7

Composite Events

For certain applications, the expressiveness of subscriptions used by the local notification matching algorithms introduced in Chap. 3 is not sufficient. As a remedy, a service for *composite event detection* facilitates the management of a large volume of events by enabling subscribers to specify their interest more precisely. The composite event service supports the advanced correlation of events through the detection of complex *event patterns*. In this chapter we describe composite event detection services for publish/subscribe systems.

We start with two application scenarios that benefit from composite event detection in the next section. After that, we list the requirements for such a detection service (Sect. 7.2) and introduce composite events in more detail in Sect. 7.3. We then give an example of composite event detectors based on finite state automata (Sect. 7.4.1) and a corresponding language (Sect. 7.4.2). Composite events (CE) are detected by automata that support distribution and a flexible time model for composite events. Event subscribers of the composite event service use a core composite event language to specify event patterns using a series of operators. We also gave examples for three higher-level specification languages for composite events that are domain-specific. Section 7.5 has a discussion of centralized and distributed architectures for composite event detection. We also explain how distribution is controlled by distribution and detection polices. The design space for distribution polices gives rise to a variety of different strategies for distributing composite event expressions. We conclude the chapter with an overview of other composite event detection services in Sect. 7.6.

7.1 Application Scenarios

Many application scenarios for a publish/subscribe service benefit from a general-purpose composite event service that enhances the expressiveness of subscriptions. In the following we will consider how composite events can be

used in a ubiquitous computing environment and for network systems monitoring. For each application scenario, we provide two examples of composite event subscriptions.

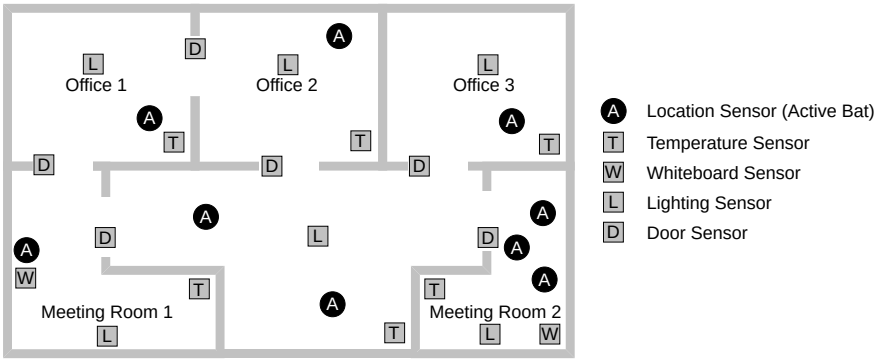


Fig. 7.1. The Active Office with different sensors

The Active Office

One example of a ubiquitous computing environment is the*Active Office*, a sensor-rich environment inside a computerized building (Fig. 7.1). In this building, sensors that are installed in offices provide information about the environment to interested devices, applications, and users. The Active Office is aware of its inhabitants' behavior and enables them to interact with it in a natural way. The large number of sensors potentially produce a vast amount of data. Building users wear *Active Bats* [5] that publish location information, and static sensors gather data about doors, lighting, equipment usage, and environmental conditions. However, information consumers prefer a high-level view of the primitive sensor data. Thus, a middleware used in this application scenario has to cope with high-volume data and be able to aggregate and transform it before dissemination. Composite event detection can help process the primitive events produced by the large number of sensors and provide a higher-level abstraction to users of the Active Office.

1. A user may subscribe to be notified when a meeting with at least three people working in the messaging department takes place during working hours in one of the meeting rooms.
2. Building services may be interested in composite events about a drop in temperature by 15 degrees for at least 15 min in any occupied office.

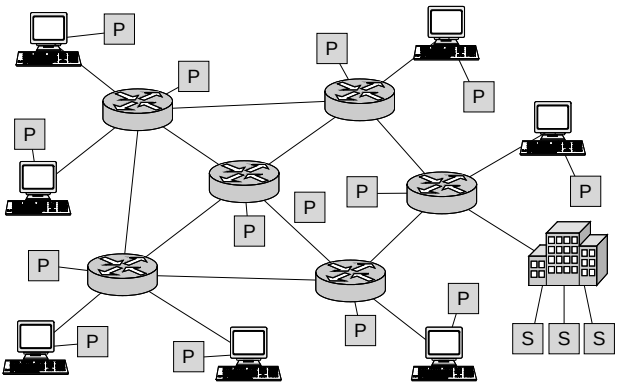


Fig. 7.2. A system for monitoring faults in a network

Network Systems Monitoring

When monitoring the operation of networks [49], network entities publish notifications that are related to fault conditions in the network, such as that shown in Fig. 7.2. In practice, millions of events may be published daily in respect of fewer than a hundred real faults that require human intervention. The task of network systems monitoring can thus be simplified by expressing patterns associated with real problems as composite event subscriptions.

1. The network management center may want to be notified when at least five workstations in different parts of the network detect a degradation in network bandwidth.
2. A network customer may be interested in composite events when none of its load-balanced Web servers are available to the outside world unless the downtime is part of scheduled maintenance work.

The XenoTrust Framework

Recent efforts, such as the XenoServer project [30], are building large-scale, public infrastructures for general-purpose, distributed computing with resource management and sharing. In such environments, reputation information about participants must be disseminated in a timely and scalable fashion so that entities can make trust-dependent decisions. Composite events can help participants receive notifications about changes in reputation of their resource providers or consumers, thus creating a global-scale trust management system [225].

1. A user may want to be notified when the reputation of any of its currently active resource providers drops below a certain threshold and there is an alternative provider that is capable of taking over the current resource contract.

2. A resource provider may submit a subscription causing a composite event when a new client receives a low reputation rating from at least three other providers within three days while requesting significant resources.

7.2 Requirements

From the above application scenarios for composite event detection, we derive several requirements for a composite event detection service.

- The composite event service must be *expressive* enough when it comes to the specification of composite events. Depending on the application domain, users will describe event patterns of varying complexity. The composite event service must naturally capture common use patterns.
- The composite event service must be *usable*. From a user's perspective, it must be easy to express complex event patterns in the composite event service. A too-expressive language for the specification of composite events may lead to poorly usability.
- The composite event service must be *efficient* in terms of the user's performance goals, such as low detection delay or bandwidth consumption. In particular, there must exist an efficient implementation technique for composite event detectors in the service. Often, a distributed implementation improves the service.

Obviously, there is a tension between these requirements: A very expressive composite event service may not result in an efficient or usable system. In contrast, a very efficient implementation of composite event detector may lead to a limited system with low expressiveness. In the following, we will describe a composite event service based on extended finite state automata that attempts to balance these trade-offs.

7.3 Composite Events

A composite event service is based on the notion of a *composite event*. Composite events prevent subscribers from being overwhelmed by a large number of primitive event publications by providing them with a higher-level abstraction. A composite event is published whenever a certain pattern of events occurs in the publish/subscribe system. This means that subscribers can subscribe directly to complex event patterns, as opposed to having to subscribe to all the primitive events that make up the pattern and then performing the detection themselves.

Often a subscriber is interested in the primitive events that caused a composite event. Therefore, when a composite event has been detected by the composite event service, it is published and contains all primitive events that

contributed to its occurrence. Since composite events are build from primitive ones according to a well-defined set of rules, every composite event can be assigned a *composite event type*. It is built from the types of the included events and the relation between them in the composite event subscription. Note that primitive events can also be considered degenerate composite events, thus unifying primitive and composite event types within the publish/subscribe system.

Definition 7.1 (Composite Event). *Every* composite event c *has a composite event type* τ_c *and belongs to the composite event space* $\mathbb{C}$*,*

$$(c : \tau_c) \in \mathbb{C},$$

A composite event type τ_c *corresponds to a valid expression* $\mathcal{C}$ *in a composite event language,*

$$\tau_c \equiv \mathcal{C}.$$

A composite event c *consists of an interval timestamp* t_c *and a set of composite subevents* $\{c_1, c_2, \ldots, c_k\}$*,*

$$c : \tau_c = (t_c, \{c_1, c_2, \ldots, c_k\}).$$

A composite event is associated with a timestamp t_c that states when it has occurred. In a distributed system, there is no concept of global time [227], which is why the timestamps of composite events caused by distributed sources can be captured more accurately using partially ordered *interval timestamps* [238]. An interval timestamp has a start and end time so that it can express the local clock uncertainty at an event broker and also the duration associated with a composite event from the first contributing event to the last. To capture the temporal relations between composite events, we define a partial and a total order over interval timestamps that will be used by the weak and strong transitions in the detection automata described in the next section.

Definition 7.2 (Interval Timestamp). *An* interval timestamp t_c*,*

$$t_c = [t_c^l; t_c^h],$$

has a start time t_c^l *and an end time* t_c^h *with* $t_c^l \leq t_c^h$*. Interval timestamps are partially-ordered* ($<$) *and totally-ordered* ($\prec$) *as follows,*

$$t_{c_1} < t_{c_2} \triangleq t_{c_1}^h < t_{c_2}^l,$$
$$t_{c_1} \prec t_{c_2} \triangleq (t_{c_1}^h < t_{c_2}^h) \vee (t_{c_1}^h = t_{c_2}^h \wedge t_{c_1}^l < t_{c_2}^l).$$

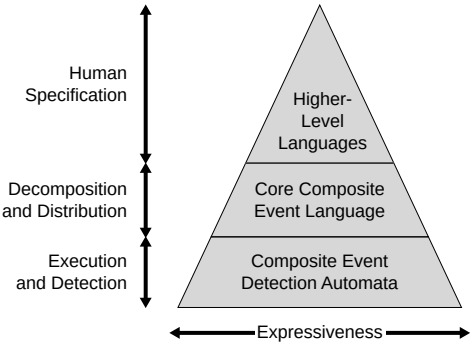


Fig. 7.3. The components of the composite event detection service

7.4 Composite Event Detection

From the description of the application scenarios, it becomes clear that it is challenging to design a single language for composite events that is both expressive and intuitive to use by, say, a human user in the Active Office environment. Therefore, an alternative approach is shown in Fig. 7.3, with several specification layers that have different powers of expressiveness. At the bottom layer, *composite event detection automata* provide maximum expressiveness and perform the actual detection of composite events described in the next section. Composite event subscriptions specified in the *core composite event language* presented in Sect. 7.4.2 can be decomposed for distributed detection. Finally, domain-specific *higher-level languages* constitute the top layer and only expose a subset of the core language—supporting a simpler definition of composite events for a given application domain. Expressions in higher-level languages are automatically compiled down to composite event detection automata by the composite event service.

7.4.1 Composite Event Detectors

In this section, we describe a composite event service that uses *composite event detection automata*, which are finite state automata [193] that are extended with support for temporal relationships and concurrent events, to analyze event streams. Basing composite event detection on extended finite state automata has several advantages. First, finite state automata are a well-understood computational model with a simple implementation. Second, their restricted expressive power has the benefit of limited, predictable resource usage, which is important for the safe distribution of detectors in the publish/subscribe system. Third, regular expression languages have operators that are tailored toward the detection of patterns, which avoids the risk of redundancy or incompleteness when defining a new composite event language. Finally,

complex expressions in a regular language may easily be decomposed for distributed detection.

A detection automaton consists of a finite number of states and transitions between them. To ensure that each state only has to consider certain events for transitions, it is associated with an *input domain* Σ, which is a generalization of the concept of an input alphabet in traditional finite state automata. An input domain is a collection of *describable event sets* $A, B, C, \ldots$, which correspond to sets of events that are matched by a primitive or composite event subscriptions. In a given state, only these events need to be considered by the automaton because other events are not relevant for the composite event being detected. In practice, the automaton issues subscriptions for all describable event sets in the input domain of a state. The resulting incoming events are ordered according to the total timestamp order ($\prec$) (see Def. 7.2) into an *event input sequence* and are consumed by the automaton sequentially.

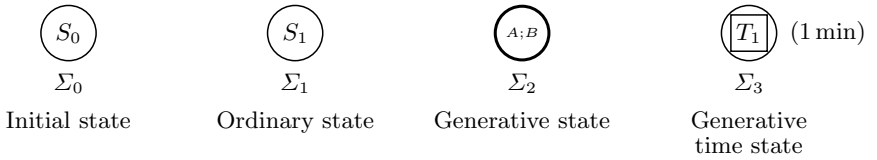


Fig. 7.4. The states in a composite event detection automaton

As shown in Fig. 7.4, a detection automaton has four types of state. Detection starts in a unique *initial state* and continues through a series of *ordinary states*. A *generative state* is an accepting state that also publishes the composite event that has been detected by the automaton. *Generative time states* deal with timing by publishing an internal *time event* when a timer (e.g., 1 min) associated with the state expires. The automaton treats time events like regular events, but they are not visible externally.

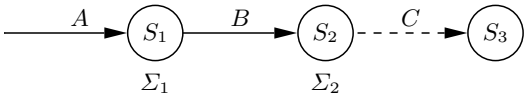


Fig. 7.5. The transitions in a composite event detection automaton

Each state can have two forms of outgoing transition that are labeled with the describable event sets of the events that trigger them. Note that since describable event sets can be defined by composite event subscriptions, our automata support the detection of event patterns involving concurrency. In the sample automaton in Fig. 7.5, the transition between states S_1 and S_2 is

a *weak transition* that requires the timestamps of the events from the describable event sets A and B to be partially ordered ($<$). A *strong transition*, such as between states S_2 and S_3, mandates a total ordering ($\prec$) between events from B and C. Strong and weak transitions therefore allow the expression of different temporal orderings between events. When an event that is part of the input domain but without a matching outgoing transition is received, the detection in the automaton fails. Several matching transitions and empty ϵ-transitions are followed nondeterministically. Although the following presentation of the detection automata uses nondeterminism, standard techniques can be used to convert them into deterministic automata [193].

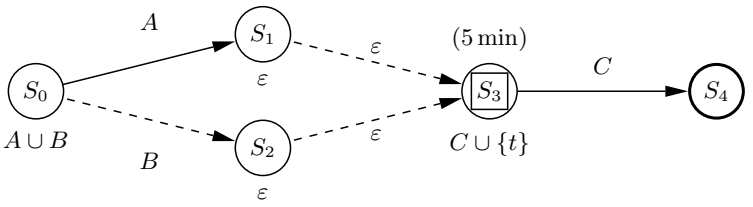


Fig. 7.6. A composite event detection automaton

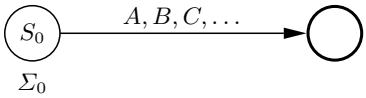
In Fig. 7.6, we give an example of a composite event detection automaton. This automaton starts in state S_0 with an input domain of $A \cup B$. A strongly followed event from A causes a transition to state S_1; a weakly followed event from B leads to state S_2. Once the generative time state S_3 is reached, a timer starts that will expire after 5 min, publishing time event t. Since this event is part of the input domain for state S_3 but there is no corresponding outgoing transition, detection will fail unless an event from C is received before the timer expires, triggering a transition to state S_4. The generative state S_4 signals the successful detection of a composite event with a composite event publication.

7.4.2 Composite Event Language

In a composite event detection service, composite event subscriptions are expressed in a composite event language. Expressions in this language define the set of composite events in which an event client is interested. In this section we describe a *core composite event language* that corresponds to the extended finite state automata introduced in the previous section. We present the language's operators and the construction of corresponding composite event detection automata from subautomata. Some operators in this language, namely concatenation, alternation, and iteration, are influenced by those found in regular languages. However, other operators reflect the special features of our detection automata. We finish with examples of core language expressions

and discuss three higher-level languages that can be built on top of the core language for domain-specific composite event specification.

Atoms. $[A, B, C, \ldots \subseteq \Sigma_0]$

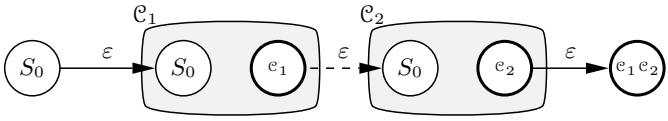


Atoms detect individual events in the input stream of all events that are in the input domain Σ_0. Here only events in the describable event sets $A \cup B \cup C \cup \ldots$ are matched and cause a transition to a generative state. Other events in Σ_0 result in failed detection, and events outside Σ_0 are ignored. The trivial atom $[A \subseteq A]$ is abbreviated as $[A]$.

Negation. $[\neg E \subseteq \Sigma] \triangleq [\Sigma \setminus E \subseteq \Sigma]$

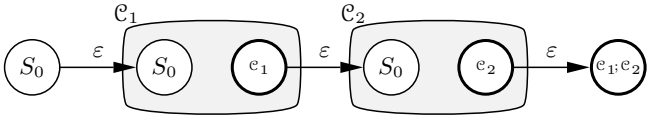
Negation is shorthand for an atom that matches all events in the input domain Σ except for events in the negated describable event set E. Note that this semantics differs from more powerful negation operators found in other event algebras.

Concatenation. $\mathcal{C}_1\mathcal{C}_2$



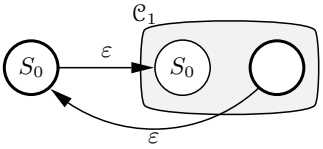
The concatenation operator detects a composite event matching expression $\mathcal{C}_1$ with a timestamp that *weakly* follows the timestamp of a composite event matching $\mathcal{C}_2$. The detection automaton for concatenation is constructed by connecting the generative state of $\mathcal{C}_1$ with a weak ϵ-transition to the initial state of $\mathcal{C}_2$.

Sequence. $\mathcal{C}_1; \mathcal{C}_2$



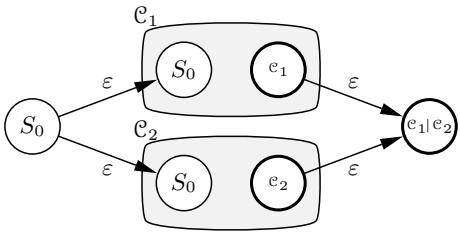
The sequence operator detects an event of type $\mathcal{C}_1$ *strongly* followed by an event of type $\mathcal{C}_2$. Unlike concatenation, this means that the interval timestamps of the events matching $\mathcal{C}_1$ and $\mathcal{C}_2$ must not overlap. The construction of the sequence detection automaton uses a strong transition for the ϵ-transition between the two subautomata.

Iteration. $\mathcal{C}_1^*$



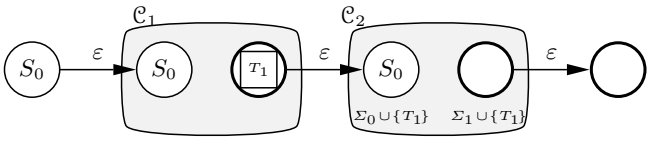
Any number of occurrences of $\mathcal{C}_1$ are matched by the iteration operator. Its detection automaton creates a loop from the generative state of $\mathcal{C}_1$ back to its initial state. If $\mathcal{C}_1$ receives an event that causes it to fail, then the composite expression $\mathcal{C}_1^*$ also fails.

Alternation. $\mathcal{C}_1 \mid \mathcal{C}_2$



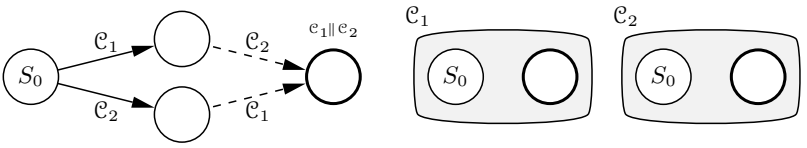
This composite event expression matches if either $\mathcal{C}_1$ or $\mathcal{C}_2$ is detected. The new automaton has an initial and a generative state with ϵ-transitions to both of the two subautomata introducing nondeterministic behavior.

Timing. $(\mathcal{C}_1, \mathcal{C}_2)_{T_1 = tspec}$



Timing relationships between composite events are supported by the timing operator that can detect event combinations within, or not within, a given time interval. This operator generates an event of type T_1 at the relative or absolute time specification *tspec* after a composite event of type $\mathcal{C}_1$ has been detected. The second expression C_2 may then use T_1 in its specification for atoms and input domains. Since time events are only locally visible, automata $\mathcal{C}_1$ and $\mathcal{C}_2$ must reside on the same node.

Parallelization. $\mathcal{C}_1 \parallel \mathcal{C}_2$



The final operator is parallelization and allows detection of two composite events $\mathcal{C}_1$ and $\mathcal{C}_2$ in parallel, only succeeding if both are detected. Unlike alternation, any interleaving of the two composite events is permitted. The detection automaton for parallelization is constructed by creating a new automaton that uses the composite events detected by $\mathcal{C}_1$ and $\mathcal{C}_2$ for its transitions.

Examples

The following examples illustrate valid expressions in the core composite event language. Let the describable event set A represent events corresponding to the subscription that "Alice is in the office", let $\bar{A}$ be "Alice has left the office", let B be "Bob is in the office", and let P be "anyone is in the office", as detected by an Active Bat.

1. $[A];[B]$. Alice enters the office followed by Bob.
2. $[A \subseteq \{A, B\}]$. Alice enters the office before Bob.
3. $([A], [B \subseteq \{B, T_1\}])_{T_1 = 1\,\mathrm{h}}$. Alice enters, and Bob follows within 1 h.
4. $[\bar{A}]\,[\neg A \subseteq P]\,[A]$. Someone else enters the office when Alice is away.

Higher-Level Composite Event Languages

We can now use the core composite language as a basic building block for other composite event detection languages. In general, when designing a language for composite event detection, we have two conflicting requirements. On one hand, the language should be *machine processable* so that it supports the efficient creation of composite event detection automata and the automatic decomposition of expressions for distributed detection. On the other hand, the syntax and semantics of the language should be high-level and intuitive, facilitating the task of writing expressions by programmers or end users. This means that the language should be *human processable*. To unify these two requirements, one can define higher-level composite event languages for the specification of composite events in a natural and domain-specific way. Expressions from higher-level languages are then translated automatically into the core language described above. The following are three possible higher-level composite event languages.

Pretty Language

The "pretty" language has a more verbose syntax compared to the core language and resembles rule-based specification languages found in active database systems. It has a redundant set of operators, and its specifications are close to English language statements. A composite event specification, such as

`Event_A followed_by Event_B within 1 hour`,

makes it easier for nonprogrammers to use composite events.

Programming Language Binding

This binding provides programming language-specific access to composite event specification. It avoids having to deal with a special composite event language by allowing the construction of composite event expressions from method calls, such as

`eventA.after(eventB.repeated(3))`.

At runtime these method calls are translated into core composite event language expressions for detector construction.

Graphical Composition Model

In a ubiquitous computing environment, a user-friendly way for composite event specification is needed that makes it easy for users to interact with the system at runtime. Composite events, such as "Turn the office light out after 7 pm when the office is empty", can be described using a graphical composition tool that is based on a simple model familiar to users. For example, composite event streams could be visualized as water flows with different forms of piping for the construction of composite event expressions [194].

7.5 Detection Architectures

In this section we present an architecture for a composite event service. The design requirements for the service can be derived from the above application scenarios. In general, the service should be applicable to a wide range of publish/subscribe designs and therefore should make few assumptions about the underlying publish/subscribe implementation. Ideally, it should only rely on standard interfaces provided by the publish/subscribe system and not require special extensions to the event model. For example, content-based routing and filtering support should be exploited for the dissemination of composite events. To satisfy the requirement of scalability, composite event detection can be distributed, decomposing complex composite event subscriptions into subexpressions and detecting them at different nodes in the system.

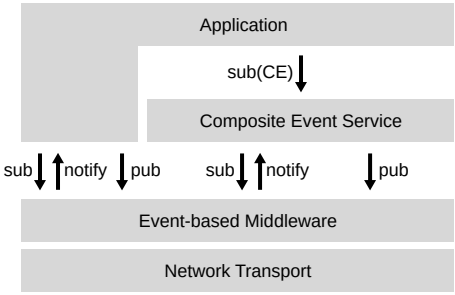


Fig. 7.7. The architecture for the composite event detection service

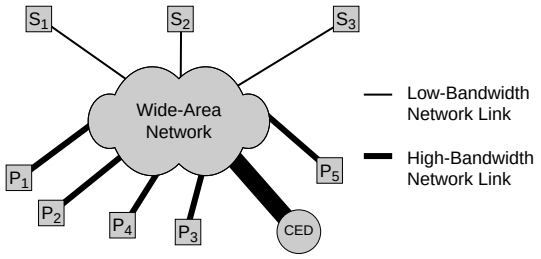


Fig. 7.8. Illustration of centralized composite event detection

The architecture for a general composite event service is shown in Fig. 7.7. The service uses the event client API of the publish/subscribe system so that composite event detectors can subscribe to primitive events and detect the occurrence of composite events. The publish/subscribe system is also used to coordinate the detection of decomposed composite event expressions and publish detected composite events. Note that the publish/subscribe system does not need to be aware of composite event types because composite event publications can be disguised using new primitive event types. Content-based routing and filtering of events is carried out by the publish/subscribe system. An application with event clients can either use the composite event service to submit composite event subscriptions and cause the instantiation of detectors, or interact directly with the publish/subscribe system for normal middleware functionality.

7.5.1 Centralized Detection

The most straightforward architecture for a composite event detection service is centralized, as shown in Fig. 7.8. In a centralized architecture, a single composite event detector (CED) is subscribes to all primitive events that may contribute toward the detection of composite events. When a composite event

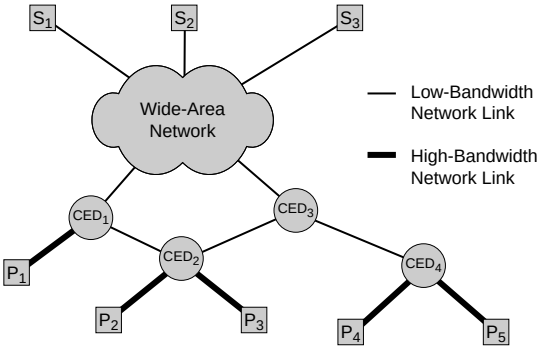


Fig. 7.9. Illustration of distributed composite event detection

has been detected, a new composite event notification is published by the detectors.

An obvious disadvantage of a such an approach is that the centralized event detector can become in a bottleneck in a large-scale system. This may happen if the network bandwidth or processing resources at the detection site are insufficient to keep up with the stream of incoming primitive events. In addition, a centralized detector wastes network bandwidth because many primitive events are sent to the detector over the network only to be discarded there. A distributed implementation of the composite event detection service is more complex but has the advantage that primitive events can be discarded close to event publishers.

7.5.2 Distributed Detection

A composite event service can also implement the detection of composite events in a distributed fashion. This is achieved by decomposing expressions from the composite event language into subexpressions that are detected by separate detectors distributed throughout the system. The support for the decomposition of composite event expressions allows popular subexpressions to be reused among event subscribers, thus saving computational effort and network bandwidth. In particular, the amount of communication is reduced because detectors for subexpressions can be positioned close to primitive event publishers that produce the events necessary for detection. Subexpressions can also be replicated for load balancing and increased availability, and computationally expensive expressions can be decomposed to prevent any detector from becoming overloaded.

A system that benefits from distributed composite event detection is shown in Fig. 7.9. The composite event detectors CED_{1-4} for subexpressions are located close to the primitive event publishers P_{1-5} that publish events at a high rate and therefore must be connected through high-bandwidth network

links. Low-bandwidth links in a wide-area network are used to connect the composite event subscribers S_{1-3}. The traffic on these network links is significantly lower because fewer event publications need to be transmitted after composite event detection. Since each detector subscribes to at most two event streams, no detector can get overwhelmed by the event rate.

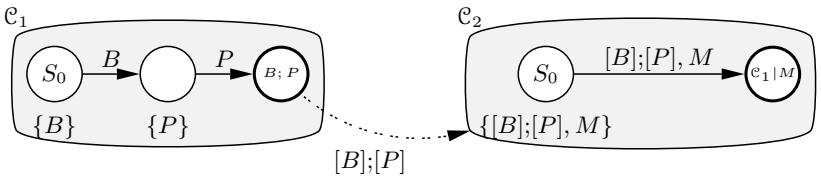


Fig. 7.10. Two cooperating composite event detectors for distributed detection

The detection automata described earlier directly support distribution because they can subscribe to composite events detected by other automata in the publish/subscribe system. In Fig. 7.10 the two automata $\mathcal{C}_1$ and $\mathcal{C}_2$ cooperate in order to detect the composite event expression $([B];[P]) \mid [M]$. The subautomaton $\mathcal{C}_1$ detects the expression $[B];[P]$, which is then used by $\mathcal{C}_2$ in the event input domain and transition of state S_0. When this composite event is received, it causes a transition to the generative state S_1. Next we present the capabilities of mobile composite event detectors.

Mobile Composite Event Detectors

A *mobile composite event detector* implements the distributed detection of composite events. Mobile composite event detectors are agentlike entities cohosted at event brokers that encapsulate one or more composite event detection automata for expressions from the core composite event language. They can subscribe to event publishers (and other mobile detectors) and publish the composite events detected by their automata. In addition, a mobile detector can move from one event broker to another in order to optimize the detection of composite events in the system.

When an event subscriber submits a new composite event subscription, a mobile detector is instantiated at an event broker and is then responsible

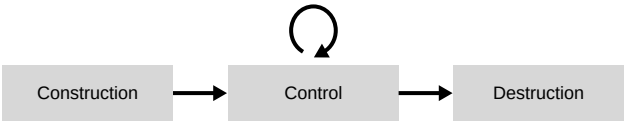


Fig. 7.11. The life cycle of a mobile composite event detector

for the detection of the new expression. The life cycle of a mobile composite event detector is summarized in Fig. 7.11. In the *construction phase*, the mobile detector establishes the detection of the new composite event subscription by cooperating with other existing mobile detectors. It then enters a *control phase*, during which the detection is optimized by adapting to dynamic changes in the environment and ensuring that it maintains compliance with distribution and detection policies described below. Finally, a *destruction phase* is reached when the mobile detector is no longer required because all event clients have unsubscribed or other detectors have made it redundant.

While in its control phase, a mobile detector can carry out several actions that are governed by distribution policies explained in the next section.

1. It can *instantiate* new automata for the detection of new composite event expressions or any subexpressions.
2. For distributed detection, it can decompose composite event expressions and *delegate* detection to other, already existing mobile detectors.
3. The mobile detector can *migrate* to another event broker that, for example, is closer to the event publishers that the detector has subscribed to.
4. Finally, it can *destroy* any of its composite event detection automata that are no longer required.

Distribution Policies

A remaining difficulty is the decision on an optimal strategy for the decomposition of composite event expressions and the placement of composite event detectors in the system. This is complicated by the fact that the requirements for distributing detectors are potentially conflicting. For example, to minimize usage of network bandwidth, existing detectors should be reused for subexpressions as much as possible. However, if low notification latency is important, detectors should be replicated in various parts of the network, thus leading to increased bandwidth consumption. An optimal solution is a trade-off that takes the static and dynamic characteristics of the application and the network into account.

To make these trade-offs explicit, we introduce the notion of a *distribution policy*, which is a set of heuristics that governs the actions of mobile composite event detectors in the control phase. Each composite event subscription submitted to the composite event service includes its own distribution policy for detection, depending on the application requirements of the event subscriber. During their lifetime, mobile composite event detectors attempt to comply with their distribution policy. Some distribution policies may require the aggregation of network or event broker statistics by mobile composite event detectors, such as communication latency or computational load. When defining distribution policies, three independent dimensions can be identified that help restrict the design space, as shown in Fig. 7.12.

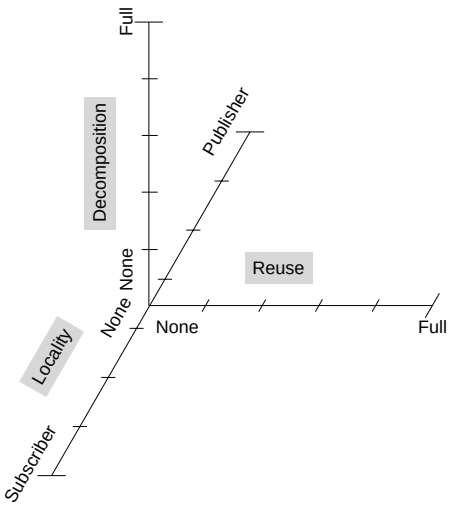


Fig. 7.12. The design space for distribution policies

Decomposition. The degree of decomposition of the composite event expression must be stated in the policy (with optional hints from the application). In order to reuse existing detectors in the system, an expression may have to be decomposed into subexpressions. Decomposition may increase the reliability of detection if multiple detectors are detecting overlapping expressions. For load-balancing reasons, a complex expression may be decomposed into manageable subexpressions. The degree of decomposition ranges from no decomposition to full decomposition, where every possible subexpression is factored out. Some policies allow decomposition only when there already exist detectors that can be reused for a subexpression.

Reuse. This dimension specifies to what extent already existing detectors are reused for a new composite event expression or any of its subexpressions. Not reusing existing detectors can result in more reliability, whereas maximum reuse will save bandwidth and computational effort. In situations in which detection latency is important, only local detectors that are in close proximity should be reused.

Locality. The location of new mobile composite event detectors must be determined. For certain scenarios, bandwidth usage can be reduced by moving detectors as close to primitive event sources as possible. Primitive events that constitute a composite event may be of interest only to the CE detector and should therefore not be widely disseminated throughout the entire system unnecessarily. This is called *publisher locality*. The opposite approach is to put new CE detectors close to application components that subscribe to them to improve reliability and detection latency. This leads to a policy with *subscriber locality*.

Table 7.1. Example of five distribution policies

Policy Name	Decomposition	Reuse	Locality
Minimum Latency	None	With locality only	Subscribers
Minimum Bandwidth	For reuse only	Max	Publishers
Minimum Impact	For reuse only	Max	None
Minimum Load	Max	Max	None
Maximum Reliability	For reuse only	At least 2	None

In practice, only certain combinations of these three dimensions will result in useful distributions policies. Table 7.1 summarizes five example policies that each attempt to optimize a different metric in the composite event framework.

Minimum Latency Policy. The detection latency is minimized by placing new detectors as close to subscribers as possible. Composite event expressions should not be decomposed into subexpressions as this would increase the detection latency. Similarly, an existing detector should only be reused if it is close to the subscriber and detects exactly the required composite events.

Minimum Bandwidth Policy. Bandwidth consumption is minimized by placing the detectors close to the primitive event publishers, leveraging the filtering aspect of composite event detectors. In addition, existing detectors should be used as much as possible so that no new traffic is generated. The reuse of subexpressions may lead to decomposition.

Minimum Impact Policy. This policy minimizes the impact that new detectors have on the entire system. This involves minimizing bandwidth, as before, but also means that computational load should be spread out evenly among detectors. Therefore, new detectors do not have locality, but existing detectors should be maximally reused.

Minimum Load Policy. The fourth policy minimizes the load on composite event detectors by decomposing an expression into the smallest possible subexpressions and distributing them evenly among detectors in the system. It attempts to reuse already existing detectors.

Maximum Reliability Policy. The last policy makes the composite event detection more resistant to node failure by instantiating redundant detectors for extra reliability. Old detectors are reused only when at least two already exist; new detectors are created otherwise. (This "at least 2" partial reuse policy lies between no reuse and full reuse in Fig. 7.12). To limit extra points of failure, detectors are decomposed for reuse only, and no locality restrictions are imposed on new detectors.

Note that a distribution policy is associated with a particular CE expression, so that every mobile CE detector can have its own policy. This enables event subscribers to specify a desired distribution policy at subscription time depending on application requirements. The effectiveness of distribution poli-

cies can be enhanced when mobile CE detectors are able to obtain network- and system-specific parameters such as the current load of a broker node or the communication latency to a particular publisher. A mobile CE detector may use this information to optimize detection in compliance with its distribution policy.

Detection Policies

In a distributed system, events from different event sources travel along separate network routes to a mobile CE detector. Even if we assume that the network itself does not reorder events, out-of-order arrival of events at the detector can occur because of the different associated network delays. Whenever a new event arrives, it has to be inserted at the correct position in the totally ordered event input stream before the stream is fed into the automaton.

The problem is to decide when the next event in the event input stream can be safely consumed by the automaton without risking that an event with an older timestamp is still being delayed by the network. Premature consumption could lead to an incorrect detection or nondetection of a composite event. Thus, each CE subscription is annotated with a *detection policy* that specifies when a detector can consume an event from an event input stream.

Best-Effort Detection. A best-effort detection policy states that events are consumed from event input streams without delay. Whenever an event is available, it will cause a state transition (or failure) in the automaton. Although this policy may lead to incorrect detection, it can be applied by applications that are sensitive to detection delay and are willing to ignore false positives.

Guaranteed Detection. Under a guaranteed detection policy, an event is consumed from an event input stream only once it has become *stable*[1] [238]. The consumption of only stable events ensures that no spurious composite events are detected. In our model, we assume that the network itself does not reorder events autonomously so that events coming from the same event source can be expected to arrive in chronological order at a detector. A detector knows that an event is stable and can be consumed after another event with a later timestamp from the same event source has been inserted in the event input stream. An event source that does not publish events at a high enough frequency can publish dummy *heartbeat events* that are used to "flush the network".

In an asynchronous distributed system, a guaranteed detection policy potentially introduces an unbounded delay at the detector. For instance, an event source might fail or decide not to cooperate by not sending heartbeat events. This could prevent the detector from consuming any events of that

[1] An event is stable if there is no other event with an earlier timestamp in the system that should be part of this event input stream and should thus be consumed instead.

type. To avoid this problem, we are currently investigating a *probabilistic stability* metric. As opposed to a simple binary stability measure, a detector attempts to model the probability that a particular event in an event input stream is stable and the event is only consumed if its stability metric is above a given threshold.

7.6 Further Reading

In this section we provide an overview of related work on composite event detection. A more detailed description of distributed composite event detection can be found in [314]. Composite event detection first arose in the context of triggers in active database systems. Other related application areas are network systems monitoring and the interaction with ubiquitous computing environments. In general, distributed publish/subscribe systems leave the detection of composite events to the application programmer. An exception is Siena (described in Sect. 9.3.2), which includes restricted event patterns without defining their precise semantics or giving a complete pattern language. A service for the detection of composite events using Corba is presented by Liebig [238]. Similar to the described composite event service, it uses interval timestamps to make the uncertainty of timestamps in a distributed system explicit. The notion of event stability is introduced to handle communication delays. A system and language for *complex event processing* is proposed by Luckham [242]. The *Rapide* language [243] supports the specification of event patterns in areas such as process management, network monitoring, and enterprise management. Event patterns are detected using event processing agents that have access to event histories and mine the event stream.

Active Database Systems

Composite event detection in active database systems is usually not distributed. Early languages for triggers follow an event–condition–action (ECA) model [105, 304] and resemble database query algebras with an expressive, yet complex, syntax. In the *Ode* object database [173], composite events are specified with a regular language and detected using finite state automata. Equivalence between the language and regular expressions is shown. Since a composite event has a single timestamp—that of the last primitive event that led to its detection—a total event order is established that does not deal with time issues. Composite event detectors based on Petri nets [307] are used in the *SAMOS* database [170]. Colored Petri nets can represent concurrent behavior and store complex event data during execution. A disadvantage is that even for simple composite event expressions, Petri nets quickly become complicated. SAMOS does not support distributed detection and has a simple time model. The motivation for *Snoop* [74] was to design an expressive composite event language with temporal support. A detector in Snoop is a

tree that mirrors the structure of the composite event expression. Its nodes implement language operators and conform to a given *consumption policy*. A consumption policy determines the semantics of operators by resolving the order in which events are consumed from an event history. For example, under a *recent* consumption policy only the event that most recently occurred is considered and others are ignored. Detection then propagates up the tree with the leaves being primitive event detectors. A drawback of this approach is that detectors are Turing-complete, which makes it difficult to estimate their resource usage in advance. In addition, consumption policies influence the semantics of operators in a nonintuitive and operator-dependent way. For simplicity we have decided to only support a *chronicle* consumption policy.

Distributed Systems Monitoring

Similar to network systems monitoring in Sect. 7.1, composite events can be used for the monitoring of distributed systems. Schwiderski presents a distributed composite event monitoring architecture [339] based on the 2g-precedence time model. This model makes strong assumptions about the clock granularity that are not valid in large-scale, loosely coupled distributed systems. The composite event language and detectors are similar to Snoop and suffer from the same shortcomings. The work addresses the issue of delayed events in distributed detection by *evaluation policies*. *Asynchronous* evaluation allows a detector to consume an event without delay, whereas *synchronous* evaluation forces it to wait until all earlier events have arrived, as indicated by a heartbeat infrastructure. Although the detection can be made distributed, the placement of detectors in the system is left to the user. The *GEM* system [250] has a rule-based event monitoring language. It also follows a tree-based approach and assumes a total time order. Communication latency is handled by annotating rules with tolerable delays, which may not be feasible in an environment with unpredictable delays, such as a large-scale distributed system.

Ubiquitous Systems

Research efforts in ubiquitous computing have resulted in composite event languages that are intuitive to use by users of environments such as the Active Office. The work by Hayton [189] on composite events in the Cambridge Event Architecture defines a language that is targeted at nonprogrammers. Push-down finite state automata are used to detect composite events, but the semantics of some of the operators is nonintuitive. Although detection automata can use composite events for input, distributed detection is not handled explicitly and only scalar timestamps are used in the time model.