Decision Support Systems 55 (2013) 256–269

Contents lists available at SciVerse ScienceDirect

Decision Support Systems

journal homepage: www.elsevier.com/locate/dss

Specifying and detecting spatio-temporal events in the internet of things

Beihong Jin *, Wei Zhuo, Jiafeng Hu, Haibiao Chen, Yuwei Yang

Institute of Software, Chinese Academy of Sciences, Beijing 100190, China

A R T I C L E I N F O

Article history:
Received 8 December 2011
Received in revised form 15 November 2012
Accepted 21 January 2013
Available online 13 February 2013

Keywords:
Spatio-temporal events
Pub/Sub
Event detection
Composite subscriptions

A B S T R A C T

Many applications in the Internet of Things (IoT) depend on the occurrences of events with temporal and spatial constraints to determine the further actions. A major challenge encountered is how to specify and detect the spatio-temporal events. The paper adopts Pub/Sub middleware to help IoT applications to capture spatio-temporal events. Specifically, the paper presents a composite subscription language CPSL and builds the corresponding Pub/Sub middleware Grus. The subscriptions in CPSL can specify diverse temporal, spatial and logical relationships of events, in particular, can describe the moving events related to mobile objects, and Grus is responsible for detecting whether events are matched with subscriptions in a distributed way. In addition, Grus provides the optimization strategies for subscriptions involving unary spatial operators. The paper also evaluates Grus's matching performance and costs through simulation experiments. The experimental results show that Grus can achieve satisfying performance and acceptable overheads, and the optimization strategies can efficiently speed up the detection of spatial events.

1. Introduction

With the maturity of locating technologies and the popularization of positioning equipment (e.g. GPS receivers, RFID devices), users can conveniently observe the events with the marks of time and locations which occur in surrounding environments, but only obtaining these events (called primitive events) is not enough to satisfy the requirements of applications in the Internet of Things (IoT). In many application scenarios, the spatio-temporal events need to be detected and used as decision information for behaviors in the next step, where spatio-temporal events refer to the complex events which consist of several events (such as primitive events) and satisfy the spatio-temporal constraints specified by users in advance. Moreover, observers of these spatio-temporal events are permitted to be uncoupled in space and time from objects experiencing events. Taking logistical applications as an example, logistics managers often need to observe the occurrences of the following events: whether a specific vehicle leaves a specific warehouse, whether a specific vehicle arrives in a specific area during a specific period, whether a fleet arrives at multiple unloading places in a specified order, which products need to be replenished from a store's backroom to the sales floor or from a warehouse to the specified retail store, etc. Besides, these managers who may be offline at the time of event occurrence hope to be notified once they are online. Another example comes from the location-based shopping promotion. The storekeepers in a shopping mall need to observe the events such as the occurrences of pedestrians nearby their stores. Obviously, storekeepers do not know who will occur nearby in advance, and pedestrians also do not know who will be interested in their occurrences. Here, storekeepers and pedestrians are space-uncoupled. Undoubtedly, capturing these spatio-temporal events is vital to successfully deploying such applications. Although the above applications can be designed and implemented separately, just as [14,7] have done, what we concern is how to satisfy the common requirements in those applications, i.e., effectively specifying and efficiently detecting the spatio-temporal events. Although active database systems and data stream systems can detect events, they cannot fully support the asynchronicity between event senders and receivers. However, Pub/Sub middleware [10] is appropriate to providing such services due to its inherent functionality of event detection and notification in a time- and space-uncoupled way. Therefore it can play an important role in constructing IoT applications.

In Pub/Sub middleware, events reflect the states of objects and subscriptions describe the events interested by subscribers. Further, in the Pub/Sub middleware which adopts the content-based subscription scheme, an event is depicted by multiple attribute-value pairs, and a primitive subscription is defined as a conjunction of predicates which are in the form of "name operator value" and designates the constraints on event attributes. However, a composite subscription is specified by introducing a set of operators applicable to (primitive/composite) subscriptions (also known as constitutive subscriptions), showing the constraints among events.

From the point of view of applications, users, as subscribers, connect to clients in Pub/Sub middleware and can submit subscriptions to the clients at any time, on the other hand, the primitive events are also published to

* Corresponding author.
E-mail address: beihong@iscas.ac.cn (B. Jin).

0167-9236/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.dss.2013.01.027

the clients. After the clients receive the subscriptions or events, they forward them to servers (also called brokers). The brokers are responsible for detecting whether incoming events are matched with existing subscriptions, and sending event notifications to subscribers if matched.

Since the existing Pub/Sub middleware has limited capabilities to support the spatio-temporal constraints among events, many event detection requirements in IoT cannot be met in a convenient way. To change this situation, we present a composite subscription language CPSL, and implement the corresponding middleware Grus (standard version and optimized version), where CPSL aims at expressing various event patterns involved in temporal, spatial and logical relationships among events.

The rest of this paper is organized as follows. Section 2 introduces the related work. Section 3 describes a new composite subscription language. Section 4 presents a set of approaches to detecting spatio-temporal events, including the temporal relation identification, the spatial relation identification, a subscription variable mechanism, a spatio-temporal event matching algorithm, and optimization strategies for unary location subscriptions. Section 5 evaluates the performance and costs of Grus. The last section concludes this paper.

2. Related work

Usually, event patterns which users want to observe are described by composite subscription languages in Pub/Sub middleware, whose expressiveness delimits the scopes of events which can be identified by the middleware. The earlier Pub/Sub middleware only provides limited kinds of composite subscriptions. For example, SIENA [4] only allows subscribing sequential events. However, Ref. [19] begins to change, and the CE language in Ref. [19] can support alternation (i.e. logical OR), weak sequence (i.e. concatenation), strong sequence, parallelization, iteration of the same kind of events, etc. It permits users to subscribe event combinations with a given interval. Unfortunately, it does not support non-spontaneous events which cannot notify their occurrences by themselves. Later, the requirements from workflow management and RFID applications drive researchers to offer more means to depict complex events [13,15,25]. For example, by applying newly-proposed temporal operators, RCEDA [25] can specify the sequences of event occurrences and the constraints on intervals of event occurrences. Our previous work [13] focuses on the temporal relationships of events, especially those with strict partial orders. The languages presented in Refs. [25,13] both can support non-spontaneous events, but totally ignore spatial relationships of events.

Some work pays attention to spatial events. Ref. [6] designs and implements two subscription predicates *Within* and *Distance* for an intelligent location-based service. It really has some capabilities for handling spatial events, but since it does not provide a complete subscription language, it cannot be considered as a full-fledged Pub/Sub system. Ref. [2] discusses the requirement of spatial alarm, i.e. a user specifies the spatial area that he is concerned with at first, and an alarm is raised to the user when the moving object (e.g. the user himself) enters the designated area. Since the spatial alarm can provide the detection of one kind of spatial events, it can be regarded as a variant of Pub/Sub mechanism. Refs. [2,3] give the solutions of spatial alarm, in particular, introducing safe interval and safe region to speed up the processing of spatial alarms. In addition, Ref. [9] gives a preliminary progress report on spatial event processing at IBM.

Some work related with spatio-temporal events is based on application scenarios in sensor networks. Ref. [16] provides a SQL-style language for detecting the events in a sensor network. The language can depict non-spontaneous events, but the spatio-temporal constraints provided are limited. For example, the distance between locations of two events is its only means to specify the location constraint. As to temporal constraints, only the *and*, *or* and *sequence* patterns between two events can be specified. Ref. [22] presents a composite event language SpaTec for monitoring physical phenomena and their spatio-temporal features in sensor networks. SpaTec provides six base event operators (same location, remote, sequence, concurrency, conjunction, disjunction), and four composite event operators (same location and sequence, remote and sequence, same location and concurrency, remote and concurrency). The main contribution of Ref. [22] is to give the set-based semantics of complex event expressions in theory.

In terms of complex event detection, although, as Siena and PADRES do, identifying logical relations and simple temporal relations can be added to any matching algorithm for primitive subscriptions, most of Pub/Sub middleware present new matching algorithms for their own composite subscription languages. Ref. [19] employs an automaton to handle a composite subscription after factorizing the composite subscription along its expression structure. When an automaton reaches the state with no outgoing transition, a complex event corresponding to the state is successfully detected. RCEDA [25] adopts a DAG-based structure for matching a composite subscription, wherein leaf nodes stand for primitive subscriptions and non-leaf nodes stand for the operators occurring in the composite subscription. However, Refs. [19,25] lack the mechanisms to specify and deal with the temporal relation which is bound to a specific time point.

Besides Pub/Sub middleware, ECA rules in the active database can be employed to detect the composite events which entirely exclude spatial events. For example, Snoop [5] supports temporal composite events such as sequential events and periodic events, and Ref. [23] adopts XML-based ECA rules to monitor temporal and logic composite event in e-services. Data stream systems can also detect the spatio-temporal events but in another way, i.e. by executing continuous queries. Ref. [20] proposes a query indexing mechanism for continuous static range queries on moving objects. Query indexing mechanism builds the indexes for queries instead of objects, which can decrease the number of index updates since queries are not added/removed as frequently as objects move. Further, for the first time, the notion of safe region is introduced. So-called safe regions refer to the areas which no queries care about and they are calculated so as to filter the useless object-moving events. SINA [17] implements incremental evaluation of continuous range queries on spatial–temporal data streams. It uses the shared execution paradigm to incrementally evaluate a large number of concurrent queries. The shortcoming in SINA is that it cannot permit users to add or cancel queries dynamically. Ref. [12] models continuous constraint queries as binary CSPs (Constraint Satisfaction Problems) and gets the results by running CSP solvers, but due to the limitations of implementation of existing CSP solvers, such a solution cannot deal with a large amount of continuous queries efficiently.

Compared with the existing work, our work emphasizes on the following points:

- Designs an expressive composite subscription language to specify spatio-temporal events, particularly, events reflecting objects' motion in IoT applications.
- Gives a set of full-blown implementation approaches to detecting spatio-temporal events, including temporal relation identification, spatial relation identification, subscription variable checking, etc.
- Presents optimization strategies for unary location subscriptions by borrowing the ideas from continuous query processing in data stream systems.

3. Composite subscription language specification

For a system of detecting spatio-temporal events, the way of specifying time and location of events is its foundation. We adopt an interval to denote the duration of event occurrence, using the event attribute *stm* to denote an event's starting time and *etm* to denote an event's ending time. Meanwhile, we treat the geographical location where an event occurs as a simple region (i.e. a region without holes), and employ as an event's spatial attribute a set of points which describe the region's boundary in the geographical map and the minimum convex polygon which contains all points in the set.

3.1. Operators and subscription variables in composite subscriptions

The composite subscription language CPSL adopts the content-based subscription scheme and follows the BNF in the Appendix A. In brief, a primitive subscription in CPSL is still a conjunction of predicates but predicates are in the form of "type: name operator value", and a composite subscription in CPSL is still composed of several constitutive subscriptions but the operators which connect constitutive subscriptions are newly-defined ones.

CPSL provides the following temporal operators:

- negation operator (in the basic form of $!(T;t)$, or its variants $!(A;t)$ and $!(A;B)$),
- happen-between operator (in the basic form of $=|(T;t)$, or its variants $=|(A;t)$, $=|(A;B)$),
- happen-after operator (in the basic form of $|=(T;t)$, or its variants $|=(A;t)$, $|=(A;B)$),
- concurrent operator (in the form of $|(A)$),

where T stands for an actual time point, and t denotes a temporal interval expressed by an integer, and A, B denote subscriptions, which, in the above operators, convey the time points of the occurrences of events satisfying A or B.

The choice of the above temporal operators is the result of synthesizing theory results with practical needs. The classical interval algebra [1] has recognized a total of 7 temporal relations between the temporal intervals in which the events occurred, i.e. precede, meet, overlap, cover, start, finish, and equal. From a practical point of view, we refine "precede" relation by distinguishing "happen-between" from "happen-after" relation, and merge "overlap", "cover" and the others into "concurrent" relation. Next, the negation operator is introduced for observing non-spontaneous events. Finally, by providing the basic forms of the above temporal operators, the subscriptions in CPSL are permitted to bind to specific time points.

The choice of spatial operators follows the similar working style. Spatial relationships between two regions can be classified into topological relationships, direction relationships, and metric relationships [11]. First, the topological relationships over regions can be obtained by enumerating the intersections of boundaries and interiors of two regions. The most famous theory, by far, is the spatial logic RCC-8 [21], which gives eight kinds of spatial topological relationships, that is, if there are two regions called *RegionA* and *RegionB*, then one of the following relations holds: *RegionA* disjoints *RegionB*, *RegionA* externally connects *RegionB*, *RegionA* overlaps *RegionB*, *RegionA* equals *RegionB*, *RegionB* internally contacts *RegionA*, *RegionB* in *RegionA*, *RegionA* internally contacts *RegionB*, and *RegionA* in *RegionB*. In accordance with RCC-8 and from a practical point of view, we provide three kinds of spatial operators, i.e. "overlap", "happen-in", and "same-place", where "overlap" has the same meaning as "overlap" plus "externally connect" in RCC-8, "happen-in" has the same meaning as "in" plus "internally contact" in RCC-8, and "same-place" has the same meaning as "equal" in RCC-8. Next, spatial metric relationships include the area of a region, the perimeter of a region, etc. We provide the operator for expressing the distance between the locations of two events, i.e. Euclidean distance between the centers of two areas where events take place. Now the distance operator is designed to work only on the condition that the two locations of two events are disjoint, so we can omit the disjoint operator. Finally, for the time being, we ignore the spatial direction relationships. From our observation, although the spatial direction relationships can help declare the spatial relationship between two disjoint areas, the usages of such relationships are limited in IoT applications. As a result, CPSL provides the following spatial operators:

- happen-in operator (in the basic form of $@(R)$, or its variant $IN(A)$),
- overlap operator (in the basic form of $OVLP(R)$, or its variant $OVLP(A)$),
- same-place operator (in the basic form of $SPL(R)$, or its variant $SPL(A)$),
- distance operator (in the basic form of $DIST(R;<;l)$, or $DIST(R;=;l)$, or $DIST(R;>;l)$, or their variants $DIST(A;<;l)$, or $DIST(A;=;l)$, or $DIST(A;>;l)$),

where R denotes the region which is represented as a convex polygon, l is an integer denoting the distance and A denotes a subscription, which, in the above operators, conveys the location of the occurrence of event satisfying A. Here, we refer to the subscription involving one basic form of spatial operator as unary location subscription.

In the following, we use some subscription examples to give temporal operators' semantic explanations. Assuming that A, B, and C represent subscriptions, a, b, and c denote the event instances satisfying subscription A, B and C, respectively, and ai represents the ith$(i>0)$ event instance which matches A, subscription $=|(A;t)B$ can be used to denote the events which satisfy subscription B and occur between $ai.etm$ and $ai.etm+t$; subscription $|=(A;B)C$ can be used to denote the events which satisfy subscription C and occur between some a and some b; subscription $|(A)B$ can be used to observe that some a and some b occur concurrently. However, subscription $!(A,t)B$ is satisfied only when no event b happens in the interval of $(ai.etm, ai.etm+t]$. On the other side, subscription $@(R)A$ can be used to detect the events which satisfy subscription A and occur within region R. Submitting subscription $OVLP(A)B$ means that the user wants to be alarmed when the location of some a overlaps the location of some b. Submitting subscription $SPL(A)B$ means that the user wants to observe event a and event b which happen at the same place. However, to observe the event a satisfying the condition that the distance between a's location and region R is equal to l, greater than l or less than l, subscription $DIST(R;=;l)A$, $DIST(R;>;l)A$, and $DIST(R;<;l)A$ should be employed.

CPSL also defines two logical operators && and ||, which denote logical AND and logical OR between two events respectively. Finally, same as in [13], CPSL defines operators P(t), Q(n), and S(n,t,k) so as to specify different iteration events.

In order to express the correlation among different subscriptions in one composite subscription, in particular, to express the concurrent spatial and temporal relationships among different events, we introduce the concept of subscription variable. A subscription variable is the symbol which is associated with a subscription. In a composite subscription, a subscription variable begins with "$" and ends with ";", and it can show up in the same position where a subscription may appear.

Assuming that, in composite subscription A, there are two subscription variables "$*var*1;"and "$ *var*2;" we have to conform to the following form:

$$A,\ \ var1; = (sub1),\ \ var2; = (sub2)$$

where "$*var*1;"and "$*var*2;" are bound to subscriptions *sub*1 and *sub*2 respectively, and they can occur in such a place within A where a subscription may appear.

3.2. Expressiveness

We note that the motivation behind observing spatio-temporal events comes from the fact that the objects are in constant motion. The objects may move around as time elapses, but all the changes of topological relations between two objects are reduced to a total of six modes [18]: leave, hit, reach, external, internal, and cross. Further, we can decompose such a motion mode into several related events happening within some interval and then capture them by submitting subscriptions in CPSL. However, the subscriptions submitted will be related with specific applications. The following is an example of the cross mode. The cross mode involving object A and B can be viewed as the following three sequential events happening within some interval:

Event 1::$=A$ and B occur within a certain distance (e.g. between $d1$and $d2$) at one time point;
Event 2::$=A$ and B occur at the same time and place;
Event 3::$=A$ and B occur with a certain distance (e.g. between $d1$and $d2$) at another time point;

In general, Event1 can be viewed as the starting of cross mode, therefore, the following subscription can be employed.

$$
\begin{aligned}
&= |(= |(\$e1;\ \ t)\$e2;,t2)\$e3;,\\
&\$e1; = (|(\$A;)(\$B;)\&\&DIST(\$A;;>;d1)(\$B)\&\&DIST(\$A;;<;d2)(\$B;)),\\
&\$e2; = (|(\$C;)(\$D;)\&\&\ \ SPL(\$C;)(\$D;)),\\
&\$e3; = (|(\$M;)(\$N;)\&\&DIST(\$M;;>;d1)(\$N;)\&\&\ \ DIST(\$M;;<;d2)(\$N;)),\\
&\ \$A; = E{-}A,\ \ \$B; = E{-}B,\ \ \$C; = E{-}A,\ \ \$D; = E{-}B,\ \ \$M; = E{-}A,\ \ \$N; = E{-}B
\end{aligned}
$$

where E-A and E-B denote the subscriptions used for observing that object A and B occur respectively, t denotes a specific interval, and $d1$ and $d2$ ($d1<d2$) denote two specified distances respectively.

If the exact interval (e.g. $(T, T+t]$) when the above three events occur is known in advance, then the following subscription can be used.

$$
\begin{aligned}
&= |(T,t)(\$X;),\ \ \$X; == |(\$e1;;\$e3;)\ \ \$e2;\\
&\$e1; = (|(\$A)(\$B)\&\&DIST(\$A;;>;d1)(\$B)\&\&DIST(\$A;;<;d2)(\$B)),\\
&\$e2; = (|(\$C)(\$D)\ \ \&\&SPL(\$C)(\$D)),\\
&\$e3; = (|(\$M)(\$N)\&\&DIST(\$M;;>;d1)(\$N)\&\&\ \ DIST(\$M;;<;d2)(\$N)),\\
&\ \$A; = E{-}A,\ \ \$B; = E{-}B,\ \ \$C; = E{-}A,\ \ \$D; = E{-}B, \$M; = E{-}A,\ \ \$N; = E{-}B
\end{aligned}
$$

Table 1 lists a kind of possible subscriptions corresponding to six motion modes of two moving objects (where d denotes a specific distance). The subscriptions in CPSL can also depict the six motion modes which involve one moving object and one static object. They are omitted due to the limited space. By applying proper operators to constitutive subscriptions, composite subscriptions in CPSL can express the event patterns which are related to multiple moving objects. This shows that any movement can be depicted by some subscription in CPSL, and then captured by Grus.

3.3. Examples

In this subsection, we illustrate the usage of CPSL through some specific application scenarios.

3.3.1. Logistics application scenarios

As one of the logistics management businesses, managers in logistics companies need to monitor whether vehicles carrying goods leave the source warehouse, and whether they reach the destination warehouse. Suppose that (1) the source warehouse locates in *Region*1 and the destination warehouse locates in *Region*2, (2) subscription A concerns the events of occurrences of a certain vehicle. As thus, Subscription1 in the below can be used to observe the events that vehicles locate originally in *Region*1, and after a period of $t1$ they occur at a place to which

Table 1
The subscriptions corresponding to six motion modes.

Motion modes of two moving objects	Corresponding subscriptions
Leave: object *A* leaves object *B*	=|($e1;; t)($e2;), $e1; = (\|($A;)($B;)&&DIST($A;;<;d)($B;)), $e2; = (\|($C;)($D;)&& DIST($C;;>;d)($D;)), $A; = E-A, $B; = E-B,$C; = E-A, $D; = E-B
Hit: object *A* arrives at the outside of object *B* or at the location very close to *B*	=|($e1;;t)($e2;), $e1; = (\|($A;)($B;) &&DIST($A;;>;d)($B;)), $e2; = (\|($C;)($D;) && DIST($C;;<;d)($D;)), $A; = E-A, $B; = E-B, $C; = E-A, $D; = E-B
Reach: objects *A* and *B* meet together	=|($e1;;t)($e2;), $e1; = (\|($A;)($B;)&&DIST($A;;>;d)($B;)), $e2; = (\|($C;)($D;) && SPL($C;)($D;)), $A; = E-A, $B; = E-B, $C; = E-A, $D; = E-B
External: object *A* always moves outside of object *B*	=|($e1;;t) $e2; $e1; = (\|($A;)($B;)&&DIST($A;;>;d2)($B;)), $e2; = (\|($C;)($D;) && DIST($C;;>;d2)($D;)), $A; = E-A, $B; = E-B, $C; = E-A, $D; = E-B
Internal: object *A* always moves inside of object *B*	=|($e1;;t) $e2; $e1; = (\|($A;)($B;)&&IN($B;)($A;)), $e2; = (\|($C;)($D;) &&IN($D;)($C;)), $A; = E-A, $B; = E-B, $C; = E-A, $D; = E-B
Cross: object *A* passes by object *B*	=|($e1;;$e3;) $e2; $e1; = (\|($A;)($B;)&&DIST($A;;>;d1)($B;)&&DIST($A;;<;d2)($B;)), $e2; = (\|($C;)($D;) && SPL($C;)($D;)), $e3; = (\|($M;)($N;)&&DIST($M;;>;d1)($N;)&& DIST($M;;<;d2)($N;)), $A; = E-A,$B; = E-B, $C; = E-A, $D; = E-B, $M; = E-A, $N; = E-B

the distance from *Region*1 is more than *x*1. If some events satisfying Subscription1 are detected, it means that the vehicles have left the source warehouse. On the other hand, Subscription2 is used to observe the events that vehicles locate originally in a place more than *x*2 far from *Region*2, and arrive at *Region*2 during the period of *t*2, that is, Subscription2 can examine the vehicles which reach the destination warehouse.

$$= |(@(Region1)(A); t1)(DIST(Region1; >; x1)(A)) \qquad \text{(Subscription1)}$$

$$= |(DIST(Region2; >; x2)(A)\ ;\ t2)(@(Region2)(A)) \qquad \text{(Subscription2)}$$

Regarding the warehouse as a static object, Subscription1 and Subscription2 describe the LEAVE and REACH moving mode of a moving object and a static one.

In the process of cargo transportations, the distance between the head vehicle and another vehicle *V* should not be more than *x*. If the distance is beyond *x*, then it means vehicle *V* is deviating from the fleet of the vehicles and the correlated managers are supposed to receive the notification, that is, they need to observe the LEAVE mode of two moving objects. Therefore, the correlated managers can submit subscriptions in the following format:

$$\begin{aligned}&= |(\$e1;;\ t)(\$e2;), \$e1; = (|(\$A)(\$B)\&\&\ DIST(\$A;;<;x)(\$B)),\\ &\$e2; = (|(\$C)(\$D)\&\&\ DIST(\$C;;>;x)(\$D)), \qquad \text{(Subscription3)}\\ &\$A; = E-A,\ \ \$B; = E-B,\ \ \$C; = E-A,\ \ \$D; = E-B\end{aligned}$$

where subscription *E-A* concerns the events of occurrences of other vehicles except the head vehicle, and subscription *E-B* concerns the events of occurrences of the head one. Subscription3 is used to observe the events that the distance between the head vehicle and a certain other vehicle changes from less than *x* to more than *x* in the period of *t*, which means the vehicle is deviating from the fleet of the vehicles.

3.3.2. Taxi service scenarios

The taxi service provides services for ordinary travelers. When people are in need to go out, they can submit the related subscriptions for uncarry taxi information around them. For example, people who want to know the information of uncarry taxis within 2000 meters can submit subscriptions like Subscription4.

$$DIST((x,y)); <; 2000)(STRING : type = \text{"TAXI"}; STRING : state = \text{"uncarry"}) \qquad \text{(Subscription4)}$$

In Subscription4, (*x*,*y*) represents the current location of the subscriber, *STRING*:*type* = "*TAXI*";*STRING*:*state* = "*uncarry*" means that the subscriber has interests in the taxis with uncarry state.

3.3.3. Location-based shopping promotion services

The storekeepers want to send promotions to potential customers if they occur nearby the stores. In order to find these customers, the storekeepers can submit subscriptions like Subscription5.

$$\begin{aligned}&@(PointA.x, PointA.y), (PointB.x, PointB.y), (PointC.x, PointC.y)\\ &\quad (PointD.x, PointD.y))(ARITH : ageRange = v{-}range). \qquad \text{(Subscription5)}\end{aligned}$$

In Subscription5, *PointA*, *PointB*, *PointC* and *PointD* represent the four vertexes of the rectangle region in subscription respectively, and *v-range* is a positive integer, whose value can be taken from the set of {"15", "25", "40", "65"}, representing the age group of a person: a teenager, a youth, a middle-aged person and an elderly person. What this subscription means is that the subscriber cares about those customers around his store whose ages are restricted.

4. Spatio-temporal event detection

In Grus, the basic procedure for detecting spatio-temporal events is as follows: When a composite subscription submitted by a subscriber arrives at a broker X of Grus, the corresponding matching structure is built on X and its constituent primitive subscriptions, now with X as a subscriber, are propagated to all the other brokers. When a primitive event arriving at any broker is matched with a primitive subscription with X as a subscriber, this event is sent to X for further executing the matching of complex events. If the complex event matching algorithm reports that some composite subscriptions are satisfied, then notifications are sent to corresponding subscribers. This section focuses on complex event matching.

4.1. Matching structure

Matching structures, i.e. DAGs (Directed Acyclic Graph) are used to record the information of composite subscriptions. All nodes in a matching structure can be classified into three kinds: (1) subscription nodes, recording the primitive subscriptions, (2) operator nodes, recording the operator constraints, and (3) time nodes, recording time points.

As for the simplest kind of composite subscriptions, i.e. the composite subscriptions containing only one single operator and excluding any subscription variable, the built DAGs have the following common features: there are no other nodes but zero in-degree nodes and zero out-degree nodes, where the zero in-degree nodes are subscription nodes or time nodes and the zero out-degree ones are operator nodes. Fig. 1 shows some typical matching structures built for those simplest composite subscriptions, where *t* stands for the time node. If a subscription includes temporal operators, then one or more time nodes will be built in its corresponding matching structure.

Typically, the operator node maintains the following information: (1) an index for subscription variables. If there is no subscription variable in the matching structure, this index will be set to null. To be specific, an index records (a) the definitions of the subscription variables, whether they are delivered to this node or initially defined on this node, and (b) the predecessors from which the subscription variables are delivered

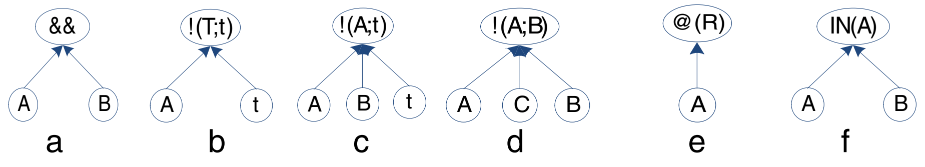


Fig. 1. Matching structures for some simplest composite subscriptions such as: (a) *A*&&*B*, (b) !(*T*;*t*)*A*, (c) !(*A*;*t*)*B*, (d) !(*A*;*B*)*C*, (e) @(*R*)*A* and (f) *IN*(*A*)*B*.

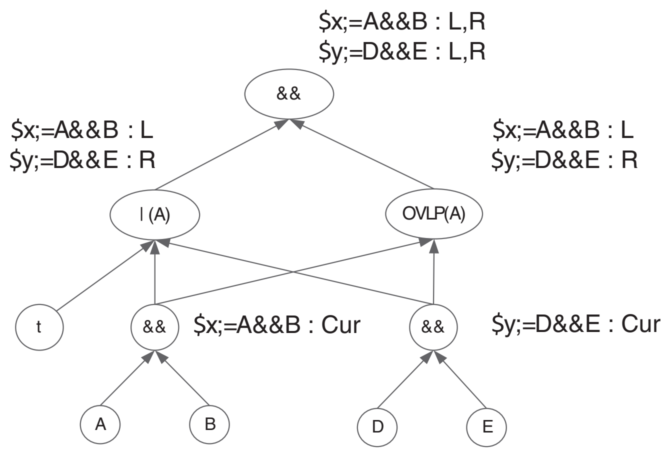


Fig. 2. Matching structure for "|($x;)$y; && *OVLP*($x;)$y;, $x; = *A*&&*B*, $y; = *D*&&*E*".

if the subscription variables are delivered; (2) one or multiple queues recording the events which have arrived; (3) pointers directing to successor nodes; (4) pointers directing to predecessor nodes; (5) a list of subscribers.

Fig. 2 shows the matching structure of subscription |($*x*;)$*y*; && *OVLP*($*x*;)$*y*;, $*x*; = *A*&&*B*, $*y*; = *D*&&*E*, and the index of subscription variables on every node. Here *L* and *R* represent that a subscription variable comes from the left node or the right node of the predecessor operator nodes.

4.2. Temporal relationship identification

In order to provide temporal relationship identification, we introduce time event and *Time Event Generator*.

Time events are events that are generated by brokers themselves and occur at the scheduled time points. For a clear description, all the events except time events are called normal events.

Usually, the time point at which a time event occurs is registered in Time Event Generator by the corresponding time node in advance, and this step is called "registering time event". When the registered time is up, Time Event Generator generates the time event and sends it to the corresponding time node.

Time events may be registered at different time points, and all the situations can fall into two categories: (1) some are registered when the operator nodes in the matching structure are created. Taking subscription = |(*T*,*t*)*A* as an example, only the events which match subscription *A* and happen in the interval of (*T*, *T* + *t*] can satisfy the constraint condition, so when the matching structure is created, two time events, *T* and *T* + *t*, will be registered; (2) some are registered when a time node's successor node receives a normal event. For example, for subscription !(*A*,*t*)*B*, when operator node !(*A*,*t*) receives an event *a*, a time event *a.etm* + *t* will be registered in Time Event Generator by the time node *t*.

Time node, as a receiver of time events, can trigger its successor nodes to execute the corresponding actions, e.g. starting or stopping dealing with normal events, and making the events in the event queue(s) valid or invalid. For example, operator node | = (*T*,*t*) will start processing incoming normal events after it receives time event *T* + *t* from predecessor time node; but operator node = |(*T*,*t*) will execute the opposite action when it receives the same time event; operator node | = (*A*,*t*) will make event *a* in the event queue valid when it receives time event *a.etm* + *t*, so that the operator node can further check whether *a* and the other incoming normal events satisfy the temporal constraint, however, for operator node = |(*A*,*t*), the same time event will cause event *a* to be deleted from its event queue.

4.3. Spatial relationship identification

In Grus, spatial relationship identification involves five types of computing:

(1) Computing the convex hull for a set of *n* points in *O*(*nlogn*) time by employing Graham algorithm [8]. When two or more events form a complex event, we calculate the convex hull which contains all the points of constituent events as the spatial attribute of the complex event.
(2) Identifying the relationship between a point and a convex polygon with *n* vertexes in *O*(*n*) time. We use cross product operation to decide whether a point is in/on the edge of/out of a convex polygon.
(3) Identifying the relationships between two convex polygons. We concentrate on three topological relationships, i.e. "in", "overlap" and "disjoint". If the two polygons have *n* and *m* vertexes respectively, we can draw the results in at most *O*(*nm*) time through calculating the relationship between one polygon and a point of another polygon one by one.
(4) Computing distance between two disjoint convex polygons. If the two polygons have *n* and *m* vertexes respectively, we can get the distance through rotating calipers algorithm [24] in *O*(*n* + *m*) time and use it as the distance between two events or between an event and an area.
(5) Computing the center of a convex polygon with *n* vertexes in *O*(*n*) time. We divide a convex polygon into *n*-2 triangles, and then calculate the center of gravity and area for every triangle. Finally we draw the polygon's area by summing up each triangle's area and the polygon's center by calculating the weighted mean value of triangle centers with triangle areas as weights. In the aid of the center of a convex polygon, we can calculate the distance between the centers of two convex polygons, and judge whether two events occur in the same place by comparing the distance with the predefined threshold.

4.4. Subscription variable checking mechanism

If an operator node in a matching structure holds the definition of a subscription variable, supposed as $*x*;, then the event *Event* that satisfies the operator constraint of the node will be delivered to its successor node in the form of "*Event*,<$*x*;, *Event*>". The successor will perform the consistency check for the assignments of subscription variables. In the following, the subscription |($*x*;)$*y*;&&*OVLP*($*x*;)$*y*;,$*x*; = *A*&&*B*,$*y*; = *D*&&*E* is taken as an example to illustrate the subscription variable checking mechanism. Here, we use a concatenation of lowercase letters to represent the instances of complex events which are composed of the events that the separate lowercase letter stands for.

Assuming that four primitive events *a*, *b*, *d* and *e*, which satisfy subscription *A*, *B*, *D* and *E* respectively, reach some broker at the same time. First, two complex events *ab* and *de* are formed in Fig. 3(a). Next, in Fig. 3(b), since the operator node representing *A*&&*B* is the node which initially defines the subscription variable $*x*;, *ab* is directly delivered to the successor nodes in the form of "*ab*,<$*x*;,*ab*>". In the same way, *de* is delivered in the form of "*de*,<$*y*;,*de*>". Then, in Fig. 3(c), since that subscription variables $*x*; and $*y*; have only appeared once in the operator node of |(*A*), there is no need to check these two variables, and it is the same for subscription variables $*x*; and $*y*; on the node of *OVLP*(*A*). Assuming that both *ab* and *de* satisfy the constraints on the nodes of |(*A*) as well as *OVLP*(*A*), the events will be delivered directly to the successor nodes in the form of "*abde*,<$*x*;,*ab*>,<$*y*;,*de*>" from both left predecessor and right predecessor. Finally, in Fig. 3(d), the above complex events reach the operator node representing |($*x*;)$*y*; && *OVLP*($*x*;)$*y*;. Since $*x*; has appeared in both left and right predecessor operator nodes, we need to check whether those events binding

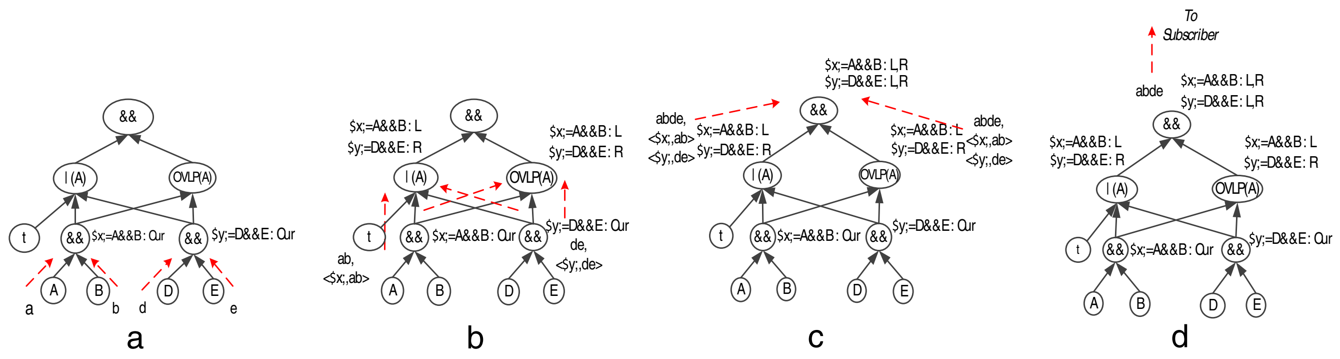


Fig. 3. Processing of subscription "|($x;)$y; && *OVLP*($x;)$y;, $x; = *A&&B*, $y; = *D&&E*".

to $x; are identical or not. For the above subscription, *ab* in "*abde*,<$*x*;,*ab*>,<$*y*;,*de*>" (from left predecessor) and *ab* in "*abde*,<$x;,*ab*>,<$*y*;,*de*>" (from right predecessor) are equivalent, therefore the constraint for subscription variable $*x*; is satisfied. The checking process for $*y*; is the same as that for $*x*;. Consequently one complex event "*abde*,<$*x*;,*ab*>,<$*y*;,*de*>" is composed and sent to the subscribers.

Let's look at another case. If the events which have occurred have the following features: (1) four events *a*1, *b*1, *d*1 and *e*1 arrive simultaneously, and at the next time point another two events *d*2 and *e*2 arrive. (2) The complex event formed by *a*1 and *b*1 cannot satisfy the spatial relationships of operator *OVLP*(*A*) with the complex event formed by *d*1 and *e*1, but it can form the overlap relationship with the complex event formed by *d*2 and *e*2. In the light of the subscription variable checking mechanism, the operator node |($*x*;)$*y*; && *OVLP*($*x*;)$*y*; will first get the complex event "*a*1*b*1*d*1*e*1,<$*x*;,*a*1*b*1>, <$*y*;,*d*1*e*1>" from its left predecessor operator node and then "*a*1*b*1*d*2*e*2,<$*x*;,*a*1*b*1>,<$*y*;,*d*2*e*2>" from its right predecessor operator node. Further, because the above two complex events have different bindings of $*y*;, the operator node representing |($*x*;)$*y*; && *OVLP*($*x*;)$*y*; will not be triggered.

4.5. Matching algorithm

When a normal event arrives at a subscription node in the matching structure, it is supposed that the event can match some primitive subscription, so it will be delivered to the subscription node's successors to trigger further matching. Each successor node will check the received event with its constraints, and once the constraints can be satisfied, a new complex event will be created. If the list of subscribers is not null on this node, the complex event will be sent to these subscribers as a notification. At the same time, the complex event will continue to be delivered to successors for further processing. This procedure will be repeated until a zero out-degree node is reached. With regard to the time event, it follows the above processing procedure except that it is sent to the corresponding time node at the beginning.

Consumption semantics [5] determines how to form complex events from the incoming events and when to delete those events. Ref. [5] proposes four kinds of semantics, i.e. Recent, Chronicle, Continuous and Cumulative semantics. We present a new one, i.e. First-Matching semantics. Following First-Matching semantics, the oldest instance of each component event is selected to construct a complex event and the event will be discarded once it is matched. In Grus, we support Recent, Chronicle, and First-Matching semantics.

The existence of subscription variables may cause an event to be delivered to multiple operator nodes, and an event may become invalid due to different reasons (e.g. the event just arrived at a temporal operator node may become invalid due to the arrival of a time event), therefore, we design the event invalidation mechanism. Once an event becomes invalid, this mechanism is triggered to delete the copies of the invalid event which have been delivered to the other operator nodes, so as to guarantee that events are consistently recorded in the matching structure.

Fig. 4 gives the matching algorithm. Concretely, once an event arrives at a subscription node, the matching method *CompositeSubMatch*(*event*, *null*) will be invoked automatically.

In lines 3–5, a primitive event on a subscription node is driven into the matching structure. In lines 7–8, *registerTimer*() is used to judge whether a time event needs registering according to the incoming event. If it returns true, a time event is registered. In lines 9–11, corresponding events are extracted from event queues according to the operator constraint and consumption semantics at first. Then the logical, temporal, spatial and subscription variable constraints are detected. If all constraints are satisfied, one complex event will be composed. In lines 12–13, if there is any subscriber on the current node, it means that the current node stands for some subscription, so the subscribers will be notified. In lines 18–19, if *event* is a time event, *timeEventAction*() will be called to trigger a time node to start corresponding processing. Since a time event may make some normal events invalid, the event invalidation mechanism may be executed in this method. In lines 21–22, during the matching process, if the arrival of an event makes some other existing events on a node unable to form new complex events, we refer to the newly-arrived event as a tail event. Those unavailable events and the tail event will be discarded; and meanwhile the event invalidation mechanism needs to be executed. In lines 23–26, if *event* is not a time event or a tail event, but it may form a complex event with some other events some time later, then it will be put into the event queue, otherwise event invalidation process is run.

For a composite subscription, assuming that in its matching structure, there are *M* operator nodes and the longest length of the paths from zero in-degree nodes to zero out-degree nodes is *H*, an incoming event will trigger (1) at most *H* constraint checking on the operator nodes in the worst case, and (2) at most *M* event invalidation processing. Moreover, we note that checking spatial constraints takes more time, and by contrast, the time taken to check the other constraints (such as temporal constraints, logical constraints, etc.) is negligible, therefore, if n is the maximum number of vertexes of polygons met in checking the spatial constraints and $O(f(n))$ denotes the time complexity of detecting a spatial constraint on a node, then the algorithm will finish a matching procedure in $O(H \times f(n) + M)$ time.

Grus (standard version) implements the above matching algorithm.

4.6. Optimization strategies

We present the optimization strategies for unary location subscriptions: build multi-level indexes at brokers so as to improve the processing performance of brokers, and on the other hand, utilize the computational capacity of client computers to calculate and

Algorithm 1: matching algorithm

```
//Input: event, gNode, where event is a newly-arrived event, gNode is the node from which the
event is delivered,
//gNode is set to null while the algorithm is called at first time.
//Output: notify the subscriber who has interested in event.
1 VOID CompositeSubMatch (Event event, GraphNode gNode)
2 BEGIN
3   IF gNode = null
4     FOR each successor node pNode
5       pNode.CompositeSubMatch(event, this)
6   ELSE
7     IF this.registerTimer(gNode) = true
8       TimeEventGenerator.register(event)
9     Do logical/temporal/spatial constraints and subscription variable checking
10    IF all the constraints are satisfied
11      Form the matched events into a new CompositeEvent cEvent
12      IF this.SubscriberSet.size() != 0
13        notifyAllSubscribers(cEvent)
14      ELSE
15        FOR each successor node pNode
16          pNode.CompositeSubMatch(cEvent, gNode)
17    ELSE
18      IF event instanceof TimeEvent
19        Do this.timeEventAction(gNode, event)
20      ELSE IF event is a tail event
21        Delete useless events related to event
22        Do Event Invalidation Process for these deleted events
23      ELSE IF event can trigger the node with some events later
24        this.addToEventList(gNode, event)
25      ELSE
26        Do Event Invalidation Process for event
27 END
```

Fig. 4. Matching algorithm.

maintain safe regions on the clients, so that the events can be filtered on the clients and the workloads on the brokers can be reduced.

4.6.1. *Safe region*

We define the safe region as a region that is on the outside of all regions which are concerned by subscriptions. If the location of an event is in this region, then it will not trigger any subscription. In other words, the spatial events in safe region can be filtered out directly, and do not need to be sent to the broker for further matching. The ideal safe region is the rest region of the entire map from which all the regions concerned by subscriptions are removed. It is obvious that the load to compute the ideal safe region is enormous while

Algorithm 2: GLR algorithm

```
//Input: p_s C_{k,l} , where p_s is a location point, and C_{k,l} is the grid corresponding to the position point
//Output: safe region $\varphi_s$
1 safeRegion GLRSafeRegionAlgorithm(Position $p_s$, Grid $C_{k,l}$)
2 BEGIN
3   FOR each sub in $A_s$
4       $R_{s,k,l} = R_{s,k,l} \cup \{Loc(sub) \cap C_{k,l}\}$
5   IF $R_{s,k,l}$.isEmpty() = true
6       $\varphi s = C_{k,l}$
7   ELSE
8       regionQuads = assignRegionToQuads($R_{s,k,l}$)
9       canPointsQuads = getQuandrants(regionQuads)
10      canPointsQuads = generateCandidatePoints(canPointsQuads)
11      canPointsQuads = generateTensionPoints(canPointsQuads)
12      $\varphi_s$ = getFinalSafeRegion(canPointsQuads)
13 RETURN $\varphi_s$
```

Fig. 5. GLR algorithm.

facing a huge map or lots of subscription regions. It's difficult to calculate the ideal safe region, so we need to find a way to get a reasonable and calculable safe region. Therefore, we design and implement an algorithm to express and calculate safe region, we name it Greedy Largest Rectangle (GLR) algorithm, which is shown in Fig. 5.

First of all, we regard the entire map U as a large rectangle $Rect(x,y,w,h)$, where (x,y) represents the bottom left point of the map, w and h represent the width and height of the map respectively. Secondly, we divide the rectangle into $M \times N$ grids, and the grid size is not less than the size of the safe region of any mobile object in any time. Thus, taking the bottom left vertex of the rectangle as the origin of coordinates, cell C_{ij} can be defined as $C_{ij} = Rect(x + i \times \alpha, y + j \times \beta, \alpha, \beta)$, $M = \lceil w/\alpha \rceil$, $N = \lceil h/\beta \rceil$. After that, we can define $f(p) = C_{\frac{p_x - x}{\alpha} \frac{p_y - y}{\beta}}$, where $f(p)$ is the mapping from point $p(p_x, p_y)$ to its grid. Moreover, let p_s be the current location, and ϕ_s be the largest safe region with the center of p_s. ϕ_s can be calculated by the formula $\phi_s = C_{kl} - \bigcup_{i=1}^{|A_{s,k,l}|} R(A_i)$, where C_{kl} represents the gird in which p_s locates, $R(A_i)$ represents the region which is the scope of subscribers' concerns and locates in the grid of p_s, and $A_{s,k,l}$ represents the set of subscriptions related to the grid of p_s, that is, the regions which $A_{s,k,l}$ concerns locate in the grid of p_s.

Let A_s be the set of unary location subscriptions related to the grid of p_s in the entire map. GLR algorithm uses as input data the current location p_s and the grid C_{kl} which can be calculated by the mapping function $f(p)$ and works as follows.

At first, we need to get $R_{s,k,l}$, the set of the regions of unary location subscriptions locating in the grid C_{kl}. That is, for each subscription in A_s, get the region which it concerns through the function $Loc(sub)$. If the region is not a rectangle, expand it to its minimum bounding rectangle. In detail, if the region is in the grid C_{kl}, then put it into $R_{s,k,l}$, otherwise put the intersection between the region and C_{kl} into $R_{s,k,l}$. If set $R_{s,k,l}$ is empty, then return the entire grid C_{kl} as the safe region. After that, we call function *assignRegionToQuads*( ), it can map the subscription in $R_{s,k,l}$ to each quadrant while taking p_s as the center of four quadrants and return an array *regionQuads*, of which the subscript represents the number of quadrants and the element represents the set of regions (rectangles now) in the corresponding quadrant. Then, we calculate a rectangular safe region: (1) for each

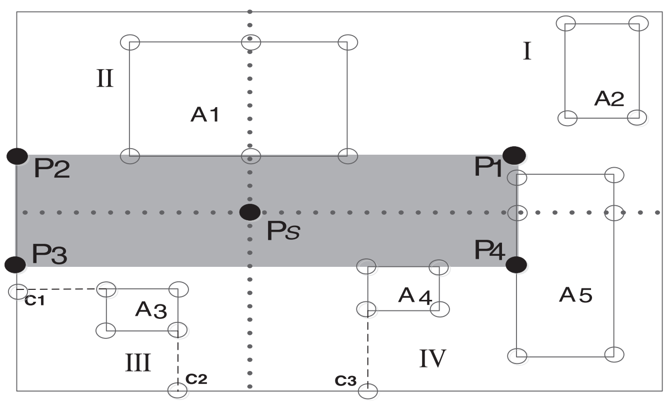


Fig. 6. An example of safe region.

vertex of rectangles in every quadrant, get rid of those points far away from p_s in $O(|R_{s,k,l}|)$ time, and return *canPointsQuads* which is the initial set of candidate points; (2) sort these points in *canPointsQuads* according to the quadrants and get rid of those points which can be dominated by other points in the same quadrant, which takes O(|*canPointsQuads*| × log|*canPointsQuads*|) time; (3) do further processing to these points in *canPointsQuads*, and return the corner points of the rectangle with p_s as one of its vertexes in each quadrant, which also takes O(|*canPointsQuads*| × log|*canPointsQuads*|) time; (4) construct a maximum rectangular safe region in O(|*canPointsQuads*|) time by employing a greedy strategy on the points in *canPointsQuads*.

Therefore, the computational complexity of GLR algorithm is at most O(*nlogn*), where *n* is the number of unary location subscriptions in a grid.

When a client receives a location event from a mobile object, it will determine whether there is a history record of the mobile object's position. If so, it means that the safe region for the mobile object has been established. Otherwise, the client will send this location event to the broker, ask the broker to send the subscriptions related to the grid which is calculated according to the current location reported by the mobile object's location event, and then call GLR algorithm to build a safe region for the mobile object. If the safe region has already existed and the mobile object's current location is in it, then this event is ignored. On the contrary, the client will send this location event to the broker and call the GLR algorithm to calculate the safe region for the mobile object again. By calculating and maintaining safe regions on the clients, some events can be filtered on the clients, and the workloads on the brokers can be reduced, but at the expense of it, subscription information needs to be stored in the clients. Fig. 6 is an example of a safe region. In Fig. 6, P_s is the mobile object's location stored in a certain client, rectangles A1–A5 represent the regions concerned by subscriptions, *P*1, *P*2, *C*1, *C*2, *C*3, and *P*4 are corner points in each quadrant and the rectangle in the shadow is the final safe region of the mobile object.

4.6.2. Spatial indexes

In order to enhance the event detection performance, we explore the relationships between regions which occur in unary location subscriptions, and build spatial indexes for these regions in the format of R-Trees. R-Tree is the extension of B + tree in multi-dimensional space, and it is also an AVL tree. R-Tree consists of two kinds of nodes, that is, non-leaf nodes and leaf nodes. Leaf nodes store bounding rectangles of actual spatial objects. Non-leaf nodes store the minimum bounding rectangles of all objects within their children nodes. By building spatial indexes with the help of R-Tree, we can find the rectangle region which contains a known location quickly.

We use multilevel index mechanism to manage the spatial information in unary location subscriptions. The first level is the index of location subscription operators, such as @, *DIST*, etc., the second level is the index of lower level subscriptions (they are primitive subscriptions in general), and the last level is R-Tree level. In particular, the information recorded in R-Tree contains not only the information of spatial regions, but also the information of subscribers who submit subscriptions. Therefore, when a location event arrives, if a spatial region of which the subscribers are not null can be found in R-Tree index, it indicates that there might be some subscriptions related to the location event. Furthermore, if the restraint relationship between the location event and the regions in subscriptions is satisfied, event notifications can be sent directly according to the subscriber information. It's seen that in some scenarios, for example, in the shopping promotion scenario, there are a lot of subscriptions in the same format but with different values of regions, so it greatly accelerates matching spatial events with

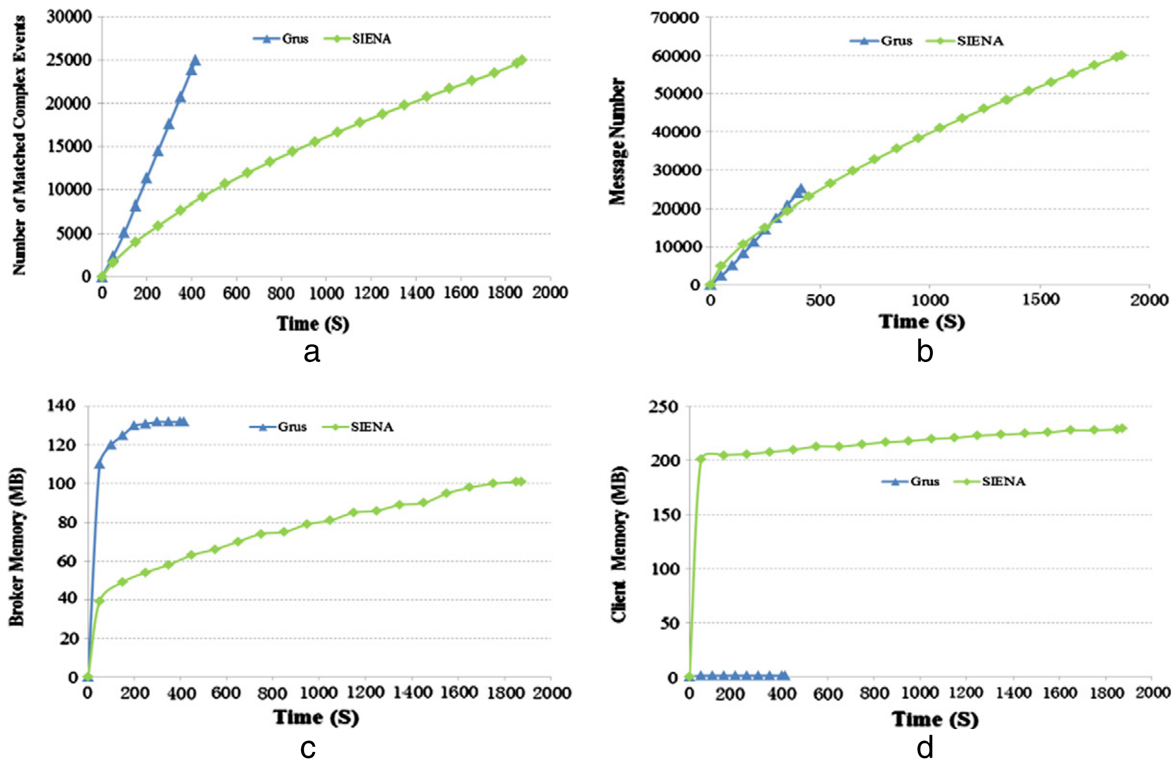


Fig. 7. For the logistics application scenario: (a) number of matched complex events; (b) number of network messages; (c) broker memory consumption; and (d) client memory consumption.

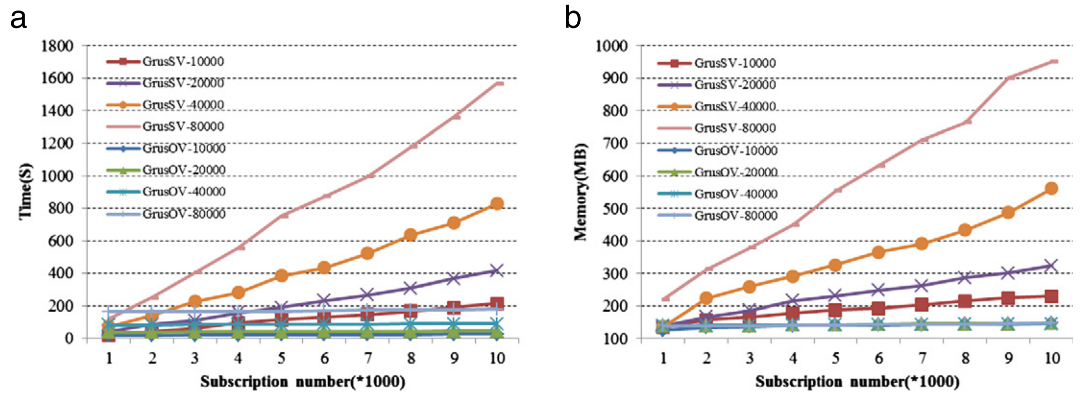


Fig. 8. For the taxi service scenario: (a) time consumption and (b) memory consumption.

unary location subscriptions using the multilevel index structure described in the above.

The optimization strategies described above are implemented in the optimized version of Grus. The optimized version can support wireless clients by deploying the client software on users' Android-based mobile devices.

5. Evaluation

In order to evaluate the performance and costs of Grus (standard and optimized versions), we carry out a series of experiments. Firstly, we design the experiment which simulates the logistics application scenario, so as to compare the standard version and SIENA in terms of performance and costs of matching spatio-temporal events. For the sake of the fairness of the evaluation, we develop a fat client for SIENA to make it offer the same composite subscription matching functionality as Grus does. This experiment focuses mainly on testing the matching performance and the corresponding costs of the system, therefore, we deploy the standard version of Grus and the modified SIENA on one broker and four clients. Secondly, we evaluate the optimization strategies in optimized version for unitary location subscriptions. We simulate the scenarios of taxi service and shopping promotion service in the experiments, so that the optimization strategies in optimized version can be invoked.

5.1. Logistics application scenario

We take Subscription1–3 in Section 3.3 as three kinds of inputs of the experiment. In particular, subscription *A*, *E-A*, *E-B* in the above Subscription1–3 are all in the following format: *ARITH*: *id* = *v*1,*STRING*: *type* = *v*2, where *id* represents the identifier of the vehicle, *type* represents the type of the vehicle, such as trucks, vans, etc., *v*1 is a positive integer, and *v*2 is a string constant. We generate 5000 specific composite subscriptions for each kind of subscription. Furthermore, events are generated in the format as follows: *ARITH*: *id* = *v*1,*STRING*: *type* = *v*2, *timestamp*, *x*, *y*, where the meaning and range of *v*1 and *v*2 are the same as the subscriptions mentioned above, *timestamp* represents the time of event occurrence, and (*x*, *y*) represents the location of event.

In the experiment, 15,000 composite subscriptions are first injected into the system under test, and then 60,000 events are continuously published at a ratio of 1500 event/s. The event order in the event stream can guarantee that two or four consecutive events can satisfy an existing composite subscription.

We record the number of matched complex events, the number of network messages, and the memory consumption in the broker and the clients. The experimental results are shown in Fig. 7. The memory consumption of four clients is nearly the same, so only one client's memory consumption is provided. From the results shown in Fig. 7, we can see that the standard version of Grus takes 416 s to match 25,000 complex events, but the modified SIENA takes 1873 s to finish

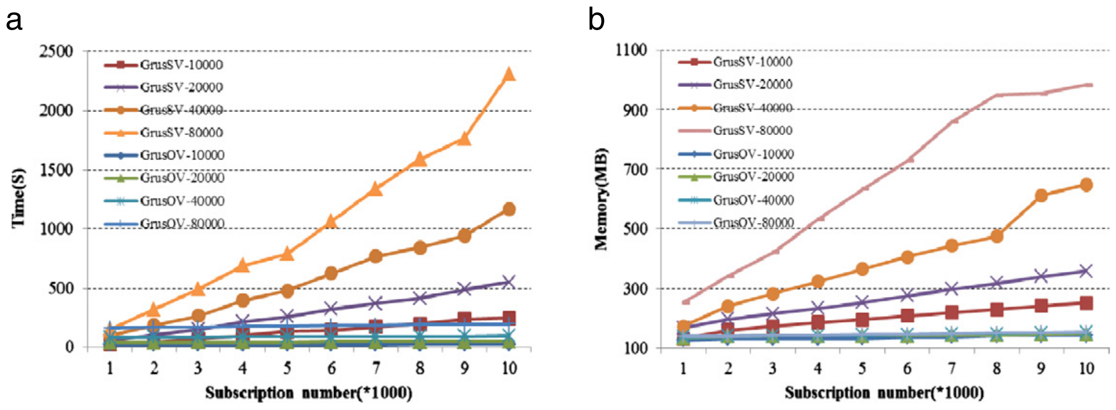


Fig. 9. For the shopping promotion scenario: (a) time consumption and (b) memory consumption.

the same task, which means that the throughput of Grus is 4.5 times of that of the modified SIENA. The number of network messages used in Grus is 25,000 but 60,000 in the modified SIENA, therefore, as far as the number of network messages consumed by each matched event is concerned, the modified SIENA is 2.4 times as many as Grus. Besides, during the process of the experiment, broker memory consumption of Grus is at most 1.3 times as many as the modified SIENA's, but client memory consumption of Grus is far less than the SIENA's.

From the above experiment, it concludes that the throughput of Grus is much higher than the modified SIENA's. Since fat client strategy completes the computation of the spatio-temporal detection in clients, it not only brings about the fact that the throughput is lower than Grus, but also results in the increase of the number of events transferred in the system, which increases the network messages. Meanwhile, if the same subscription is submitted by different clients, the computation of matching the same subscription has to be done in multiple clients, which increases the consumption of computing resources in the system.

5.2. Taxi service scenario

This experiment simulates the taxi service scenario where different users submit multiple Subscription4s in Section 3.3 and uncarry moving taxis send events in the following format:

$$\begin{aligned}&ARITH: id = v{-}id,\ \ STRING: type = \text{"}TAXI\text{"},\ \ STRING: state\\ &\quad = \text{"}uncarry\text{"},\ \ timestamp, x, y,\end{aligned}$$

where *v-id* is a positive integer which is used for identifying the taxi, *timestamp* represents the time of event occurrence, and (x,y) represents the current location of the taxi.

The procedure of the experiment is as follows. At first, a certain number of Subscription4s are injected into the standard version and the optimized version of Grus, and then events are sent which follow a Poisson arrival process at a rate of 500 per second. In all the published events, 50% of them can be matched with subscriptions. In particular, events are generated to simulate the driving routes of taxis, so they have continuity. The continuity of location events makes it possible for clients to filter those events which are impossible to be matched through the safe region mechanism.

Fig. 8 shows the time and space consumption of the standard version and the optimized version of Grus under different numbers of published events. In Fig. 8, *X* axis represents the number of subscriptions, which increases from 1000 to 10,000 by 1000 and *Y* axis represents the time or memory consumed by the broker.

As shown in Fig. 8, while facing the same events, the optimized version, compared with the standard version, has relatively strong advantages in matching the subscriptions with $DIST(R;<;L)$ operator. For example, for matching 20,000 events, the average time of the 10 experiments of the optimized version is only 1.02% of that of the standard one's, and the average memory consumption of the optimized version is 9.32% of the standard one's.

The experimental data in Fig. 8 also show how the processing time and the occupied memory of a broker change with the increase of the total number of events. For example, when the number of subscriptions is fixed at 10,000 and the total number of events multiplies from 10,000 to 80,000, the time consumed by the broker for the optimized version is 26 s, 47 s, 91 s and 179 s, and the memories occupied are 147M, 147M, 146M, and 148M, respectively. However, the corresponding time for the standard version is 215 s, 417 s, 829 s, and 1567 s, and the memories are 230M, 323M, 561M and 951M, respectively. Considering that the time used for sending the above events is 20 s, 40 s, 80 s, and 160 s, respectively, the above results indicate that the optimized version can finish dealing with the events within a short period of time after all the events are sent, however, the standard version has to take a longer time. The experimental data also illustrate that the optimized version can offer scalability to support a large number of events with relatively stable memory costs. In essence, these are mainly due to the filtering of events by clients so as to offload work from the broker.

5.3. Shopping promotion scenario

This experiment simulates the shopping promotion scenario where different storekeepers submit multiple Subscription5s in Section 3.3 and moving users publish events about their own ages and locations in the format as follows:

$$\begin{aligned}&STRING: username = v{-}name, ARITH\\ &\quad : ageRange = v{-}range, timestamp, x, y,\end{aligned}$$

where *v-name* represents the user's name, whose value can only be taken from the set {“*a*”, “*b*”, “*c*”, “*d*”, “*e*”}, representing five different users, the meaning of *v-range* is the same as that of *v-range* in Subscription5, *timestamp* represents the time of event occurrence, and (x,y) represents the user's current location.

The procedure of the experiment is as follows. At first, a certain number of Subscription5s are injected into the system, then five users walking in different routes are simulated, each one with the same number of consecutive location events. All the events follow a Poisson arrival process at a rate of 500 per second, and 50% of them can be matched with subscriptions which have been injected into the system. Fig. 9 shows the time and memory consumption of the standard version and the optimized version under different numbers of published events. In Fig. 9, X axis and Y axis have the same meanings as the corresponding ones in Fig. 8.

While the data in Fig. 9 are compared with the ones in Fig. 8, the similar data trends can be found in the case of the same number of events. This situation indicates that optimized version can match a great quantity of events against the subscriptions with @(R) operator efficiently at the expense of relatively stable memory, which also illustrates the optimized version has the intrinsic scalability.

6. Conclusions

In this paper, we focus on detecting spatio-temporal events through Pub/Sub middleware. We provide an expressive subscription language for complex spatio-temporal events. The new language equips ten temporal operators of four kinds and eleven spatial operators of four kinds, whose composition can describe a majority of spatio-temporal events in IoT applications, especially in the scenarios where the spatio-temporal events relate with two or more moving objects. On the other hand, we propose a set of detecting methods which incorporate temporal relationship identification, spatial relationship identification, subscription variable checking, etc. Moreover, we present the speed-up strategies for unary location subscriptions. All mentioned above have been implemented in our Pub/Sub middleware Grus (standard version and optimized version). Experimental results show that Grus (standard version) is efficient and low-cost, as compared with the modified SIENA, and Grus (optimized version) can effectively accelerate the processing of spatial event detection as compared with Grus (standard version). Our next work is to apply Grus in practical applications and improve system robustness.

Acknowledgment

Supported by the National Natural Science Foundation of China under Grant No. 60970027 and No. 91124001.

Appendix A. BNF for composite subscription language CPSL

```
BINARY::=('0'|'1')+('b'|'B')
INT::= ('-')? DECIMAL_LITERAL (['l','L'])? | HEX_LITERAL (['l','L'])?
DECIMAL_LITERAL::=['1'-'9'] (['0'-'9'])*
HEX_LITERAL::='0' ['x','X'] (['0'-'9','a'-'f','A'-'F'])+
Digit::= ['0'-'9']
Numeric::= ('-')? (Digit–'0') Digit* ('.' Digit+)?
Letter::= ['a'-'z', 'A'-'Z']
Name ::= Letter (Letter | Digit | '.' | '-' | '_' )*
SubVariable ::= '$' Name ';'
PredVariable ::= '&' Name ';'
Subscription ::= CompositeSub (',' Timestamp)? '@'
Timestamp= Numeric
compositeSub ::= Sub (',' SubVariable '=' '(' Sub ')')*
Sub ::= LogicalOrSub | SubVariable
LogicalOrSub ::= LogicalAndSub ('||' LogicalAndSub)*
LogicalAndSub ::= OtherCSub ('&&' OtherCSub)*
OtherCSub ::= TemporalSub | RepetitiveSub | SpatialSub | PrimitiveSub | '(' Sub ')'
TemporalSub ::= NotSub| SeqBeforeSub | SeqConcurrentSub | SeqAfterSub
NotSub :: = '!(' TimeRange ')(' Sub ')'
SeqBeforeSub ::= '=|(' TimeRange ')(' Sub ')'
SeqConcurrentSub ::= '|(' Sub ')(' Sub ')'
SeqAfterSub ::= '|=(' TimeRange ')(' Sub ')'
TimeRange ::= INT ';' INT | Sub ';' INT | Sub ';' Sub
RepetitiveSub::= SntkSub | PtSub | QnSub
SntkSub ::= 'S(' n ',' t ',' k ')(' Sub ')'
PtSub ::= 'P(' t ')(' Sub ')'
QnSub ::= 'Q(' n ')(' Sub ')'
SpatialSub ::= SpatialHappenInSub | SpatialTopologicalSub | SpatialDistanceSub
SpatialHappenInSub ::= '@(' Region ')(' Sub ')'
Region ::= '((' X ',' Y')'(',' PointOther)* ')'
PointOther ::= '(' X ',' Y')'
X ::= Numeric
Y ::= Numeric
SpatialTopologicalSub ::= SpatialInSub | SpatialOverlapSub | SpatialSamePlaceSub
SpatialInSub ::= 'IN(' Sub ';' Sub ')'
SpatialOverlapSub ::= 'OVLP(' Place ';' Sub ')'
SpatialSamePlaceSub ::= 'SPL(' Sub ';' Sub ')'
SpatialDistanceSub ::= 'DIST(' Place ';' DistOp ';' INT ')(' Sub ')'
Place ::= Sub | Region
DistOp ::= '>' | '=' | '<'
PrimitiveSub ::= Predicate (',' Predicate)*
Predicate ::= RFIDType | ArithType | StringType
RFIDType ::= 'RFID:' (RFIDExpress | RFIDExpressVa)
RFIDExpressVa ::= ('PRODUCT' | 'READER') RFIDOp PredVariable
RFIDExpress ::= ('PRODUCT' | 'Reader') RFIDOp RFIDValue (('&&' | '||') RFIDExpress)*
RFIDOp ::= '<(' m ',' n ',' id ')' | '>(' m ',' n ',' id ')' | '<=(' m ',' n ',' id ')' | '>=(' m ',' n ',' id ')' | '=(' m ',' n ',' id ')' | '!= (' m ',' n ',' id ')'
RFIDValue ::= BINARY | HEX_LITERAL
ArithType ::= 'ARITH:' (ArithExpressVa | ArithExpress | RangeExpressConj | RangeExpressDisj)
ArithExpressVa ::= Name ArithOp PredVariable ('+' | '-' | '*' | '/' Numeric)?
ArithExpress ::= Name ArithOp Numeric (('&&' | '||') ArithExpress)*
ArithOp ::= '<' | '>' | '=' | '<=' | '>=' | '!='
RangeExpressConj ::= Name '<-' ('[' | '(') Numeric ',' Numeric (')' | ']')(('&&' | '||') RangeExpressConj)*
RangeExpressDisj ::= Name '<+' ('[' | '(') Numeric ',' Numeric (')' | ']')(('&&' | '||') RangeExpressDisj)*
StringType ::= 'STRING:' (StringExpressVa | StringExpress | EnumerationExpress)
StringExpressVa ::= Name StringOp PredVariable
StringExpress ::= Name StringOp String (('&&' | '||') StringExpress)*
StringOp ::= '*>' | '>*' | '*=' | '=' | '!=' | '*<' | '<*' | '=*'
String ::= '"' (#x9 | #xA | #xD | [#x20-#x21] | [#x23-#xFF] | #x22#x22)+ '"'
EnumerationExpress ::= Name ('<-' | '<+') '{' String (',' String)+ '}'
```

References

[1] J.F. Allen, Maintaining knowledge about temporal intervals, Communications of the ACM 26 (11) (1983) 832–843.

[2] B. Bamba, L. Liu, P.S. Yu, G. Zhang, M. Doo, Scalable processing of spatial alarms, Annual IEEE International Conference on High Performance Computing, 2008.

[3] B. Bamba, L. Liu, A. Iyengar, P.S. Yu, Distributed processing of spatial alarms: a safe region-based approach, International Conference on Distributed Computing Systems, 2009.

[4] A. Carzaniga, D. Rosenblum, A. Wolfal, Design and evaluation of a wide-area event notification service, ACM Transaction on Computer Systems 19 (3) (2001) 332–383.

[5] S. Chakravarthy, D. Mishra, Snoop: an expressive event specification language for active databases, Data and Knowledge Engineering 14 (1) (1994) 1–26.

[6] X. Chen, Y. Chen, F. Rao, An Efficient Spatial Publish/Subscribe System for Intelligent Location-Based Services, DEBS, 2003.

[7] C. Condea, F. Thiesse, E. Fleisch, RFID-enabled shelf replenishment with backroom monitoring in retail stores, Decision Support Systems 52 (4) (2012) 839–849.

[8] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, Second edition MIT Press, 2001.

[9] O. Etzion, N. Zolotorevsky, Spatial Perspectives in Event Processing. From Active Data Management to Event-Based Systems and More. LNCS 6462, Springer-Verlag, Heidelberg, 2010, pp. 85–107.

[10] P.Th. Eugster, P.A. Felber, R. Guerraoui, A.M. Kermarrec, The many faces of publish/subscribe, ACM Computing Surveys 35 (2) (2003) 114–131.

[11] R.H. Guting, An introduction to spatial database systems, The VLDB Journal 3 (4) (1994) 357–399.

[12] M. Hadjieleftheriou, N. Mamoulis, Y. Tao, Continuous Constraint Query Evaluation for Spatiotemporal Streams, SSTD, LNCS 4605, 2007, pp. 348–365.

[13] B. Jin, X. Zhao, Z. Long, F. Qi, S. Yu, Effective and efficient event dissemination for RFID applications, The Computer Journal 52 (8) (2009) 988–1005.

[14] K. Li, T.C. Du, Building a targeted mobile advertising system for location-based services, Decision Support Systems 54 (1) (2012) 1–8.

[15] G. Li, H. Jacobsen, Composite subscriptions in content-based publish/subscribe systems, The 6th ACM/IFIP/USENIX International Middleware Conference, 2005, pp. 249–269.

[16] J. Mao, J. Jannotti, M. Akdere, U. Cetintemel, Event-based constraints for sensornet programming, the Second International Conference on Distributed Event-Based Systems, 2008.

[17] M.F. Mokbel, X. Xiong, W.G. Aref, SINA: scalable incremental processing of continuous queries in spatio-temporal databases, ACM SIGMOD conference, 2004, pp. 623–634.

[18] P. Muller, Topological spatio-temporal reasoning and representation, Computational Intelligence 18 (3) (2002) 420–450.

[19] P.R. Pietzuch, B. Shand, J. Bacon, A framework for event composition in distributed systems, the 4th ACM/IFIP/USENIX International Conference on Middleware, 2003, pp. 62–82.

[20] S. Prabhakar, Y. Xia, D. kalashnikov, W.G. Aref, S. Hambrusch, Query indexing and velocity constrained indexing: scalable techniques for continuous queries on moving objects, IEEE Transactions on Computers 51 (10) (2002) 1124–1140.

[21] D.A. Randell, Z. Cui, A.G. Cohn, A spatial logic based on regions and connection, The 3rd International Conference on Knowledge Representation and Reasoning, 1992, pp. 165–176.

[22] S. Schwiderski-Grosche, K. Moody, The SpaTeC composite event language for spatio-temporal reasoning in mobile system, the Third International Conference on Distributed Event-Based Systems, 2009.

[23] S. Swamynathan, A. Kannan, T.V. Geetha, Composite event monitoring in XML repositories using generic rule framework for providing reactive e-services, Decision Support Systems 42 (1) (2006) 79–88.

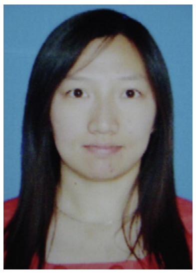
[24] G. Toussaint, Solving geometric problems with the rotating calipers, IEEE Melecon (1983).
[25] F. Wang, S. Liu, P. Liu, Y. Bai, Bridging Physical and Virtual Worlds: Complex Event Processing for RFID Data Streams, EDBT, 2006, pp. 588–607.

Beihong Jin received B.S. degree in 1989 from Tsinghua University, China, M.S. degree in 1992 and Ph.D. degree in 1999 from Institute of Software, Chinese Academy of Sciences, all in computer science. Currently she is a professor at Institute of Software, Chinese Academy of Sciences. Her research interests include mobile and pervasive computing, middleware and distributed systems. She has obtained research support from the National Natural Science Foundation of China, the 863 plan of China MOST (Ministry of Science and Technology), etc. She has published over 70 research papers in international journals and conference proceedings and holds one China patent. She is a senior member of the CCF (China Computer Federation) and a member of the ACM and the IEEE.

Wei Zhuo received B.S. degree in 2009 from Chinese University of Science and Technology, and M.S. degree in 2012 from Institute of Software, Chinese Academy of Sciences, all in Computer Science. Currently he is a software engineer focusing on NLP and Data Mining in Ren Min Search Company.

Jiafeng Hu is an M.S. candidate at Institute of Software, Chinese Academy of Sciences in Computer Science. He received B.E. degree in 2011 from Jilin University, China. His research interests include mobile and pervasive computing, middleware and distributed systems.

Haibiao Chen received B.S. degree in 2008 from Xi'an Jiaotong University, and M.S. degree in 2011 from Institute of Software, Chinese Academy of Sciences, all in Computer Science. Currently he is a software engineer focusing on mobile PaaS in Yahoo! Beijing Global R&D center.



Yuwei Yang is an M.S. candidate at Institute of Software, Chinese Academy of Sciences in Computer Science. She received B.E. degree in 2011 from Jilin University, China. Her research interests include mobile and pervasive computing, middleware and distributed systems.