Electronic Communications of the EASST
Volume 37 (2011)

Workshops der wissenschaftlichen Konferenz Kommunikation in Verteilten Systemen 2011 (WowKiVS 2011)

Distributed Composite Event Detection in Publish/Subscribe Networks – A Case for Self-Organization

Enrico Seib, Helge Parzyjegla, and Gero Mühl

12 pages

Guest Editors: Horst Hellbrück, Norbert Luttenberger, Volker Turau
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Distributed Composite Event Detection in Publish/Subscribe Networks – A Case for Self-Organization

Enrico Seib[1]*, **Helge Parzyjegla**[1], **and Gero Mühl**[1]

[1] {enrico.seib,helge.parzyjegla,gero.muehl}@uni-rostock.de
University of Rostock, Institute of Computer Science

Abstract: Event-based cooperation is well suited to model the interaction of components in distributed, dynamically changing environments prevalent, for example, in ubiquitous computing scenarios. Publish/subscribe middleware can be used to efficiently implement event-based cooperation. However, while application components may be interested not only in single events, but also in spatio-temporal patterns of events, called composite events, research in the area of routing algorithms for publish/subscribe systems has focused mainly on efficiently routing individual notifications from producers to their consumers without providing means for correlation. In order to avoid every application having to subscribe to all events that may form an interesting event pattern to detect, which can waste large amounts of network bandwidth, we propose to realize composite event detection as a middleware service. While a centralized implementation of this service would be simpler to realize, we favor distributed composite event detection inside the broker network because this way locality in publication rates and subscriber interests can be exploited. However, placing detectors for composite events such that the required network bandwidth is minimized is a complex on-line optimization problem. In this paper, we present our ideas to place composite event detectors inside the publish/subscribe broker network and to adapt this placement at runtime. The placement is based on a self-organizing optimization using a spring relaxation heuristic considering multiple event patterns.

Keywords: Composite Event Detection, Publish/Subscribe, Self-Optimization

1 Introduction

Event-based cooperation has been identified to be well suited for various application domains such as ubiquitous computing, sensor networks, electronic commerce, and information-driven network applications. It is characterized by asynchronous and indirect exchange of *events* which are happenings of interest corresponding to state changes of application components or the surrounding environment. In contrast to request/reply-based approaches, this style of cooperation results in self-focused application components that are loosely coupled facilitating flexibility, changeability, and extendability. On this layer of abstraction, it is left open how events are detected, how the events that are of interest to an application component are determined, and how these events are routed to the interested components.

* Funded by Deutsche Forschungsgemeinschaft (DFG SPP 1183 Organic Computing)

Publish/subscribe middleware can be used to efficiently implement event-based cooperation as described above: *producers* publish *event notifications*, which describe occurred events and their context, and *consumers* issue subscriptions to receive the event notifications they are interested in. Interposed between producers and consumers is the *notification service* that is responsible to deliver published notifications to all components with a matching subscription.

In a distributed setup, the notification service is usually realized by a network of interconnected *brokers* forwarding event notifications, just called events in the following, from the publishers to the subscribers stepwise through the broker network. To achieve this, each broker manages a set of *local clients* and a *routing table* that is updated to reflect the subscriptions that are currently active in the system. How the routing tables are updated when subscriptions are issued and revoked in the system is determined by the applied routing algorithm. Since in many scenarios, like sensor networks, network bandwidth and computing power is scarce, the main goal of routing algorithms is usually to use as low network bandwidth and computing power as possible.

Research on routing algorithms in publish/subscribe systems has concentrated on routing individual events efficiently from their publishers to their subscribers. However, applications might not only be interested in single events, known as *primitive events*, but also in *composite events* which are *spatio-temporal patterns* of two or more events being either primitive or composite events themselves. Composite events can, in general, be specified by constraints that evaluate and correlate the content as well as the temporal and spatial characteristics of a set of candidate events. To detect a composite event, the sets of events that fulfill the constraints of a respective event pattern have to be identified. If a notification service only allows to subscribe to single events based on their characteristics, the only option is that the application detects the composite events it is interested in itself by subscribing to all candidate events that may lead to a certain composite event. However, this approach may be very inefficient since the set of all single candidate events might be huge compared to the number of composite events detected.

Therefore, we propose to offer composite event detection as a middleware service. To achieve this, the composite events to be detected have to be specified explicitly by the applications, e.g., using a suitable *event algebra*. For implementing a middleware service for composite event detection different possibilities exist. In the basic case, detection is done by a centralized middleware component that subscribes to all events that might lead to a composite event instead of the applications. While this approach is simple to realize and slightly more efficient, it is still far from optimal, because it is not able to exploit the locality in event patterns and publication rates. This potential can only be lifted by distributed composite event detection. Moreover, a distributed approach also enables further optimization opportunities such as splitting detectors into subdetectors that can be placed at different brokers. In this case, composite event detection is delegated to the brokers inside the publish/subscribe network, where *composite event detectors* are placed. Our goal is to avoid unnecessary network traffic as well as wasting memory capacity by placing detectors as near as possible to the event sources and evaluating conditions as early as possible to filter out inappropriate events. However, placing detectors adaptively in a dynamic system is a complex on-line optimization problem that is only feasible using heuristics. For these kind of problems, self-organizing algorithms are a promising solution [MWJ+07, JPMH07].

In this paper, we sketch a self-organizing algorithm that adaptively places composite event detectors and carries out further optimizations to minimize network bandwidth consumption. The algorithm is based on an analogy to spring relaxation and heuristically applies four basic

operations for composite event detectors (detector decomposition, replication, migration, and recombination) which are chosen based on local knowledge.

The remainder of this paper is structured as follows: In Sect. 2, we discuss the specification of composite events using constraints and elaborate on event algebras. Section 3 depicts the problem and the challenges of distributed composite event detection and presents our self-organizing algorithm for adaptive composite event detector placement. In Sect. 4 we review existing approaches for distributed event detection from different areas and compare them to our approach. Section 5 presents the conclusions of our paper and outlines future work.

2 Composite Events Specification

Events are happenings of interest which are reified by event notifications materializing the respective event and the context of its occurrence. Composite events are event patterns that can be specified by content-based, temporal, and spatial constraints on events. Conceptually, we define an *event history* that contains all events ever detected. If a new event is detected it is added to the event history and for each event pattern all subsets of the history (i.e., sets of events) are determined that contain the new event and satisfy the constraints of this event pattern. These subsets are the newly detected composite events. If constraints refer to time, a subset of the event history may also newly fulfill the constraints of an event pattern if solely time has advanced. Thus, event detection has not only to consider new events but also time. We now explain in more detail how constraints can be used to specify composite events. Then, we discuss the role of event algebras.

2.1 Constraints on Event Patterns

Constraints can refer to the content of events as well as to its spatial and temporal characteristics. However technically, temporal, and spatial information is often just a part of the event's content. Since it is common to model event notifications as data structures consisting of a set of name/value pairs called *attributes*, constraints refer to the attributes of the candidate events. Thus, the constraints must be satisfied by the attributes' values of the candidate events to trigger a composite event. We distinguish constraints by the number of events that are involved in the evaluation of a constraint. Constraints that only evaluate a single event can easily be pushed to a place in the broker network, where filtering will be most effective. For example, consider the composition of two events E_1 and E_2. The specification $\{(E_1,E_2) \mid E_1.humid > 50\% \ \wedge \ E_2.temp > 30°\text{C}\}$ consists of two constraints that each evaluate only a single event. Both can simply be pushed near to the sources of the events to filter out uninteresting events early. Constraints which correlate attributes of multiple events, however, are more complex to handle because they cannot simply be broken down into constraints on individual events. An example of a constraint evaluating two events E_1 and E_2 is: $\{(E_1,E_2) \mid |E_1.temp - E_2.temp| > 10°\text{C}\}$. To evaluate such constraints, all events E_1 and E_2 have to be correlated by an event detector. An example for a more complex specification is $\{(E_1,E_2) \mid [E_1.room \neq E_2.room] \ \wedge \ [\nexists E_3 : E_3.room = E_1.room \ \wedge \ E_3.time > E_1.time] \ \wedge \ [\nexists E_4 : E_4.room = E_2.room \ \wedge \ E_4.time > E_2.time] \ \wedge \ [|E_1.temp - E_2.temp| > 10°\text{C}]\}$. Here, only the newest instances of E_1 and E_2 are eligible for composition.

Once the event sets matching a pattern have been determined, several options remain. One is

to hand over all these sets to the applications. Another option is to derive from each subset a data structure that contains only the information the application is really interested in and to deliver this data structure to the application. This could involve selection of attributes, but also further processing of attributes (e.g., subtraction) to derive new attributes as specified by the application. Finally, the application could also instruct the middleware to publish the derived data structure as a new event of some type such that other applications can receive this new event by subscribing to the respective event type instead of the event pattern.

2.2 Event Algebras

While constraints on sets of events that are evaluated against the event history form the theoretical basis of event composition, constraints with many variables, logical operators (such as $\vee$, $\wedge$, and $\neg$) as well as existential and universal quantifiers ($\exists, \forall$) can get very complex to write down. Because of that *event algebras* have been introduced in the context of active databases [CKAK94]. The main ingredients of event algebras are *event composition operators* and *event consumption modes*. While the former define how events are composed, the latter define which events are eligible for composition. Common composition operators are:

- *Disjunction* ($E_1 \vee E_2$): Composite event fires if E_1 or E_2 occurs.
- *Conjunction* ($E_1 \wedge E_2$): Composite event fires if E_1 and E_2 occurs.
- *Sequence* ($E_1; E_2$): Composite event fires if E_1 occurs followed by the occurrence of E_2.
- *Any* (any$(m, E_1, E_2, \ldots, E_n)$): If at least m of the given n events occurred, the composite event fires.
- $E_1^*(t_i)$: The composite event fires only once, even if event E_1 occurs more than once in a certain time period t_i.
- *Times* ($\times(n, E_1, t_i)$): The composite event fires ones if E_1 occurred at least n-times in a certain time interval t_i.
- *Not* ($\neg(E_1, t_i)$): The composite event fires if event E_1 did not occur during time interval t_i.

The presented operators are commonly used in centralized active databases, but are also well suited for distributed event detection. More sophisticated operators as well as a formal definition of operators on event histories can be found in [CKAK94].

Using event composition operators, composite events can be specified much more easily. However, considering a specification such as $(E_1 \wedge E_2)$, it becomes evident that if a new instance of event E_1 is detected, all pairs of the E_1 and all E_2's that are already in the event history are detected as new composite events. This is also called *unbounded context* because the events eligible for event composition are not restricted and all events remain available for future compositions. With this approach, the number of detected composite events may be huge and, in many cases, not all events will be meaningful to the application. Because of this *consumption modes* have been proposed as a comfortable mean to restrict the events eligible for composition. Common consumption modes are *Recent*, *Chronicle*, *Continuous*, and *Cumulative* [CKAK94]. They

differ in what initiator and terminator events can be paired, in which order events are combined, and which events cannot participate in future compositions anymore. A possible alternative to consumption modes are *windows* which may be defined in terms of time or number of events. For example, a window may contain the events that occurred in the last 10 minutes or the last 100 event occurrences. While a consumption mode is usually defined for the whole pattern, windows are usually applied to each composition operator.

3 Distributed Composite Event Detection

Realizing composite event detection as a middleware service requires applications to explicitly specify the event patterns they are interested in. This enables the middleware to identify sub-patterns, instantiate appropriate event detectors, and optimize, evaluate, and adapt its detection strategies. Particularly, by decomposing and distributing detectors in the network, the middleware is able to exploit different publishing rates and spatial distributions of events to significantly reduce the amount of network traffic. In the following, we first identify requirements, problems, and challenges to efficiently detect composite events in dynamic environments. Afterwards, we address these issues and present a self-organizing approach for distributed event detection.

3.1 Requirements and Challenges

The problem of placing composite event detectors in a network can be seen as assigning tasks to brokers (cf. task assignment problem), which is known to be NP-complete [Bok81] even for static environments. Moreover, since static optimization strategies are in most cases inappropriate for dynamically changing environments, we face a complex dynamic on-line optimization problem. Due to the problem complexity our objective is to develop a self-organizing algorithm with minimal overhead that enables the middleware to adapt the composite event detection to a dynamically changing environment by applying heuristic local rules. To reach this target, the following requirements and challenges have to be tackled:

- **Flexible and precise event algebra.** Application developers need a convenient and expressive way to describe the event patterns they are interested in, while middleware developers require precise specifications of composite events in order to implement appropriate detectors. Thus, an event algebra with a sound set of operators is needed that allows for formal and accurate event definitions, while still providing enough composition flexibility.

- **Adaptivity.** In dynamic environments, network conditions and client behavior change over time affecting the magnitude and occurrence of event streams and, thereby, the efficiency of placed event detectors. Thus, any placement decision made must be continually reviewed and adapted. In fact, this requires flexible detector implementations that can be seamlessly migrated and relocated along event streams within the network.

- **Distributed and self-organizing algorithm.** Due to system scalability, a centralized optimization approach based on global knowledge is not feasible. Instead, brokers have to take optimization decisions based on local knowledge. Employing such a distributed optimization algorithm leads towards a self-organizing placement of composite event detectors.

- **Detector decomposition and recombination.** In order to save network bandwidth and processing capacity, it is beneficial to early determine and filter out those primitive events that will not trigger a composite event detection. To achieve this, detectors for subpatterns need to be placed as close to event sources as possible. Thus, an effective decomposition strategy is required allowing to split complex event detectors into detectors for constituent subpatterns. Moreover, the inverse operation must be applied if the recombination of several subdetectors to a complex but more efficient detector becomes possible.
- **Multiple event patterns and event streams.** When placing event detectors, it is not sufficient to just consider those belonging to a single composite event pattern. Since composite events may share common subpatterns, the placement of new detectors needs to be carefully coordinated with existing detectors and integrated into established event streams.

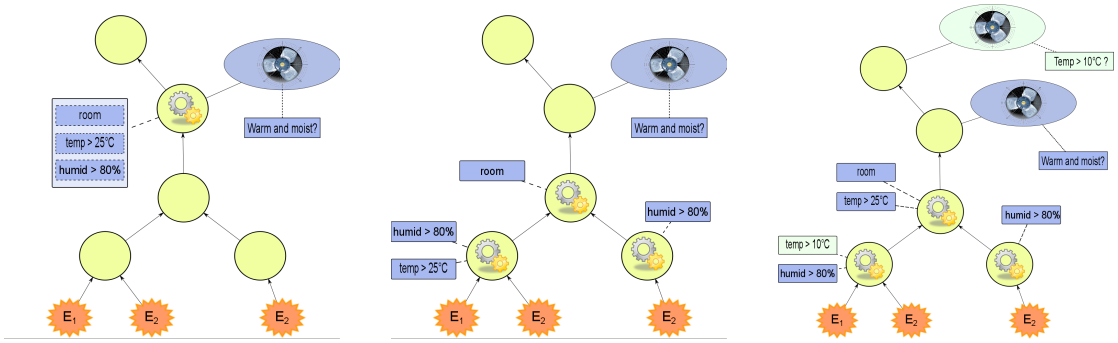
3.2 Detector Placement

The previous section highlighted requirements and challenges for realizing an efficient and effective distributed detection of composite events. Particularly, dynamic environments and the lack of an overall system view make it difficult to identify those brokers that are able to efficiently detect requested event patterns. Thus, instead of trying to directly determine a beneficial detector placement, it is rather sensible to concentrate on continually improving an existing placement. To achieve this, we define four basic optimization steps—*decomposition*, *replication*, *migration*, and *recombination* of event detectors—that brokers can trigger based on local knowledge. By a sequence of these steps, many different global detector placements can be reached. In order to support these optimization steps, event detectors are implemented in such a way that they can be seamlessly migrated and replicated.

In the following, we discuss the four basic optimization steps in greater detail. To illustrate each step, we use an elementary example taken from the facility management domain. An application controls and monitors the air conditioning units of an office building with green recreation areas. Certain plants are quite sensitive and pose strict climatic requirements. Thus, the building is equipped with temperature and humidity sensors that produce measurement events on a regular interval. These events are collected and routed by a broker network. The application analyzes the events and triggers an alert whenever an anomaly is detected. This is the case, when a room gets too warm (temperature is higher than 25°C) and too moist (humidity rises above 80%). Formally, this case is described by the composite event pattern $\{(E_1, E_2) \mid [E_1.temp > 25°C] \wedge [E_2.humid > 80\%] \wedge [E_1.room = E_2.room]\}$.

3.2.1 Detector Decomposition

Complex event detectors can be decomposed hierarchically into simpler ones that are responsible for constituent subpatterns. Basically, the rules how to split up detectors have to be derived from the event algebra. Usually they depend on the operators used to define the composite event. In case of the *And* ($\wedge$) operator, the complex pattern can be split up, treated as product of commutable functions [GJS92]. Decomposing the *Or* ($\vee$) operator is even simpler as the left hand and right hand side can be treated as individual subpatterns. Also the *Sequence* (;) operator can be split up into a left and a right hand side subpattern, which are evaluated individually.



(a) Decomposition (b) Replication and Migration (c) Recombination

Figure 1: Example application illustrating optimization steps of detector distribution

Considering our example in Fig. 1(a), the application issues a subscription for composite events specified by $\{(E_1,E_2) \mid [E_1.temp > 25°C] \wedge [E_2.humid > 80\%] \wedge [E_1.room = E_2.room]\}$. The hosting broker first creates a complex detector for this expression and identifies its constituents parts as well as their dependencies. In this example, subdetectors for the expressions $\{(E_1) \mid [E_1.temp > 25°C]\})$ and $\{(E_2) \mid [E_2.humid > 80\%]\}$ are created. Furthermore, their results are the input for a third subdetector representing $\{(E_1,E_2) \mid [E_1.room = E_2.room]\}$.

3.2.2 Detector Replication

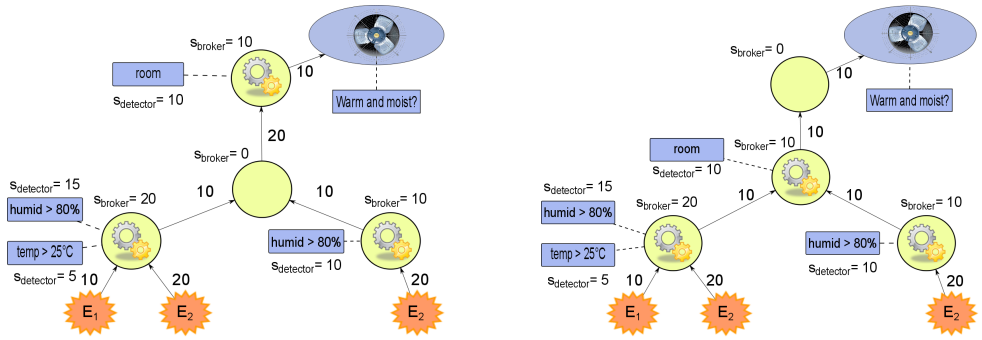
Detectors representing an unary event operator are usually subject to replication. This is especially beneficial if the replicas are placed close to event sources so that many primitive events get filtered out early. However, detectors directly depending on multiple event streams can be replicated, too. In this case, the event space is split into disjunct domains, where each detector replica is responsible for one of these domains exclusively. Detector replication is usually considered together with migration as replicas are hosted on different brokers.

In our example, as shown in Fig. 1(b), events with humidity measurements originate from different sources. Thus, it is sensible to replicate the humidity detector $\{(E_2) \mid [E_2.humid > 80\%]\}$ and instantiate copies on the two lower brokers closer to the sources. Afterwards, the humidity detector on the central broker is not necessary anymore.

3.2.3 Detector Migration

Event detectors can be migrated seamlessly from one broker to a neighboring broker. This is useful to shift detectors or subdetectors along an event stream closer towards the source in order to filter out primitive events or partial event patterns earlier. If events originate from multiple sources, detector migration has to be combined with replication.

Considering the example application, it is beneficial to migrate the detector for the subpattern $\{(E_1) \mid [E_1.temp > 25°C]\}$ from the central broker to the lower left broker as shown in Fig. 1(b). Likewise, the two replicas of the $\{(E_2) \mid [E_2.humid > 80\%]\}$ detector have been shifted to the lower two brokers. Regarding the subdetector $\{(E_1,E_2) \mid [E_1.room = E_2.room]\}$, however, it



(a) Before Relaxation (b) After Relaxation

Figure 2: Example illustrating optimization approach based on spring relaxation

is not yet clear whether a migration is advantageous. As the detector receives events from two different sources, this is only the case if the events from one source outweigh the other.

3.2.4 Detector Recombination

Recombination dissolves detectors that are no longer beneficial anymore and recombines their state with related or dependent detectors on neighboring brokers. Therefore, recombination includes revoking of event subscriptions as well as shifting of cached events. Recombination may be triggered by various reasons, e.g., by network reconfigurations, by load balancing mechanisms, or even by new client subscriptions that influence and change established event streams.

Recombination is also necessary in the running example. Assume another application (lime green ellipse in Fig. 1(c)) that is hosted on the upper left broker subscribes to all temperature events higher than 10°C. Using migration, the corresponding detector $\{(E_1) \mid [E_1.temp > 10°C]\}$ eventually arrives at the lower left broker close to the source of temperature events. Thereby, the existing subpattern detector $\{(E_1) \mid [E_1.temp > 25°C]\}$ loses its selectivity as a superset of temperature events has to be routed through the network towards the upper broker. Hence, the subdetector can be dissolved on the lower broker and recombined with the dependent subdetector $\{(E_1, E_2) \mid [E_1.room = E_2.room]\}$ from the first application on the broker in the middle.

3.3 Self-Organizing Optimization

By using the placement operations presented in Sect. 3.2 and carrying them out subsequently in appropriate order, many different global detector placements can be achieved. Moreover, these basic operations are well suited to gradually adapt a placement when event streams change. The problem, however, is to decide which operation or sequence of operations to perform next in order to improve the placement. Brokers take these decisions based on a cost model, the selectivity of a detector, and a spring relaxation heuristic.

Costs. Our cost model considers three kinds of costs: forwarding costs, storage costs, and placement costs. Forwarding costs arise for each event that is processed by a broker and for-

warded to a neighboring broker or client. Our primary goal is to reduce these costs as they directly reflect network consumption and broker utilization. Storage costs are induced by event detectors that need to cache candidate events relevant for the respective pattern they recognize. Forwarding and storage costs incur independent from an adaptive detector placement strategy. Placement costs, however, are additional costs that reflect the overhead necessary for performing one of the basic placement operations in order to improve the detector configuration. In case of detector migration, e.g., the number of cached events that need to be transported are considered.

Selectivity. The selectivity of a detector measures how well the detector is suited to reduce the number of events needed to be forwarded. Therefore, brokers monitor the input and output streams and count the events per stream and direction that each hosted detector consumes. The more events a detector consumes exclusively, i.e., no other detector, broker, or client is interested in the particular event, the higher is the detector's selectivity. If the event needs to be forwarded to another broker or client, however, it is not counted. Thus, selectivity especially identifies those detectors that have a major impact when subject to placement operations while respecting existing event streams that are caused by other detectors or subscriptions.

Spring Relaxation. Detector placement decisions are made by brokers based on a spring relaxation heuristic similar to those of Pietzuch [PSB04] and O'Keeffe [O'K10]. Detectors and subdetectors are modeled as a system of interconnected springs. The tensions of the springs are proportional to the costs that are caused by the respective event streams and detectors per time unit. The algorithm determines the sum of the forces that act on a particular detector and moves it along the direction of the resulting force in order to balance the whole system.

The primary force causing a placement operation is derived from a detector's selectivity as its value, while respecting other detectors and subscribers, determines the forwarding costs that can be saved by migrating the detector towards the events' source. If a spring clearly dominates a detector in the model, the detector becomes subject to migration. If two or more springs pull apart the detector in different directions in a way that the forces cancel out each other, however, we check whether the detector can be decomposed or replicated. If this is possible, new subdetectors or replicas are inserted into the spring model while continuing the relaxation. However, additional storage costs need to be considered now that arise from subdetectors or replicas needing their own memory to store events for combination. In the spring model, this is reflected by a force that pulls related detectors together with a strength that is proportional to the storage costs that could be saved by a recombined detector. If such a spring dominates a set of related detectors a recombination is performed.

To counter oscillations, we use friction as a third force in the spring model. Its value is proportional to the placement costs that measure the overhead necessary to perform the respective placement operation. Moreover, by introducing a tunable friction coefficient we can thus control the system's responsiveness by which it adapts the detector placement to changed event streams.

As an example for a migration, we use a broker network similar to our running example in Sect. 3.2, where an application is interested in all warm and moist rooms. The number of events passing each broker is also included in the figures. The subdetectors for temperature and humidity are already placed as near as possible to the event sources as shown in Fig. 2(a). By

acting as filters on single events, they do not require additional memory capacity, the resulting force derived from additional storage costs is equal to zero. The uppermost broker contains the *Room* detector. By the resulting selectivity of this detector, the room detector is pulled to its predecessor broker in the middle. Because the detector was not replicated or split up before, there is no force derived from additional costs for storage. If we assume that the force derived from selectivity is greater than the friction, the room detector is migrated to its predecessor broker as depicted in Fig. 2(b).

4 Related Work

In this section, we review existing approaches for distributed event detection from different areas such as publish/subscribe middleware, stream processing, and sensor networks.

Publish/Subscribe Middleware. O'Keeffe [O'K10] presents an overview of the state of the art in composite event detector placement strategies in publish/subscribe networks including an outline on static and dynamic detector placement algorithms. He identifies re-using of (partial) event patterns as crucial for efficiency. The approach presented considers different layers in a rendezvous-based middleware, like Pietzuch et al. [PSB04] introduced for stream processing. Each broker locally computes its coordinates inside a virtual (cost and/or latency) space, whereas the intermediate layer uses information provided by the broker layer and the upper entire query layer to compute the optimal positions of detectors using the cost space. Thereafter, detectors are mapped to the underlying network. The approach uses a mixture of centralized and decentralized approaches offering the opportunity for central computation of optimal detector placement as well as decentralized refinements. Carzaniga et al. [CRW01] present an approach based on matching of advertisements and subscriptions for (composite) event patterns from neighbor brokers by maintaining tables of available patterns. Each broker maintains a table with patterns, which is the poset of advertisements arriving at the broker and patterns already processed on this broker, containing the neighbor brokers from which each pattern is available. The approach of Carzaniga et al. introduces splitting up and distributing detectors. But only a very restricted set of patterns is supported. Furthermore, it is not mentioned clearly which conditions lead to certain placement decisions.

Stream Processing. Bonfils and Bonnet [BB03] present an approach for detector placement based on neighbor exploration. The approach uses additional cost messages, which flow from children to their parent nodes. Local cost information of each node is forwarded to a specific candidate set for each parent node consisting of certain neighboring nodes which compute their own cost situation and compare it to the cost situation of the active node. If this leads to an improvement, the query is bypassed to this candidate. The approach is based on local decisions executed on each node in relation to a bounded candidate set. The additional cost messages basically conflict the idea of saving bandwidth consumption. Moreover, the choice of suitable candidates and the computation of alternative paths causes additional computational overhead.

Sensor Networks. Kamiya et al. [KMI+08] enable users to subscribe for composite events in heterogeneous sensor networks consisting of separate subnets interconnected by gateway nodes. An issued subscription is sent to the closest gateway node which parses and decomposes the subscription and forwards the constituent parts to responsible gateways. Each gateway manages all event detection trees within its subnet and tries to integrate new partial subscriptions by reusing as much of the established tree structures as possible. The authors describe the initial decomposition and delegation of composite event detection in detail, but omit to show how the system reacts and adapts to dynamically changing network structures. Ying et al. [YLTX08] present a distributed algorithm to adaptively place composite event detectors. The authors state that placing detectors as near as possible to event sources is intuitive but not always the best choice as nodes close to event sources are likely to have limited processing and storage capabilities. The approach is based on a distributed Bellman-Ford algorithm where nodes maintain additional state information summarizing routing, communication, and storage costs. These costs are determined and updated according to state information provided by neighboring nodes. The algorithm sensibly adapts the placement of detectors and event caches, but neglects the optimization potential of decomposing and recombining event detectors.

5 Conclusions

In this paper, we proposed to integrate composite event detection into publish/subscribe middleware. We argued that due to efficiency and scalability reasons, composition should not be centralized, but distributed to the brokers of the publish/subscribe network. This enables the middleware to exploit locality in event publication rates and event pattern interests. However, distributed composite event detection in dynamically changing environments is a complex task. First of all, the complexity of the resulting on-line optimization problem requires to apply heuristics based on local knowledge. Furthermore, optimization strategies must look at more than one pattern at once because patterns can share common subpatterns. Thus, optimizing event patterns separately will likely underperform strategies taking dependencies among the patterns into account. We have proposed a self-organizing optimization that uses migration and replication of event detectors as well as detector decomposition and recombination to improve the efficiency of composite event detection even for multiple queries. While detector migration is always possible, detector replication, decomposition, and recombination depend on the operators' semantics.

The work presented in this paper will serve as a starting point for our future research in composite event detection. One of the most urgent problems is that depending on the semantics of the event algebra consistency issues may arise due to different and varying network delays. Although these problems also arise when event composition is done centrally, they are more difficult to tackle when composition is distributed. Furthermore, we want to address operators specified by the application which exhibit certain characteristics guaranteed to enable optimization. Another interesting issue is to include application components in our optimization. Finally, we want to investigate optimistic event detection. In this case, a composite event is fired although not all constituent events have yet arrived at the detector if the probability that the composite event will be completed is high. The detection of a composite event sooner but with a rest of uncertainty might in many applications be beneficial even if the composite event might be revoked.

References

[BB03] B. J. Bonfils, P. Bonnet. Adaptive and decentralized operator placement for in-network query processing. In *IPSN'03: Proceedings of the 2nd international conference on Information processing in sensor networks*. Pp. 47–62. Springer-Verlag, Berlin, Heidelberg, 2003.

[Bok81] S. Bokhari. A Shortest Tree Algorithm for Optimal Assignments Across Space and Time in a Distributed Processor System. In *IEEE Transactions on Software Engineering*. Volume SE-7 Issue:6, pp. 583 – 589. Nov. 1981.

[CKAK94] S. Chakravarthy, V. Krishnaprasad, E. Anwar, S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts and Detection. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*. Pp. 606–617. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.

[CRW01] A. Carzaniga, D. S. Rosenblum, A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.* 19:332–383, August 2001.

[GJS92] N. H. Gehani, H. V. Jagadish, O. Shmueli. Composite Event Specification in Active Databases: Model & Implementation. In *VLDB '92: Proceedings of the 18th International Conference on Very Large Data Bases*. Pp. 327–338. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[JPMH07] M. A. Jaeger, H. Parzyjegla, G. Mühl, K. Herrmann. Self-organizing broker topologies for publish/subscribe systems. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*. Pp. 543–550. ACM, New York, NY, USA, 2007.

[KMI+08] H. Kamiya, H. Mineno, N. Ishikawa, T. Osano, T. Mizuno. Composite Event Detection in Heterogeneous Sensor Networks. *IEEE/IPSJ International Symposium on Applications and the Internet* 0:413–416, 2008.

[MWJ+07] G. Mühl, M. Werner, M. A. Jaeger, K. Herrmann, H. Parzyjegla. On the Definitions of Self-Managing and Self-Organizing Systems. In *KiVS 2007 Workshop: Selbstorganisierende, Adaptive, Kontextsensitive verteilte Systeme (SAKS 2007)*. Informatik aktuell, pp. 291–301. VDE Verlag, Berlin, Mar. 2007.

[O'K10] D. O'Keeffe. Distributed Complex Event Detection for Pervasive Computing. Technical report 783, University of Cambridge, June 2010.

[PSB04] P. R. Pietzuch, B. Shand, J. Bacon. Composite Event Detection as a Generic Middleware Extension. *IEEE Network* 18(1):44–55, 2004.

[YLTX08] L. Ying, Z. Liu, D. F. Towsley, C. H. Xia. Distributed Operator Placement and Data Caching in Large-Scale Sensor Networks. In *INFOCOM 2008. 27th IEEE Intern. Conf. on Computer Communication*. Pp. 977–985. Phoenix, AZ, USA, Apr. 2008.