Distributed complex event processing using rule deployment

Kang Sun[1,a], YongHeng Wang[2,b], ShuGuang Peng[3,c]

[1,2,3] College of Information Science and Engineering, Hunan University, Changsha 410082, China

[a]908481574@qq.com, [b]yh.wang.cn@gmail.com,[c]pengsg2007@126.com

Keywords: Complex event processing; Distributed processing; Event based systems

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 proposes the rule patition and deployment technique for distributed complex event processing. 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.

Introduction

At present, the continuous development of the Internet of Things(IoT) technology led to a large number of applications.The information processing is one of the key problems needed to be solved . A IOT application may contain a wide variety of connected devices,these devices constantly produce signals. For a large complex network system, these signals are massive,we call these events formed by directly obtained signals as the primitive events, which is unable to deal with or understand by the upper application or the user.A event just like "the vehicle left the garage"or"indoor temperature rise of 1 degrees Celsius"is meaningful to the user. We call the more upper events obtained by processing the primitive events as complex events. The task of identifying so called composite events from primitive ones is referred as Complex Event Processing (CEP) [1]. It operates by interpreting a set of event definition rules, which describe how composite events are defined from primitive ones.

Event-based applications possibly dispersed over a wide geographical area. Typical examples are sensor networks for environmental monitoring [5] and financial applications requiring a continuous analysis of stocks to detect trends [8]. To better support these scenarios, the CEP system can be internally built around several, distributed processors, connected together to form an overlay network, and cooperating to provide the processing and routing service.

The first important 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. In the last few years, different solutions have been proposed for operator placement [9]. The problem is known to be extremely complex to solve, even for small instances with a reduced number of processors and rules. Only a few proposals have considered a decentralized algorithm for solving the operator placement [10].

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. [2, 4]). Even when distributed processing is allowed, the communication among processors often requires manual configuration [3].

According these challenges, we have proposed the rule patition and deployment technique for distributed complex event processing. The solutions presented in this paper 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.

Background

Events and subscriptions models

In order to be understood and processed, events are observed by sources and encoded in event notifications (or primitive 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 [11]. 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)

Figure 1 summarizes the key architectural components and we consider in this paper and their interactions.

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 system: only events whose type has been advertised will be processed[6] .

The interests of sinks 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)

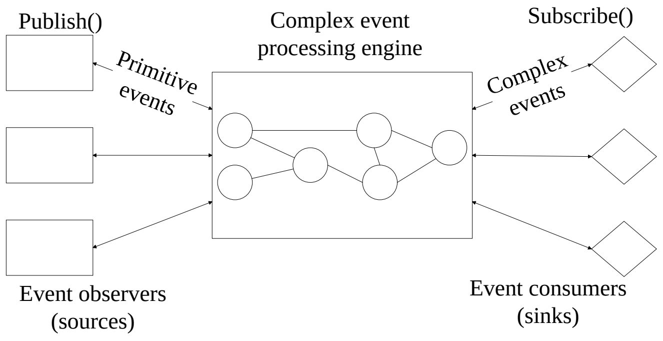

Fig.1 The high level view of a CEP application architectural

Rule definition language

In this paper, we consider rules written using the TESLA language [7]; since it includes all the typical operators used for CEP, we believe this choice will not impact the generality of our results

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
```

Shortest Path Tree

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 *processing trees* on top of the overlay network.

Since we want to minimize latency in collecting information from sources and delivering results to sinks, we build Shortest Path Tree using the link delay as a cost metric. In particular, to build the tree Tp rooted at processor p, p sends a special message CreateTreep 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 Tp, 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 ∈ P receives an ACK, it marks the sender as its child in Tp. 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 Tp, i.e., their father and the set of their children.

Rule partition and 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.

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 R1

```
Rule   R1
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 T1 shown in Fig. 2. 1 is the root of the tree and there are three sources:

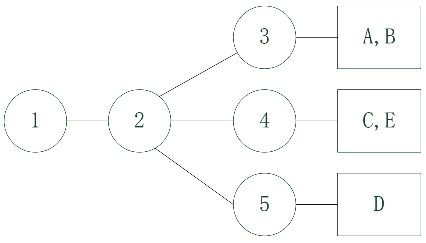

Fig.2 Rule deployment:an example

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.

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 P E that satisfy the pattern in R, it forwards all events in P E 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 R1, 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 min; 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 R1, 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 min. Finally, processor 5 receives the following partial rule, asking for all events of type D:

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 R1 and the processing tree T1 represented in Fig. 3.

At a first sight, it may be tempting to split Rule R1 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

Fig. 6, Rule R1 is split into three partial rules. The first one, involving events A and B, is forwarded to 2:

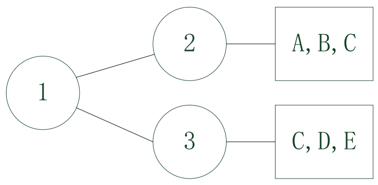

Fig.3 Handling events from multiple sources

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 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 R2 below and the two processing trees in Fig. 4.

```
Rule    R2
define      CompEvent( )
from        A( v=$x)   and   last   B(v=$x )   within   5   min   from   A
            and   last   C( )   within   5   min   from   B
```

In Fig. 4a, 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 Fig. 4b 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 R2 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 R3 and the processing trees in Fig. 3.

```
Rule   R3
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
```

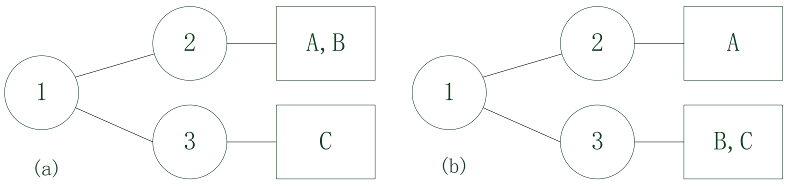

Fig.4 Handling parameters

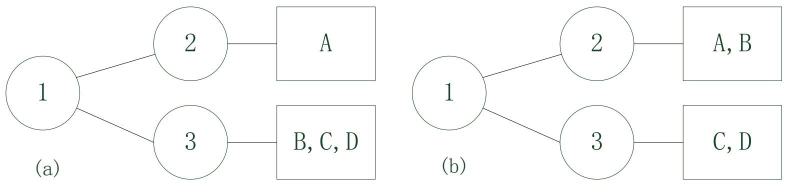

Fig.5 Handling negations

In Fig. 5a 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:

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

On the contrary, in Fig. 5b, 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()).

Evaluation

All the tests described below were performed on a 2.8 GHz AMD Phenom II PC, with 4 cores and 6 GB of RAM, running 64 bit Linux. We use a local client to generate events at a constant rate and to collect results. Omnet++ is a discrete event simulator for modelling communication networks. We create an emulated network using the Omnet++ simulator. Tree performs distributed processing on a tree architecture. Centre exploits a single processor, which receives primitive events, processes them, and delivers composite events to interested sinks.

Fig.6 shows how results change when we increase the number of deployed TESLA rules. 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. Since we are working in an emulated environment, we can measure this time without incurring in synchronization errors between processors. Figure 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 faster in Centre strategies than in Tree ones.

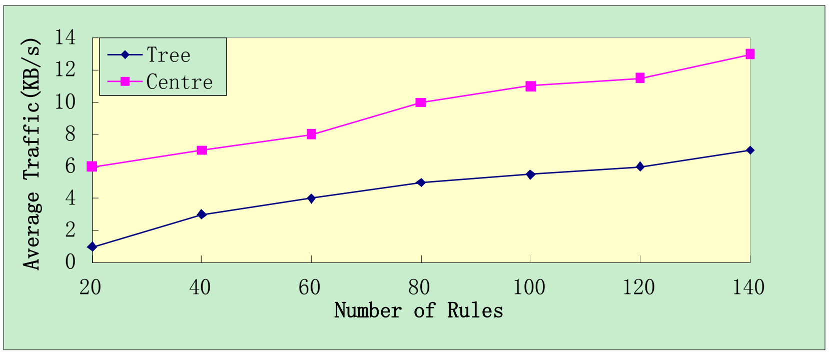

Fig.6 Number of rules deployed

Conclusion

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.

The future problem is modeling the uncertainty associated to collected information.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.

References

[1] LUCKHAM D C. The power of events: an introduction to complexe event processing in distrituted enterprise systems. Boston: Addison Wesley, 2002.

[2] Agrawal J, Diao Y, Gyllstrom D, Immerman N (2008) Efficient pattern matching over event streams. In: SIGMOD ’08, ACM, New York, pp 147–160.

[3] Ali M (2010) 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, ACM, New York, NY, USA, p 66:1.

[4] Brenna L, Demers A, Gehrke J, Hong M, Ossher J, Panda B, Riedewald M, Thatte M, White W (2007) Cayuga: a high-performance event processing engine. In: SIGMOD ’07, ACM, New York, pp 1100–1102.

[5] Broda K, Clark K, Miller R, Russo A (2009) Sage: a logical agent-based environment monitoring and control system. In: AmI ’09, pp 112–117.

[6] Carzaniga A, Rosenblum DS, Wolf AL (2000) Achieving scalability and expressiveness in an internetscale event notification service. PODC ’00, Portland, pp 219–227.

[7] Cugola G, Margara A (2010) Tesla: a formally defined event specification language. In: DEBS ’10, ACM, New York, pp 50–61.

[8] Demers AJ, Gehrke J, Hong M, Riedewald M, White WM (2006) Towards expressive publish/subscribe systems. In: EDBT ’06, pp 627–644.

[9] Lakshmanan GT, Li Y, Strom R (2008) Placement strategies for internet-scale data stream systems. In: IEEE Internet Comput 12(6):50–60.

[10] Pietzuch P, Ledlie J, Shneidman J, Roussopoulos M, Welsh M, Seltzer M (2006) Network-aware operator placement for stream-processing systems. In: ICDE ’06, IEEE Computer Society.

[11] Srivastava U, Widom J (2004) Flexible time management in data stream systems. In: PODS ’04, ACM, New York, pp 263–274.