of multiple partial rules deployed, the selection of an appropriate master becomes more complex, since a primitive event e can participate into more than one partial rules. If at least one of these rules is selected as master, notifications about e will be received in push mode. In this case, we do not only consider the generation rate of rules, but also the number of primitive

events that are already received in push mode since they are covered by other master rules.

In summary, the mechanism combining push and pull-based forwarding, coupled with this adaptive mechanism in the choice of which part of a rule to manage as push and which to treat as pull, results in the ability for our protocol to optimize composite event detection to the actual traffic, minimizing the number of event notifications that need to be forwarded.

4 Evaluation

As mentioned in the Introduction, all the strategies presented above have been implemented into our T-Rex system [16]. The result is a full fledged CEP middleware that uses the CDP processing algorithm described in [17] to interpret TESLA rules and that can be easily deployed in a distributed network, using either one of the strategies described so far.

To measure the performance of T-Rex in a controlled environment we created an emulated network using the Omnet++ simulator [40], on top of which we run several T-Rex processors, one for each emulated host. This allowed us to obtain a detailed analysis and comparison of the various strategies, while easily controlling a number of network parameters. Notice that the fact of running several T-Rex processors on the same physical machine, under an emulated network, does not influence the times we measured during our tests. Indeed, Omnet++ is a discrete event simulator for modelling communication networks. As such, it operates step by step; at each time a single "Omnet++ event"[5] is processed, being it one of those used to emulate the network or one of those that model CEP events flowing around. Consequently, at most one T-Rex processor may be active at each time, with full access to the resources of the hosting machine, an AMD Phenom II with 6 cores, each running at 2.8GHz. On the other hand, the use of an emulated network posed some limits on the complexity of the scenarios we were able to test. Indeed, if the T-Rex processors do not have to compete for the CPU, they have to share the same host memory (8Gb). In practice, we were able to emulate networks with up to 50 processors but we had to reduce the number of rules deployed into the system, as they strongly impact the memory consumption of each T-Rex processor.

In our tests we studied five different strategies:

1 `ST` performs distributed processing on a single tree.
2 `MT` performs distributed processing on multiple trees.
3 $\texttt{ST}_{PP}$ performs distributed processing on a single tree, adopting a hybrid push-pull communication protocol.
4 $\texttt{MT}_{PP}$ performs distributed processing on multiple trees, adopting a hybrid push-pull communication protocol.

[5] Here the term is used to indicate the internal events used in Omnet++ to model a complex network, not a T-Rex event.

5 `Centr` exploits a single processor, which receives primitive events, processes them, and delivers composite events to interested sinks. It is used as a baseline.

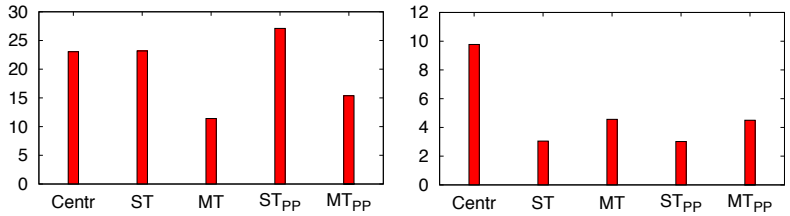
We refer to the strategies using a single processing tree (including `Centr`) as `ST` strategies, and to the strategies exploiting multiple processing trees as `MT` strategies.

Default scenario. To perform our tests, we defined a default scenario, and then we changed a number of parameters to explore their influence on the results we measured. Our default scenario includes 20 processors, each one connected with 5 others, on the average. The topology has been generated using Brite [29], with an average link delay of 5ms. Sources produce 120 different types of primitive events. Each type has the same probability of being produced, and the generation rates vary between 1 notification every 1000 seconds and 10 notifications per second, with exponential distribution. We deploy 100 TESLA rules, each one including a sequence of 3 events with time windows of 1 min, on the average. Each rule produces a different type of composite events, and the set of sinks connected with each processor are interested in 10 of them, on the average. To increase the traffic of events flowing in the network, we decided not to filter primitive events based on their content, but only based on their type. We will separately investigate how the use of content-based filtering impact on performance. Despite its simplicity, the default scenario allows us to capture all the aspects defined in our protocols. We present here the most interesting results we collected.

For each experiment, before running the actual tests phase, we execute a configuration phase, in which all the processors run the protocols to build the processing trees, and all rules and subscriptions are deployed. 30 seconds after sources begin publishing primitive events, we start monitoring the behavior of the system and we keep measuring until the most rare event has been published at least 100 times. With some type of events being published every 1000 seconds, this means that our tests span more than 24 hours of (simulated) time. We repeated every measure 10 times, using different seeds to generate the content of events and subscriptions; the 95% confidence interval was always below 1% of the measured value.

Figure 10 shows the results we measured in the default scenario. In particular, Figure 10(a) shows the delay for obtaining composite events, while Figure 10(b) shows the overall traffic generated by the system. The delay is computed as the difference between the time in which a sink receives a composite event e, and the time in which e occurs (i.e., the time in which the last primitive event necessary for its detection occurs). Since we are working in an emulated environment, we can measure this time without incurring in synchronization errors between processors.

By looking at Figure 10(a), we first observe a significant difference between `ST` and `MT` strategies, with the second class showing lower delays. This can be easily explained by remembering that `MT` strategies do not need to deliver composite event notifications after processing, thus eliminating the delay



(a) Average Delay (ms) (b) Average Traffic (KB/s)

Fig. 10 Default Scenario

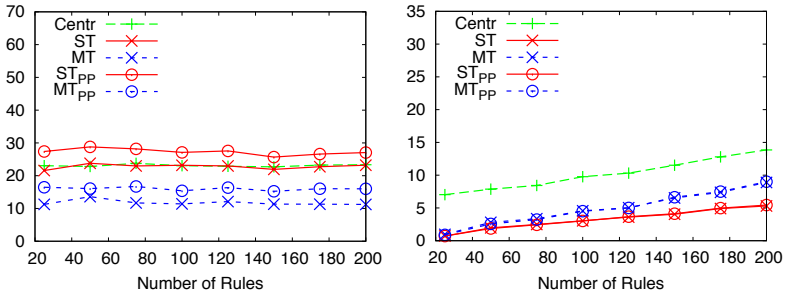
introduced in this phase. If we compare `ST` strategies, we observe that `Centr` and `ST` behave almost identically, while $\texttt{ST}_{\texttt{PP}}$ exhibits a slightly higher delay. This was expected, since pull notifications are not delivered immediately, but only when explicitly asked. The same is true for `MT` strategies.

By looking at Figure 10(b), we immediately observe that the `Centr` strategy generates significantly more traffic than the others. This means that, in our default scenario, the in-network filtering mechanisms introduced by distributed processing of events allows processors located near to sources to discard a large number of primitive events. Since our default scenario does not include content-based filtering of primitive events, the results shown in Figure 10(b) derive uniquely from the time-based filtering of events that do not contribute to valid sequences in the specified time windows. In the following, we will analyze how results change both when content-based filtering is added and when the size of windows is modified.

If we compare the four strategies that perform a distributed processing of events, we observe that `MT` strategies generate more traffic. On the one hand, these strategies need to forward primitive events along multiple trees; on the other hand, they do not need to forward composite events after detection. The results shown in Figure 10(b) demonstrate that in our default scenario the cost of duplicating primitive events over multiple trees overcomes the cost of forwarding composite events. Finally, we measure a marginal benefit in introducing pull notifications.

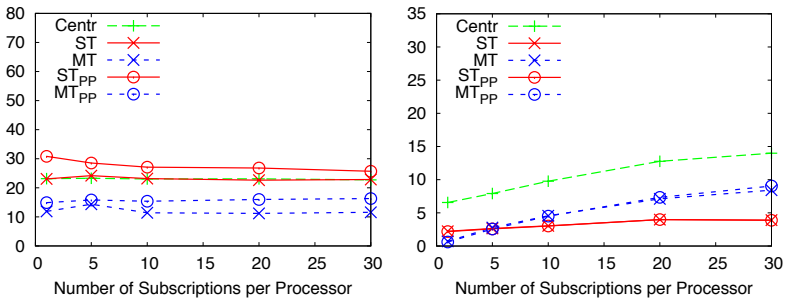
Number of rules. Figure 11 shows how results change when we increase the number of deployed TESLA rules. The delay perceived by sinks (Figure 11(a)) does not change significantly, despite increasing the number of rules also increases the complexity of processing. This is in line with our analysis of our processing algorithm [17]: processing times are in the order of tens of microseconds and are therefore dominated by communication delays. Figure 11(b) shows how the overall network traffic increases with the number of deployed rules. As rules increase in number, they attract more and more primitive events, forcing processors to forward them inside the overlay network. The traffic grows slightly faster in `MT` strategies than in `ST` ones.

Number of subscriptions. Figure 12 shows how results change with the number of subscriptions issued by sinks. The number of subscriptions deter-



(a) Average Delay (ms) (b) Average Traffic (KB/s)

Fig. 11 Number of Rules Deployed



(a) Average Delay (ms) (b) Average Traffic (KB/s)

Fig. 12 Number of Subscription per Processor

mines where composite events have to be forwarded after detection. Moreover, in the case of `MT` strategies, the number of sinks subscribing to a composite event e determines the number of processing trees used to detect e.

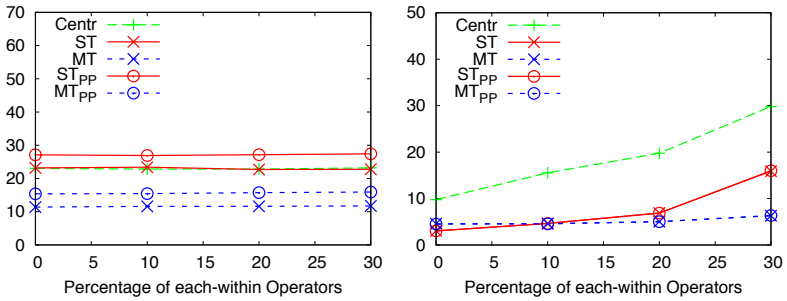
By looking at Figure 12(a), we observe that the delay measured is not influenced by the number of subscriptions. This means that a larger number of interested sinks for each produced event (and an increased number of processing trees, for `MT` strategies) does not significantly impact on the delay, which is dominated by the latency of network links, as already observed in the previous section.

As expected, the network traffic increases with the number of subscriptions (Figure 12(b)). It is interesting to note how `MT` strategies perform better than `ST` ones when the number of subscriptions is low (below 5 subscriptions per processor). Indeed, the main advantage of `MT` strategies is that they do not need to forward composite event notifications; at the same time they introduce an additional cost for moving primitive events over different trees. When the number of subscriptions is low, so is the number of processing trees adopted and the advantages of `MT` strategies come at a small cost. On the other hand, when the number of subscriptions increases, so does the number of process-

ing trees. Accordingly, with more than 5 subscriptions per processor, the ST strategies become more convenient in terms of network traffic.

Finally, having a large number of subscriptions negatively influences the $\mathtt{MT}_{\mathtt{PP}}$ strategy. With a large number of different processing trees, the pull mechanism does not produce benefits: in the communication between two processors, primitive events asked in pull mode for a tree may indeed be necessarily forwarded in push mode for another tree, thus preventing traffic optimizations.

Selection policy. In our default scenario all sequences were defined using a single selection policy, and more precisely using the `last-within` operator. We now investigate how results change when introducing a percentage of `each-within` operators (up to 30%) inside rules.



(a) Average Delay (ms) (b) Average Traffic (KB/s)

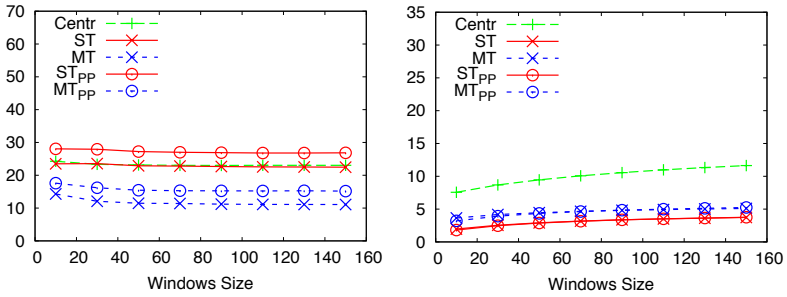
Fig. 13 Selection Policy

As in the previous experiments, the average delay registered by the sinks remains constant (Figure 13(a)). Once again, the differences in processing times are dominated by the latencies of network links. When considering the network traffic (Figure 13(b)), the adopted selection policy contributes in two ways: on the one hand, our recursive partitioning of rules becomes less efficient when it cannot exploit single selection constraints to filter primitive events; on the other hand, a multiple selection policy produces a larger number of composite events, which need to be forwarded to interested sinks.

By looking at Figure 13(b), we can conclude that the first aspect has only a marginal impact. This can be deduced by observing the traffic generated by MT strategies, which do not need to forward composite events after detection. Moving from 0 to 30% of `each-within` operators only produces a limited increase in network traffic. On the other hand, ST strategies significantly increase the network traffic when a multiple selection policy is adopted. As expected, MT strategies become more convenient as the ratio between composite events and primitive events increases, i.e., when a multiple selection policy is adopted.

Size of windows. Figure 14 shows how performance changes with the size of the time windows used inside rules. As in previous sections, the average delay observed by sinks (Figure 14(a)) does not change significantly. On the

other hand, the size of windows has a visible impact on the network traffic. As expected, all strategies exhibit higher traffic with larger windows; indeed, increasing the size of windows also increases the number of primitive events participating in valid patterns, and hence the number of composite events generated. Although it is not evident from the graph, the differences in terms of traffic between distributed `ST` strategies and the `Centr` strategy decrease. Indeed, larger windows reduce the possibility to filter out primitive events before they reach the root of the processing tree. On the contrary, when comparing `MT` strategies against `Centr`, we measure larger differences with larger time windows. Indeed, increasing the size of windows also increases the number of composite events generated, which advantages `MT` strategies. Finally, the use of pull notifications is more efficient with small windows, which increase the number of stored events that can be deleted because they violate timing constraints.

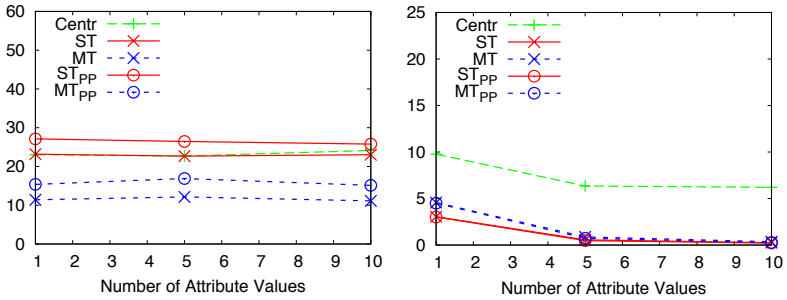


(a) Average Delay (ms) (b) Average Traffic (KB/s)

Fig. 14 Size of Windows

Content-based filtering. In this section we investigate how results change when the value of event attributes is used to filter out primitive events. When considering distributed processing of rules (as in the `ST` and `MT` strategies), this operation can be performed by processors close to sources, thus reducing the number of events that flow the network. We consider this analysis extremely important, since we expect real applications to make wide use of content-based selection selection of primitive events.

In the tests presented in Figure 15, each rule only selects primitive events that present a specific value for a given attribute. We changed the number of values that each attribute can assume, moving it from 1 to 10. A value of 1 means that all events are accepted for creating valid sequences (as in previous sections), while a value of 10 means that only 10% of primitive events is valid, and 90% of events can be immediately discarded. As expected, the average delay is not influenced by the presence of filtering (Figure 15(a)). On the other hand, increasing the number of attribute values reduces the network traffic (Figure 15(b)). As more primitive events are filtered, fewer

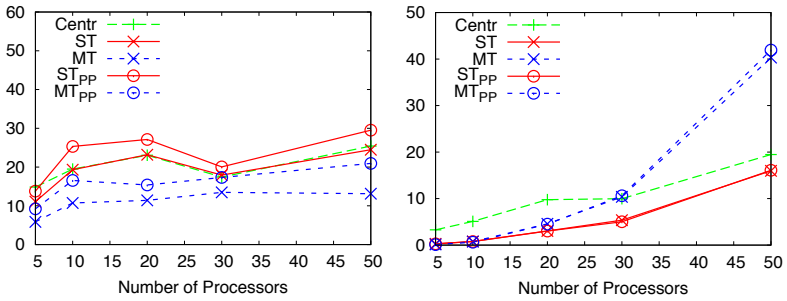


(a) Average Delay (ms) (b) Average Traffic (KB/s)

Fig. 15 Filtering of Primitive Events

composite events are generated. While this advantages all strategies, including `Centr`, distributed strategies obtain an additional benefit, as they can discard primitive events close to sources.

Number of processors. Figure 16 shows how performance changes with the number of processors in the overlay network, moving from 5 to 50. While increasing the number of processors, we also increase the number of connections, thus trying to limit the differences between topologies in terms of number of hops needed to forward a packet from one processor to another one. As for the default scenario, all network topologies have been generated using Brite. When increasing the scale of the network, we also increased the number of primitive event types, and the number of deployed rules. We believe this better represents realistic scenarios, in which the heterogeneity of events observed and generated grows with the scale of the network.



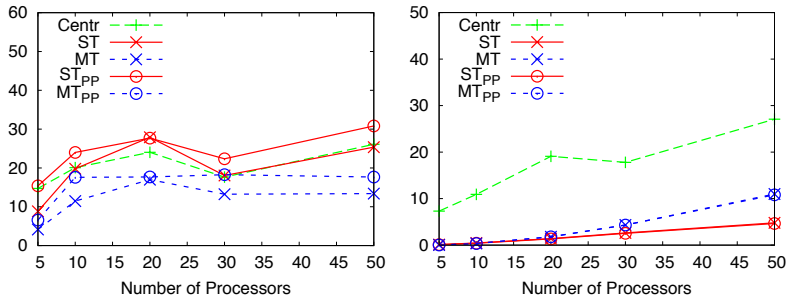
(a) Average Delay (ms) (b) Average Traffic (KB/s)

Fig. 16 Number of Processors

Despite we tried to limit the differences between topologies in terms of number of hops, the delay measured by sinks presents visible oscillations (Figure 16(a)). This behavior is more evident for `ST` strategies, which strongly rely

on the available overlay links when building the unique tree used to forward events.

Figure 16(b) shows how the network traffic increases with the number of processors. As we mentioned, indeed, we increased the number of primitive events and deployed rules together with the number of processors. `MT` strategies exhibit an interesting behavior: since they need to forward primitive events over different paths, the traffic they generate increases with the number of possible trees. This result suggests that `MT` strategies may not be convenient for large scale networks. However, it is worth mentioning that in our scenario we do not consider locality of events: primitive events may be generated at different (and distant) processors, and they can participate in several different rules. We expect real applications to present two forms of locality. (*i*) Locality of primitive events in rules, meaning that rules can be divided into classes, and primitive events participate only in a limited number of classes. As an example, events of type `Temperature` could participate in rules related to environmental monitoring, but they will not probably appear in rules related to inventory management. (*ii*) Locality of publishers: events that participate in the same rule are often produced by nearby sources. For example all events related to inventory management are produced in a single building.



(a) Average Delay (ms) (b) Average Traffic (KB/s)

Fig. 17 Number of Processors (with Locality)

To better understand the benefits of locality, we defined and tested a new scenario, in which we significantly increased the locality of primitive events (we defined classes of 10 rules, all using the same primitive events), and the locality of publishers. The results measured in this scenario are shown in Figure 17. Since we did not impose locality for sinks, the measured delay does not change significantly with respect to the previous scenario (see Figure 17(a)). On the other hand, Figure 17(b) shows a remarkable difference in the network traffic. While `ST` strategies are still more efficient, `MT` strategies now show similar results, with an average network traffic that is significantly lower than in the `Centr` strategy. To conclude, we can say that, while `MT` strategies suffer from the need of delivering primitive events over different paths, the presence of locality significantly mitigates the problem.

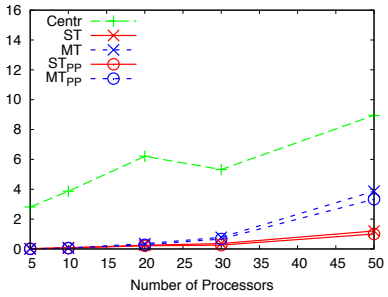


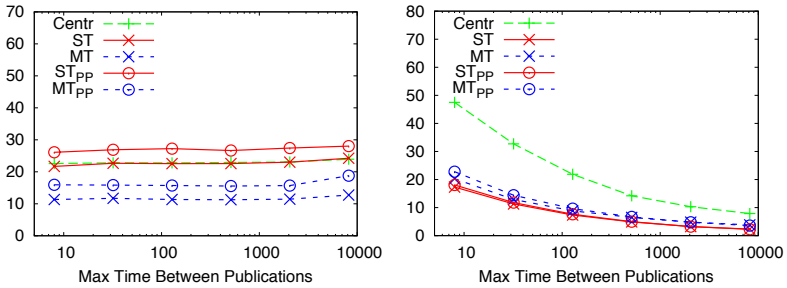
Fig. 18 Number of Processors (with Filtering)

Finally, we repeated our scalability tests introducing content-based filtering of primitive events. Figure 18 shows the network traffic measured when 90% of primitive events is filtered out based on the value of attributes. As in the case of locality, distributed processing becomes more efficient than centralized processing, with a significantly reduced network traffic. Moreover, the reduced number of primitive events flowing the network increases the advantages of pull notifications, especially with a large number of processors.

Publication rate. Figure 19 shows how performance changes with the publication rate. In particular, during our tests, we fixed a maximum publication rate of 10 events per second, while we changed the minimum publication rate, moving it from 1 event every 5 seconds to 1 event every 10000 seconds. Since we adopt an exponential distribution, this operation has only a minimal impact on the average publication rate. On the other hand, it has a great impact on the number of composite events generated, since certain event types become extremely rare. As expected, the average delay is not influenced by the publication rate (Figure 19(a)). When increasing the maximum time between publications, all strategies present a lower traffic (Figure 19(b)), since fewer composite events are generated. While this advantages `ST` strategies that need to forward composite events after detection, it produces a greater advantage in strategies performing distributed processing, which filter events near sources. Accordingly, the advantage of both `ST` and $\texttt{ST}_{\texttt{PP}}$ over `Centr` increases with the maximum time between publications.

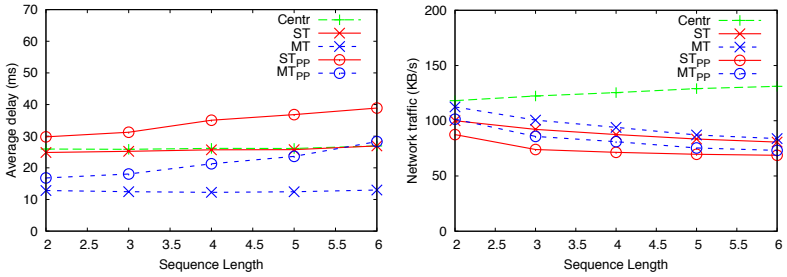
On the contrary, `MT` strategies are not influenced by the number of generated composite events, but only by the possibility of filtering out primitive events. As a result, the advantage of `MT` strategies over `Centr` in terms of generated traffic does not change significantly with the maximum time between publications. Finally, we observe how a small difference between publication rates makes it inconvenient to adopt mechanisms for pull notifications. Indeed, the number of primitive events forwarded in pull mode becomes small, and the cost of the mechanism overcomes the advantages.

A scenario for pull notifications. Our analysis so far did not show significant benefits from the use of a hybrid push-pull approach. In particular, the use of pull notifications introduces some additional delay for the delivery, while



(a) Average Delay (ms) (b) Average Traffic (KB/s)

Fig. 19 Publication Rate



(a) Average Delay (ms) (b) Average Traffic (KB/s)

Fig. 20 A scenario for pull notifications

providing a minimum advantage (if any) in terms of network traffic. We claim that this is mostly a result of using synthetic scenarios, where a lot of aspects are randomly chosen using a uniform distribution. We know there are scenarios where the push-pull approach must offer some advantages and we present one in this section. It differs from our default scenario in three main aspects: (*i*) each primitive event participates in a very small number of rules, such that if an event is processed in pull mode in a rule it is not processed in push mode in another; (*ii.*) it considers a (relatively) large network, with 50 processors, to increase the benefit of storing events instead of pushing them immediately toward a distant root; (*iii.*) it considers (relatively) small windows of 1s, to increase selectivity of rules.

Notice that, albeit built ad-hoc, this scenario is not uncommon in CEP applications as it reflects a medium to large scale situation in which, even when many rules are deployed, they process different types of primitive events. This may happen either because each rule considers a different aspect of the same system, or because the CEP infrastructure offers its services to multiple applications, each with its own rules and event types.

Figure 20 shows the results we measured in this scenario, while changing the length of sequences, i.e., the number of primitive event types captured by each rule. First, also in this case, the delay remains almost constant (Figure 20(a)),

with the exception of push-pull strategies, whose delay slightly increases with the length of sequences. Indeed, longer sequences increase the number of pull-based partial rules, which provide results only when asked, thus introducing delay. By looking at Figure 20(b) we first observe an advantage of distributed strategies with respect to `Centr` that increases with the length of sequences. Indeed, long sequences favor distributed filtering of primitive events, introducing a larger number of timing constraints between them. Most importantly, in this scenario we observe a reduction of about 15-20% of network traffic when introducing pull notifications, both in `ST` and in `MT` strategies. Moreover, the advantage remains almost constant with the length of sequences.

Final considerations. As we already observed in our previous experience [16, 17], when dealing with a CEP system the number of parameters that impact on performance is huge. It is even larger when distributed processing is taken into account: it becomes fundamental to consider the topology of the network, and the capabilities of links, but also *where* information is produced and consumed, as our experiments on locality demonstrates. To further extend our analysis of strategies, we plan to try real deployments.

However, the results we measured allow us to derive some interesting conclusions. First, distributed solutions provide significant benefits with respect to a centralized deployment in terms of network traffic, by filtering events close to sources. Moreover, `MT` solutions also reduce the forwarding delay, often considered the most significant metrics for event-based applications. Indeed, processing times contribute only marginally to the delay for delivering notifications, making `MT` strategies advantageous even if they duplicate processing over multiple trees.

A further consideration regards the strategies that include pull notifications. In most of the tests we performed they provide only limited benefits. However, as shown in Figure 20, there are scenarios in which the use of pull notifications can contribute in reducing the overall network traffic. This is a key metric in many application fields, where network bandwidth is a scarce resource (e.g., monitoring with wireless sensor network).

Finally, as a future work, we plan to test how the different strategies react to dynamic behaviors of sources and sinks. Indeed, we are also interested in understanding the benefits and the limitation of each strategy when applied to environments where clients come and go frequently and unpredictably. In such scenarios, it becomes relevant to study the overhead introduced for re-computing the processing trees and re-deploying the rules when the topology of the network changes.

5 Related Work

One of the aspects addressed by the deployment strategy of a CEP system is the operator placement problem, which specifies how the operators defined in rules are deployed on available processors. A good survey of existing solutions can be found in [26].

A fundamental consideration about the operator placement problem regards the kind of operators it addresses. By using the terminology introduced in [18], we notice that most existing operator placement algorithms are based on transforming languages. These languages define a sequence of transformations that incoming data has to pass through to produce the desired output: since these transformations are applied one after the other, it is easy to deploy them on different processors. On the contrary, detecting languages specify complex patterns, often involving timing constraints, which are not trivial to split.

Distributed detection of pattern has been first explored in [27], with a very simple language. An important contribution comes from [37], where the authors study how patterns can be rewritten for efficient distribution.

Existing solutions for the operator placement problem [7,4,27,1,25,32,44, 6,36,42,24,14] differ from each other on many aspects. (*i*) They start from different assumptions: some of them consider large scale networks of processors, while others assume clusters of colocated machines; some consider processors with heterogeneous computational resources, while others consider homogeneous processors; some of them assumes that single operators can be duplicated at different processors, if needed, while others do not allow duplications. (*ii*) They are designed for different goals: for example, minimize the delay required to produce results other, minimize the usage of network resources, or a combination of the two, or again minimize processing resource usage, and hence power consumption.

Since the problem is known to be NP hard, it is usually solved using approximated algorithm or heuristics. Most systems rely on a centralized decider, which collects all relevant information about the network status and locally computes a solution for the problem. Only a few proposals have considered a decentralized algorithm for operator placement. The deployment strategies proposed in this paper solve the operator placement problem in a distributed way, by recursively splitting rules at each processor; moreover, our push-pull approach exploits traffic information to alter the communication between processors.

Beside addressing the placement of operators, a complete deployment strategy also needs to precisely define the protocols that govern the interaction among processors, specifying how information is collected, processed, and finally delivered to interested sinks. These issues are often not considered in existing CEP systems: most of them are based on a centralized deployment, in which all the processing is performed on a single machine (e.g. [8,2]). Others define distributed processing, but are based on extremely simple languages if compared with TESLA, which do not capture all the needs of event-based applications [27]. Even when distributed processing is allowed, deployment often requires manual configuration [5].

It is worth mentioning that some remarkable example of automated distribution can be found among DSMSs [1,3].

Another interesting contribution is presented in [34,33], where the authors introduce a framework for complex event detection that works on top of exist-

ing publish-subscribe systems. While the work does not explicitly focus on the problem of operator placement, it enables distributed processing by translating all the processing rules into automata structures, whose constituent states can migrate for node to node, depending from the requirements of the system.

Finally, some of the protocols adopted in our strategies are inspired by work on distributed publish-subscribe systems and content-based routing [12].

Ortogonal to our strategies are the techniques used to ensure reliability in event-based systems [31] and to generate meaningful event notifications from the low level information collected by sources [43]. The latter problem is often addressed by modeling the uncertainty associated to collected information [41, 35]; in particular, when dealing with large scale distributed systems, it becomes of primary importance to consider the uncertainty associated with time and location of event occurrence [28,38,30]. We are currently working to integrate the strategies presented in this paper with our own model of uncertainty.

6 Conclusions

In this paper we introduced and compared different deployment strategies for distributed CEP. Given a network of processors, they precisely define the communication required to handle rule and subscription deployment, and to collect, process, and deliver event notifications. In these solutions, deployment is performed in a distributed way, with each processor autonomously taking decisions based on local knowledge about their neighbors. We analyze the different strategies and compare them against a centralized deployment.

Different aspects emerge from our analysis: first, processing delays are negligible if compared with typical network latencies. As a consequence, the most efficient strategies in terms of delay are those that choose the shortest paths to deliver events to sinks, even if this brings to duplication of processing. Second, distributed processing significantly reduces the network traffic with respect to a centralized solution, by filtering events close to their sources. Finally, we evaluated monitoring mechanisms that allow the middleware to automatically adapt to event generation rates, by applying a hybrid push-pull approach to deliver event notifications. Despite this solution provides only limited advantages in terms of network traffic, they can provide benefits in scenarios, like wireless sensor networks, in which network resources are significantly constrained.

Acknowledgment

This work was partially supported by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom.

References

1. Abadi, D.J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.H., Lindner, W., Maskey, A.S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.B.:

The design of the borealis stream processing engine. In: CIDR '05. ACM, Asilomar, CA, USA (2005)
2. Agrawal, J., Diao, Y., Gyllstrom, D., Immerman, N.: Efficient pattern matching over event streams. In: SIGMOD '08, pp. 147–160. ACM, New York, NY, USA (2008). DOI http://doi.acm.org/10.1145/1376616.1376634
3. Ahmad, Y., Berg, B., Cetintemel, U., Humphrey, M., Hwang, J.H., Jhingran, A., Maskey, A., Papaemmanouil, O., Rasin, A., Tatbul, N., Xing, W., Xing, Y., Zdonik, S.: Distributed operation in the borealis stream processing engine. In: SIGMOD '05, pp. 882–884. ACM, New York, NY, USA (2005). DOI http://doi.acm.org/10.1145/1066157.1066274
4. Ahmad, Y., Çetintemel, U.: Network-aware query processing for stream-based applications. In: VLDB '04, pp. 456–467. VLDB Endowment (2004)
5. Ali, M.: An introduction to microsoft sql server streaminsight. In: Proceedings of the 1st International Conference and Exhibition on Computing for Geospatial Research and Application, COM.Geo '10, pp. 66:1–66:1. ACM, New York, NY, USA (2010). DOI http://doi.acm.org/10.1145/1823854.1823929
6. Amini, L., Jain, N., Sehgal, A., Silber, J., Verscheure, O.: Adaptive control of extreme-scale stream processing systems. In: ICDCS '06, p. 71. IEEE Computer Society, Washington, DC, USA (2006). DOI http://dx.doi.org/10.1109/ICDCS.2006.13
7. Balazinska, M., Balakrishnan, H., Stonebraker, M.: Contract-based load management in federated distributed systems. In: NSDI '04, pp. 15–15. USENIX Association, Berkeley, CA, USA (2004)
8. Brenna, L., Demers, A., Gehrke, J., Hong, M., Ossher, J., Panda, B., Riedewald, M., Thatte, M., White, W.: Cayuga: a high-performance event processing engine. In: SIGMOD '07, pp. 1100–1102. ACM, New York, NY, USA (2007). DOI http://doi.acm.org/10.1145/1247480.1247620
9. Broda, K., Clark, K., 0002, R.M., Russo, A.: Sage: A logical agent-based environment monitoring and control system. In: AmI '09, pp. 112–117 (2009)
10. Carzaniga, A., Rosenblum, D., Wolf, A.: Design and evaluation of a wide-area event notification service. ACM Trans. on Comp. Syst. **19**(3), 332–383 (2001). URL citeseer.nj.nec.com/482106.html
11. Carzaniga, A., Rosenblum, D.S., Wolf, A.L.: Achieving scalability and expressiveness in an internet-scale event notification service. In: PODC '00, pp. 219–227. Portland, Oregon (2000)
12. Carzaniga, A., Rutherford, M.J., Wolf, A.L.: A routing scheme for content-based networking. In: INFOCOM '04. Hong Kong, China (2004)
13. Chakravarthy, S., Krishnaprasad, V., Anwar, E., Kim, S.K.: Composite events for active databases: Semantics, contexts and detection. In: Proceedings of the 20th International Conference on Very Large Data Bases, VLDB '94, pp. 606–617. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1994)
14. Cugola, G., Margara, A.: Raced: an adaptive middleware for complex event detection. In: ARM '09, pp. 1–6. ACM, New York, NY, USA (2009)
15. Cugola, G., Margara, A.: Tesla: a formally defined event specification language. In: DEBS '10, pp. 50–61. ACM, New York, NY, USA (2010). DOI http://doi.acm.org/10.1145/1827418.1827427
16. Cugola, G., Margara, A.: Complex event processing with t-rex. Journal of Systems and Software **85**(8), 1709 – 1728 (2012). DOI 10.1016/j.jss.2012.03.056. URL http://www.sciencedirect.com/science/article/pii/S0164121212000842
17. Cugola, G., Margara, A.: Low latency complex event processing on parallel hardware. Journal of Parallel and Distributed Computing **72**(2), 205 – 218 (2012). DOI 10.1016/j.jpdc.2011.11.002
18. Cugola, G., Margara, A.: Processing flows of information: From data stream to complex event processing. ACM Comput. Surv. **44**(3), 15:1–15:62 (2012). DOI 10.1145/2187671.2187677. URL http://doi.acm.org/10.1145/2187671.2187677
19. Cugola, G., Margara, A., Migliavacca, M.: Context-Aware Publish-Subscribe: Model, Implementation, and Evaluation. In: ISCC '09. IEEE Computer Society Press (2009)
20. Demers, A.J., Gehrke, J., Hong, M., Riedewald, M., White, W.M.: Towards expressive publish/subscribe systems. In: EDBT '06, pp. 627–644 (2006)

21. Eyers, D., Freudenreich, T., Margara, A., Frischbier, S., Pietzuch, P., Eugster, P.: Living in the present: on-the-fly information processing in scalable web architectures. In: Proceedings of the 2nd International Workshop on Cloud Computing Platforms, CloudCP '12, pp. 6:1–6:6. ACM, New York, NY, USA (2012). DOI 10.1145/2168697.2168703. URL http://doi.acm.org/10.1145/2168697.2168703
22. Jain, N., Kit, D., Mahajan, P., Yalagandula, P., Dahlin, M., Zhang, Y.: STAR: Self-tuning aggregation for scalable monitoring. In: VLDB'07 (2007)
23. Jain, N., Mahajan, P., Kit, D., Yalagandula, P., Dahlin, M., Zhang, Y.: Network imprecision: a new consistency metric for scalable monitoring. In: OSDI'08 (2008). URL http://dl.acm.org/citation.cfm?id=1855741.1855748
24. Khandekar, R., Hildrum, K., Parekh, S., Rajan, D., Wolf, J., Wu, K.L., Andrade, H., Gedik, B.: Cola: optimizing stream processing applications via graph partitioning. In: Middleware '09, pp. 1–20. Springer-Verlag New York, Inc., New York, NY, USA (2009)
25. Kumar, V., Cooper, B.F., Cai, Z., Eisenhauer, G., Schwan, K.: Resource-aware distributed stream management using dynamic overlays. In: ICDCS '05, pp. 783–792. IEEE Computer Society, Washington, DC, USA (2005). DOI http://dx.doi.org/10.1109/ICDCS.2005.69
26. Lakshmanan, G.T., Li, Y., Strom, R.: Placement strategies for internet-scale data stream systems. IEEE Internet Computing **12**(6), 50–60 (2008). DOI http://dx.doi.org/10.1109/MIC.2008.129
27. Li, G., Jacobsen, H.A.: Composite subscriptions in content-based publish/subscribe systems. In: Middleware '05. Springer-Verlag New York, Inc. (2005)
28. Liebig, C., Cilia, M., Buchmann, A.: Event composition in time-dependent distributed systems. In: Cooperative Information Systems, 1999. CoopIS '99. Proceedings. 1999 IFCIS International Conference on, pp. 70 –78 (1999). DOI 10.1109/COOPIS.1999.792159
29. Medina, A., Lakhina, A., Matta, I., Byers, J.: Brite: An approach to universal topology generation. In: MASCOTS '01, pp. 346–. IEEE Computer Society, Washington, DC, USA (2001)
30. Moody, K., Bacon, J., Evans, D., Schwiderski-Grosche, S.: Implementing a practical spatio-temporal composite event language. In: K. Sachs, I. Petrov, P. Guerrero (eds.) From Active Data Management to Event-Based Systems and More, *Lecture Notes in Computer Science*, vol. 6462, pp. 108–123. Springer Berlin / Heidelberg (2010). URL http://dx.doi.org/10.1007/978-3-642-17226-7_7. 10.1007/978-3-642-17226-7_7
31. O'Keeffe, D., Bacon, J.: Reliable complex event detection for pervasive computing. In: Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, DEBS '10, pp. 73–84. ACM, New York, NY, USA (2010). DOI 10.1145/1827418.1827429. URL http://doi.acm.org/10.1145/1827418.1827429
32. Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., Seltzer, M.: Network-aware operator placement for stream-processing systems. In: ICDE '06. IEEE Computer Society (2006)
33. Pietzuch, P., Shand, B., Bacon, J.: Composite event detection as a generic middleware extension. Network, IEEE **18**(1), 44 – 55 (2004). DOI 10.1109/MNET.2004.1265833
34. Pietzuch, P.R., Shand, B., Bacon, J.: A framework for event composition in distributed systems. In: Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware, Middleware '03, pp. 62–82. Springer-Verlag New York, Inc., New York, NY, USA (2003). URL http://dl.acm.org/citation.cfm?id=1515915.1515921
35. Ré, C., Letchner, J., Balazinksa, M., Suciu, D.: Event queries on correlated probabilistic streams. In: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, SIGMOD '08, pp. 715–728. ACM, New York, NY, USA (2008). DOI 10.1145/1376616.1376688
36. Repantis, T., Gu, X., Kalogeraki, V.: Synergy: sharing-aware component composition for distributed stream processing systems. In: Middleware '06, pp. 322–341. Springer-Verlag New York, Inc., New York, NY, USA (2006)
37. Schultz-Moeller, N.P., Migliavacca, M., Pietzuch, P.: Distributed complex event processing with query optimisation. In: DEBS '09. ACM, ACM, Nashville, TN, USA (2009)
38. Schwiderski-Grosche, S., Moody, K.: The spatec composite event language for spatio-temporal reasoning in mobile systems. In: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems, DEBS '09, pp. 11:1–

11:12. ACM, New York, NY, USA (2009). DOI 10.1145/1619258.1619273. URL http://doi.acm.org/10.1145/1619258.1619273
39. Srivastava, U., Widom, J.: Flexible time management in data stream systems. In: PODS '04, pp. 263–274. ACM, New York, NY, USA (2004). DOI http://doi.acm.org/10.1145/1055558.1055596
40. Varga, A.: The omnet++ discrete event simulation system. ESM '01 (2001)
41. Wasserkrug, S., Gal, A., Etzion, O., Turchin, Y.: Efficient processing of uncertain events in rule-based systems. IEEE Trans. on Knowl. and Data Eng. **24**(1), 45–58 (2012). DOI 10.1109/TKDE.2010.204
42. Wolf, J., Bansal, N., Hildrum, K., Parekh, S., Rajan, D., Wagle, R., Wu, K.L., Fleischer, L.: Soda: an optimizing scheduler for large-scale stream-based distributed computer systems. In: Middleware '08, pp. 306–325. Springer-Verlag New York, Inc., New York, NY, USA (2008)
43. Yoneki, E., Bacon, J.: Unified semantics for event correlation over time and space in hybrid network environments. In: Proceedings of the 2005 Confederated international conference on the Move to Meaningful Internet Systems, OTM'05, pp. 366–384. Springer-Verlag, Berlin, Heidelberg (2005). DOI 10.1007/11575771_24. URL http://dx.doi.org/10.1007/11575771_24
44. Zhou, Y., Ooi, B.C., Tan, K.L., Wu, J.: Efficient dynamic operator placement in a locally distributed continuous query system. In: OTM Conferences (1), pp. 54–71 (2006)