Combining Proactive and Retroactive Processing for Distributed Complex Event Detection

Mert Akdere
Department of Computer Science
Brown University
makdere@cs.brown.edu

Abstract

Complex Event Detection (CED) is a key capability for many monitoring applications such as intrusion detection, sensor-based activity/phenomenon tracking, and network/infrastructure monitoring. Existing CED solutions commonly assume centralized availability and proactive processing of all relevant events, and thus incur significant overhead in distributed settings. In this paper, we present and evaluate efficient distributed CED techniques that reduce event detection and transmission costs through a combination of proactive and retroactive processing strategies. The key idea is to generate CED plans that leverage the temporal and spatial windowing constraints associated with complex events to determine a multi-step acquisition order of constituent events that minimizes expected communication costs while meeting application-defined latency bounds for event detection. We demonstrate the utility of the proposed technique using extensive experimentation on a variety of workload scenarios.

1. Introduction

In this paper, we study the problem of complex event detection (CED) in a distributed monitoring environment that consists of potentially a large number of distributed event sources (such as hardware sensors or software receptors). CED is becoming a fundamental capability in many domains including network and software infrastructure security (e.g., denial of service attacks and intrusion detection), phenomenon and activity tracking (e.g., fire detection, storm detection, tracking suspicious behavior in an airport). It is often the case that such sophisticated (or "complex") events cannot be detected by individual sources at a single time and location: complex events usually take place over a period of time and region, thus require consolidation of many "simple" events from multiple sources.

The traditional means for CED (as exemplified in stream processing systems and traditional databases) is based on a centralized, push-based processing model. Sources generate simple events, which are continually pushed to a base where the registered complex events are evaluated in the form of continuous queries or triggers. This exclusively push-based, "proactive" model of processing is inefficient in distributed environments as it leads to significant communication overhead that may deplete batteries or hog network pipes (especially considering the fact that while many complex events are rare, some of the constituents elements may be generated relatively frequently).

Before we introduce our approach, we first make two observations. (1) *Local storage*: Event sources usually have storage capabilities (albeit limited) that enable them to keep some of their data for short-medium periods of time. Clearly, the available storage capacity depends on the hardware platform, but even with tiny devices, storage is fast becoming a non-issue due to the advances in flash- and similar technologies. (2) *Delay tolerance*: While timely detection of events is critical, applications often have varying timeliness requirements. For example, fire or storm detection exhibits much higher tolerance to delay than network intrusion.

The key topic of this paper is an approach for communication-efficient complex event detection that leverages these two observations. Given a complex event, we proactively monitor only a subset of the simpler elements as the first step, and only if they occur, we then "retroactively" check for the existence of others at the appropriate sources as the consequent step and iterate the algorithm. As such, our hybrid proactive-reactive algorithm generates a multi-step plan of event acquisition where rarer events are checked before more frequent events, thus in many cases eliminating the need for communicating the latter. To make this approach work, sources use their local storage to store their events for a pre-determined duration of time in case they need to be retroactively consulted. As each step in the algorithm introduces an additional delay, the algorithm also limits the number of steps based on application-specified per-event latency bounds.

In addition to our basic CED algorithm, the other contributions of the paper are as follows:

- A simple but expressive set of event composition operators decorated with time and space constraints (including usage examples).
- An extension that leverages temporal and spatial constraints (when available and applicable) to further reduce event transmissions.
- An extension that leverages shared sub-events that are

common to multiple complex events.

- Extensive experimentation that characterizes and quantifies the behavior and benefits of the algorithm and its extensions on a variety of workloads.

The rest of the paper is structured as follows. An overview of the system and its functionality is provided in Section 2. In Section 3, we present our event language together with usage examples. Then, we describe our multi-step approach to event detection that uses a cost model based on event occurrence probabilities for estimating monitoring costs in Section 4. We provide experimental results in Section 5. Related work is covered in Section 6 and Section 7 concludes the paper.

2. System Overview

We present a complex event detection framework for distributed monitoring applications. Our framework uses a plan-based approach to complex event detection and utilizes probabilistic models of event occurrences in finding network efficient event detection plans. Using this approach our system incurs low network cost during times of inactivity and is able to detect complex events quickly within user specified deadlines once they occur.

2.1 Complex Event Model

Events are defined as activities of interest in a system [8]. Detection of a person in a room, the firing of a cpu timer, and a denial of service attack in a network are example events from various application domains. All events signify certain activities, however their complexity degrees can be significantly different. For instance, the firing of a timer is instantaneous and simple to detect whereas detection of a denial of service attack is an involved process that requires computation over many simpler events. Correspondingly, events are categorized as complex and primitive forming a hierarchy of events.

At the base of the hierarchy are the primitive events, depicted by *bottom layer* events in Figure 1. Primitive events are defined as atomic occurrences of interest in a system. For example, a temperature reading in a sensor network and detection of a book in an RFID enabled library are examples of primitive events. Complex events form the upper levels of the hierarchy. They are built on top of simpler events, either primitive or complex, using our event specification language defined in section 3. Middle and top layers in Figure 1 represent the complex event layers.

All events are assigned a time interval that indicates their occurrence intervals. For primitive events, the time interval represents a single timepoint where the event occurs. For complex events, the assigned intervals contain the time intervals of all subevents. Hence, we use interval based semantics instead of timepoints. The reason is that interval semantics better represent the underlying structure and also

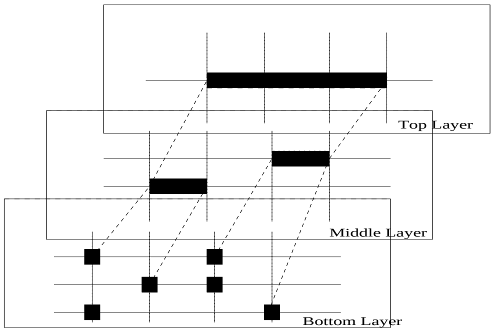


Figure 1. Illustrating Event Hierarchies: Complex events map to simpler events whereas primitive events lie at the bottom of the hierarchy.

solve certain problems that arise with timepoints. As an example, consider a complex event defined as the sequence of events *a* and *b* (see Figure 2). If timepoint based semantics are used then we only know the endpoints of events and would therefore detect the sequence complex event in Figure 2 since b happens after a. On the other hand, if interval semantics are used then the start times indicate that b actually started before a occurred which prevents the detection of the complex event. This is the required semantics if causal relations between events are to be observed. This issue is further discussed in [7].

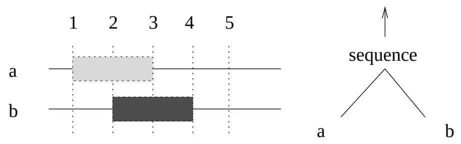


Figure 2. Point based semantics cause incorrect detection of the complex event *a sequence b*. With interval semantics, the complex event is not detected since event *b* starts before *a* occurs.

2.2 System Architecture

The main components in our system are the information sources and the base node (see Figure 2.2). The information sources, which in a broad sense we refer to as *sensors*, are the entry points of information into the system. For instance, routers and firewalls in a network monitoring application, and a wireless temperature sensor in a disaster monitoring application are example information sources. In addition to gathering information, sensors also take part in low level processing of information. The processing done by sensors include the generation of primitive events, the simplest operational units in the system. Finally, we assume that sensors have data logging capabilities. These data logs provide us with the ability to reach historical data as well as current data which is crucial for retrospective event detection.

Base station is the central component of the system that plans and executes the complex event detection. It generates event detection plans based on the hierarchical structure of complex events, chooses a plan to execute using the information from cost model and coordinates the execution of the chosen plan among the sensors. For this reason, the base

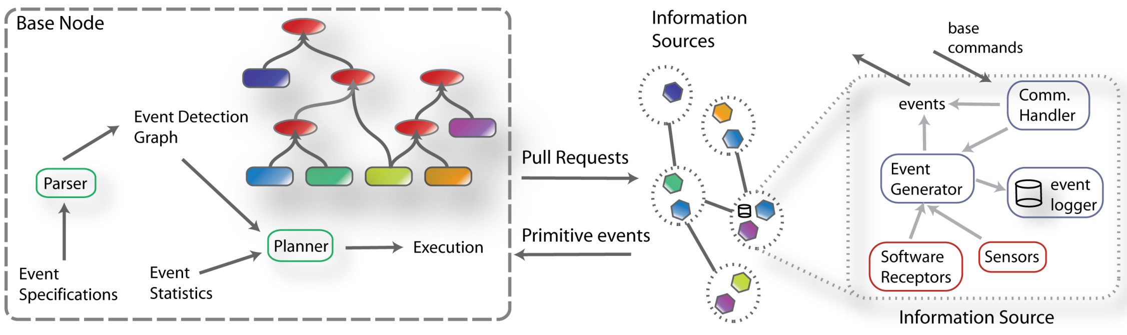


Figure 3. Complex event detection framework: The base node plans and coordinates the event detection using low network cost event detection plans formed by utilizing event statistics. The event detection model is an event detection graph generated from the given event specifications. Information sources feed the system with primitive events and can operate both in pull and push based modes.

station is provided with the ability to manage the sensors. This ability is significant since sensors transmit the detected events on demand from the base station. Therefore our system combines the pull and push paradigms of data collection to avoid the disadvantages of a push-based centralized system. We try to reduce the network traffic towards base station by carefully choosing which sensors will transmit data based on the information we have about frequency and constraints (such as spatial) of event types.

Our event detection model is based on an event detection graph constructed from the user given event specifications expressed in our language. For every event expression we construct an event detection tree and these event trees are then merged to form the event detection graph. Common events in different event trees, the shared events, are merged to form nodes with multiple parents. Nodes in an event detection graph are either operator nodes or primitive event nodes. All the non-leaf nodes are operator nodes which execute the event language operators on their inputs. The inputs to operator nodes are events (either complex or primitive) coming from their child nodes and their outputs are complex events. The leaf nodes in the graph are primitive event nodes. A primitive event node exists for each primitive event type and stores references to the instances of that primitive event type.

3. Complex Event Language

In this section, we describe a SQL-like declarative event specification language that is simple yet expressive enough for a variety of monitoring applications we have considered. Tha language includes event operators to express event correlations, similar to the specification of triggers in active database systems, and also contains other features such as the time windows from stream processing systems. At this point, we would like to emphasize that our main contribution is not the language itself but our efficient complex event detection techniques which operate based on the event specifications.

All the basic information in the system comes from the available information sources. The types and capabilities of information sources (referred to as sensors hereafter) depend on the application environment and could range from wireless cameras in a visual sensor network to logs of a web server. An output specification of each sensor type is necessary for the low level sensor information to be transformed into primitive events. More specifically, sensor types need to be introduced into the system through our event language with a name and a schema describing their attributes.

Every event type (primitive or complex) is associated with a set of attributes, forming its schema, in its declaration. Certain attributes such as location and event identifier are required for all events. Those attributes, *event_id*, *loc*, *start_time*, *end_time* and *node_id*, form a base schema that must be extended by the schemas of every event type. *event_id* is an identifier assigned to every event instance. It can be made unique for every event instance or can be set to a function of event attributes for *similar* event instances to get the same id. For example, in an RFID enabled library application a book might be detected by multiple RFID receivers at the same time. Such readings can be discarded if they are assigned the same event identifier. *loc* attribute is for storing the location of the event. *start_time* and *end_time* represent the time interval of the event and are assigned by the system based on the event operator semantics explained in Section 3.3. The last attribute, *node_id* is the id of the node that generated the event. All base schema attributes will be implicitly defined unless they are explicitly specified. Finally, a reserved but nonmandatory attribute, named *maxlatency*, is used to specify the latency deadline for an event type. When a latency deadline is specified, the system will only consider the plans satisfying the latency requirement.

3.1 Primitive Event Declaration

Primitive events, the simplest units in the event hierarchy, are formed by annotating sensor readings with metadata. Primitive event declarations specify the details of the transformation from sensor readings into primitive events. The syntax for primitive event declaration is given in Figure 4.

```
 primitive name
        on sensor_list
  schema attribute_list
```

Figure 4. Primitive events are defined using sensor information.

Each primitive event is assigned a unique name using the *name* symbol. The set of sensors used in a primitive event is listed in the *sensor_list*. Multiple sensors may be used in this list given that they lie on the same platform. We provide the pseudo-sensor *node* which enables access to context information such as the location of the sensor node and the current value of node clock. *schema* section is used to express the attributes of the primitive event type and the way they are assigned values. The attributes listed in the schema must be a super set of the base schema. An example primitive event, expressing a person detection, is given in Figure 5 together with the declaration of a *person_detector* sensor (such as a face detection algorithm running on a camera).

```
 sensor   person_detector
 schema int id, double loc_x, double loc_y

primitive person_detected
       on person_detector as PD, node
  schema event_id as hash(person_detected, node.id, node.time, PD.id),
           loc as [ PD.loc_x, PD.loc_y ],
           person_id as PD.id
```

Figure 5. The *person_detected* primitive event is defined using the *person_detector* and *node* sensors.

3.2 Complex Event Declaration

Complex events are specified on simpler subevents using the SQL-like template shown in Figure 6. Subevents of a complex event type, which can be previously specified complex or primitive events, are listed in the *source_list*. The source list may contain the *node* pseudo-sensor as well.

```
 complex name
       on source_list
   schema attribute_list
    where constraint_list
```

Figure 6. Complex events are specified using simpler events on which spatial, temporal or attribute-based constraints can also be imposed.

The *attribute_list* contains the attributes of a complex event type which together form a super set of the base schema and also describes the way they are assigned values. In this sense, the schema section specifies the transformation from subevents to complex events. Constraints of a complex event type are specified in the *constraint_list*. We discuss constraint specification in more detail in Section 3.3.

3.3 Constraint Specification

In most applications, users will be interested in complex events which impose constraints on their subevents. For instance, users may want to monitor events occurring in nearby locations or same time intervals. In order to support such constraints, our system allows temporal, spatial, attribute-based and existential constraints to be specified in the where clause of a complex event specification.

We borrowed event operators from active database research for easy specification of temporal correlations between subevents which could otherwise be expressed as a set of attribute constraints on start and end times. Our event operators, *and*, *or* and *seq*, are all *n-ary* operators. We also extended the event operators with time windows for temporal constraint specification. The time window argument, w, of an event operator specifies the maximum time between any two subevents of a complex event instance. Hence, all the subevents are separated by at most w time units. Formal semantics of our operators are provided below where we denote subevents with $e_1, e_2, \ldots, e_n$ and the start and end times of the output complex event with t_1 and t_2.

And operator: $and(e_1, e_2, \ldots, e_n; w)$
The *and* operator outputs a complex event with $t_1 = min_{i\in\{1,..,n\}}(e_i.start\_time), t_2 = max_{i\in\{1,..,n\}}(e_i.end\_time)$ if $max_{i,j\in\{1,..,n\}}(e_i.end\_time - e_j.end\_time) <= w$.

Sequence operator: $seq(e_1, e_2, \ldots, e_n; w)$
The *seq* operator outputs a complex event with $t_1 = e_1.start\_time$, $t_2 = e_n.end\_time$ if (a) $e_i.end\_time < e_{i+1}.start\_time$ for $i = 1, \ldots, n-1$, (b) $e_n.end\_time - e_1.end\_time \leq w$. Hence, *seq* is a restricted form of *and* where overlapping is not allowed and events need to occur in order.

Or operator: $or(e_1, e_2, \ldots, e_n)$
The *or* operator outputs a complex event whenever a subevent occurs. t_1 and t_2 are set to start and end times of the subevent. Observe that *or* operator does not take a window argument.

Parametrized attribute-based constraints between events and value-based comparison constraints can be specified in the where clause as well. Spatial constraints may be specified in the where clause using *loc* attribute of events and spatial functions such as *distance(loc x, loc y)*. Moreover, spatial regions can be defined in the system and constraints can then be expressed using them. For instance, a region R can be expressed as a bounding box, and then the location of an event can be required to be in the region with *loc in R*.

Nonexistence (negation) constraints can be specified using the *not exists (subquery)* SQL construct. Subquery is specified as a *select-from-where* clause where *from* section is used to specify the subevent list and constraints are specified in the *where* clause. We illustrate the use of the constraints through the unattended bag event given in Figure 7.

```
complex unattended_bag
      on BagDetected B, node
 schema event_id as hash(unattended_bag, node.id, node.time, B.bagid),
          loc as B.loc,
          bagid as B.bagid
   where not exists ( select * from person_detected P
          where and(P,B;120) and distance(P.loc, B.loc) < 3 )
```

Figure 7. *Unattended_bag* complex event specifies a bag as unattended when no person is detected 3m around it for 120 seconds.

4. Complex Event Detection Framework

A naive approach to event detection would be to constantly send all the events to base station where they would be processed as soon as possible. However, this push-based centralized system would create a permanent hot spot location at the base station even at moderate incoming data rates. The described push-based data collection paradigm is common in continuous query processing systems [10, 11] where the global view of data is important. However, event detection systems only need the fraction of the data that generates events in the system. Therefore, continuous data collection can generally be avoided without missing event detections given that not all events cause complex events.

We construct event detection plans which specify efficient event detection strategies to avoid continuous data collection. The simplest event detection plan consists of a single step in which all subevents are simultaneously monitored (i.e. the naive plan). More complex plans have up to *n* (the number of subevents) steps in each of which a subset of the subevents are monitored. The number of detection plans for a complex event with n subevents (primitive) is exponential in n as given by the recursive relation $T(n) = \sum_{i=1}^{n} \binom{n}{i} T(n-i)$ where we define $T(0)$ to be 1.

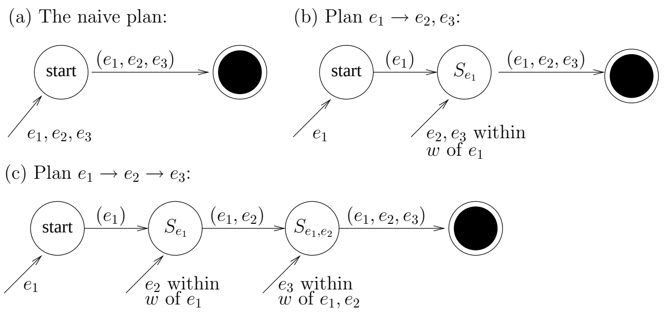
We design a cost model based on event occurrence probabilities to calculate the expected costs of event detection plans. We define the expected cost of a plan as the expected number of events it sends to base per time unit. For example, the cost of the naive plan for detecting the complex event *and*(e_1, e_2) would be the sum of unit costs of e_1 and e_2. On the other hand, a two step plan, first monitoring e_1 and looking up e_2 when e_1 occurs, could cost less but would incur higher latency. Hence, one of the main goals of our system is to try to find low network cost event detection plans meeting latency deadlines.

Latency of an event detection represents the time between the occurrence of the event and its detection by the system. Event detection latencies are based on network latencies. In our calculations, we do not consider the processing time or cost at the base station. However, since our system decreases the number of events sent to base, both the processing time and cost should be reduced as well.

4.1 Event Detection Plans

Event detection plans specify monitoring orders for the subevents of complex events. We represent the plans with finite state machines in our system. Consider the complex event *and*($e_1, e_2, e_3; w$) where e_1, e_2, e_3 are primitive events and *w* is the window size. State machines of the plans for this complex event have at most $n = 3$ states except the final state in each of which a subset of primitive events is monitored. One state machine of each size is given in Figure 8. For instance, the 3-step monitoring plan: "(1) continuously monitor e_1, (2) on e_1 lookup e_2, (3) on e_1 and e_2 lookup e_3", is illustrated in Figure 8c where the notation $e_1 \rightarrow e_2 \rightarrow e_3$ is used to denote this plan.

The finite state machines we use for representing plans are nondeterministic *(NFA)* since they can have multiple active states at a time. Every active state corresponds to a partial detection of the complex event. For example, in state S_{e_1} of the plan given in Figure 8c, there can be active instances of e_1 primitive events waiting for e_2 primitive events. Then when an instance of e_2 is detected, in addition to the transition to next state, a self transition will also occur so that an e_1 instance can match multiple instances of e_2 (self-transitions are not shown in the figure). Unlike the always active initial state, intermediate states are active only as long as the event window constraints allow.



(a) The naive plan:
start
(e_1, e_2, e_3)
e_1, e_2, e_3
(b) Plan $e_1 \rightarrow e_2, e_3$:
start
(e_1)
S_{e_1}
(e_1, e_2, e_3)
e_1
e_2, e_3 within w of e_1
(c) Plan $e_1 \rightarrow e_2 \rightarrow e_3$:
start
(e_1)
S_{e_1}
(e_1, e_2)
S_{e_1,e_2}
(e_1, e_2, e_3)
e_1
e_2 within w of e_1
e_3 within w of e_1, e_2

Figure 8. Event detection plans represented as finite state machines (FSMs)

In Section 4.1.1, we describe the plan generation process with the goal of optimizing the overall event detection cost. First, operator wise plan generation is explained where each operator forms a set of plans with different cost and latency characteristics as no single plan can be chosen that will guarantee global minimum cost in advance (this will be explained in more detail in the next section). Then, we describe how these plans are used in the global optimization of all event operators forming the event detection graph.

4.1.1 Plan Generation

In generating the plans for each operator, enumeration of the plan space is not a viable option since its size is exponential in the number of subevents as mentioned before. To address this issue we have come up with the following heuristics that together form a representative subset of all plans with distinct cost and latency characteristics:

Forward Stepwise Plan Generation: This heuristic starts with the minimum latency plan (the naive plan with the minimum latency plan selected for each complex subevent) and repeatedly alters it to form lower cost plans until latency constraint is exceeded or no more alterations are possible. At each iteration, the current plan is transformed into a lower cost plan either by moving a subevent detection to a later state or changing the plan of a subevent with a cheaper plan.

Backward Stepwise Plan Generation: This heuristic starts by finding the minimum cost plan (an n-step plan with the minimum cost plans selected for each complex subevent).

It can be found in a greedy way when all subevents are primitive, otherwise a nonexact greedy solution which orders the subevents in increasing $cost \times probability$ order can be used. At each iteration the plan is repeatedly transformed into lower latency plans either by moving an event to an earlier state or changing the plan of an event with a lower latency plan until no more alterations are possible.

Observe that the first heuristic starts with a single state FSM and extends it in successive iterations whereas the second one shrinks down the initial n-state FSM. Moreover, both heuristics choose the move with the highest cost-latency gain at each iteration and both end in a finite number of iterations since every move results in a better plan (lower cost for the first one and lower latency for the second one). While the first heuristic aims to form low latency plans with reasonable costs, the other one tries to form low cost plans meeting latency requirements.

All the plans are then merged into a feasible (i.e. meeting latency requirements) plan set. During the merge only *pareto optimal* plans are kept. Pareto optimal plans are the plans for which there exist no other plan we can use to either reduce the cost or latency without increasing the other. Moreover, only a limited number of pareto optimal plans can be stored by the operator node for use in the global optimization process (explained later in this section). In such a case, the choice is made so that plans with low latency, low cost and low latency-cost (a linear combination of the two factors) are equally represented.

We described the plan generation process for the cases where all the subevents of an operator node are primitive events. However, when complex subevents exist, the plan generation becomes a hierarchical process where the plans for the upper level nodes are built on the plans of the lower level nodes. Hence, plan generation is a bottom-up process in which the plans of lower level nodes are generated first.

As mentioned before, choosing only the minimum latency or cost plan at each node does not guarantee overall optimal solutions since (a) a lower cost but higher latency plan may be useful to reduce overall cost (e.g. when there are other events with higher latency plans such that the overall latency is not increased when a higher latency plan is used for this event) and (b) a lower latency but higher cost plan may reduce overall cost (because an event with a high cost plan may then switch to a lower cost plan with higher latency). For this reason, each node creates a set of plans with different latency and cost characteristics in the plan generation process. However, only a subset of these plans can be passed on to upper level nodes due to computational complexity. The size of this subset is a parameter trading computation with the explored plan space size. This process continues up to the root nodes, each of which then selects the minimum cost plan meeting its latency requirements. This in turn finalizes the generation of the event detection plans to be used by all nodes in a top-down manner. Finally, if a node has multiple parents requesting different plans, then it chooses the plan with the minimum latency.

The latency deadlines for complex events originate from two different sources. First, as mentioned before we may have user specified, explicit latency deadlines. Second, latency deadlines can also stem from limited data logging capabilities. More specifically, due to restricted storage some information sources may only be able to store the instances of an event type for a limited time. Therefore, any plan that relies on storage of events for longer periods are not gonna be useful. Our system considers both of the described latency requirements and uses the most strict one for each complex event.

4.1.2 Execution of Plans

Once the selection of plans is completed, the set of primitive events to monitor are identified and activated. When a primitive event arrives to the base station, it is directed to the corresponding primitive event node. The primitive event node stores the event and then forwards a pointer of the event to its active parents. An active parent is one which has expressed interest in the time interval the event arrived in. The complex event detection proceeds similarly in the higher level nodes. Each node acts according to its plan upon receiving events either by activating subevents or by detecting a complex event and passing it along to its parents.

4.1.3 Modeling Event Detection Plans

In this section, we explain the cost and latency characteristics of event detection plans. In our cost model, we use probabilistic models of event occurrences to derive expected costs of event detection plans. Our approach to cost modeling is not strictly tied to any particular probability distribution. Here, we derive the cost estimations for two different probability models: *Poisson* and *Bernoulli* distributions. In both cases we assume that events occur independently.

Poisson distributions are widely used in modeling discrete occurrences of events such as the receipt of a web request and arrival of a network packet. A Poisson distribution is characterized by a single parameter λ that expresses the average number of events occurring in a given time interval. In our case, we define λ to be the rate of occurrence for an event type, i.e. the average number of occurrences of an event type per time unit. In addition, we assume that the number of events in disjoint time intervals are independent. Under these conditions, the event occurrences follows a Poisson process with rate λ. On the other hand, when modeling an event type with the Bernoulli distribution, we assume that event occurs independently with probability p at every time step.

As described before, a complex event detection plan consists of a set of states each of which corresponds to monitoring a set of events. The cost of a plan is the sum of the costs of its states weighted by state reachability probabilities. Cost of a state depends on the cost of the subevents in that state. We define the latency of an event detection plan to be the maximum latency it could have so that we can guarantee latency deadlines. For this reason, we associate each event type with a latency value that represents the maximum latency its instances can have. Then, the latency of an event detection plan can be derived using the latencies of the subevents. Here, we consider identical latencies for all event types for simplicity. However, different latency values can be handled by the system as well. We will consider the expected cost and latency of monitoring the complex event *and(*e_1, e_2, e_3*)* for describing the process in more detail.

We define e_1, e_2 and e_3 to be primitive events with Δt latency and use Poisson processes with rates λ_{e_1}, λ_{e_2} and λ_{e_3} respectively to model the events. When the naive plan is used, all subevents will be monitored at all times. So the cost will be the sum of the expected occurrence rates of the subevents: $\sum_{i=1}^{3} \lambda_{e_i}$. The latency of the naive plan, which is simply the maximum latency among its subevents, is Δt.

The cost derivation for the three step plan $e_1 \rightarrow e_2 \rightarrow e_3$, given in Figure 8c, is more complex. We define the reachability probability of a state to be the probability of detecting the partial complex event that activates the state. For instance, the partial complex event which makes state S_{e_1} active is e_1. State reachability probabilities are derived using interarrival distributions of events. When using a Poisson process with parameter λ, the interarrival time is exponentially distributed with the same parameter. Hence, the probability of waiting time for the first occurrence of an event to be greater than *t* is given by $e^{-\lambda t}$. On the other hand, when using the Bernoulli distribution, the interarrival times have geometric distribution. The reachability probability for initial state is 1 since it is always active and the probability for final state is not required for cost estimation. Using the interarrival distributions to derive reachability probabilities the cost of the three step plan can be derived as:

$$\text{cost for } e_1 \rightarrow e_2 \rightarrow e_3 = \lambda_{e_1} + (1 - e^{-\lambda_{e_1}})2W\lambda_{e_2} + \\ ((1 - e^{-\lambda_{e_1}})(1 - e^{-W\lambda_{e_2}}) + (1 - e^{-\lambda_{e_2}})(1 - e^{-W\lambda_{e_1}}))2W\lambda_{e_3}$$

In the cost equation above and for the rest of the paper, we assume the probability of more than one events to occur in the same time step to be negligible. However, if that is not true, then the formula can be modified for the other case. Moreover, as more events are required to occur in a single time step, the occurrence probability will quickly diminish which means the terms with many concurrent events can be discarded as they will have negligible values.

The plan is assigned $3\Delta t$ latency since this is the maximum latency it exhibits (when the events occur in the order e_3, e_2, e_1 or e_2, e_3, e_1). Actually, for the exact latency we need to include the latency of sending pull requests for events e_2 and e_3 in the equation. However, the pull requests will have the same Δt latency and since we assumed all events to have the same latency it is not required to include them in our calculations (they will not change the result when comparing the cost of plans). For ease of presentation, we omit them in the rest of the paper as well.

And Operator. Here we describe the cost estimation for the n-ary *and* operator. Given the complex event *and(*$e_1, e_2, \ldots, e_n$*)* with window size *W*, and a detection plan with $m+1$ states S_1 through S_m and the final state S_{m+1}, we show the cost derivation using reachability probabilities both for Poisson and Bernoulli distributions below. For event e_j we represent the Poisson process parameter with λ_{e_j} and the Bernoulli parameter with p_{e_j}.

The cost for *and* operator with n operands is given by $\sum_{i=1}^{m} P_{S_i} * cost_{S_i}$ where P_{S_i} is the state reachability probability for state S_i and $cost_{S_i}$ represents the cost of monitoring subevents of state S_i for a period of length $2W$. In the case that all subevents are primitive $cost_{S_i} = \sum_{e_j \in S_i} 2W\lambda_{e_j}$ when Poisson processes are used and $cost_{S_i} = \sum_{e_j \in S_i} 2Wp_{e_j}$ for Bernoulli distributions.

P_{S_i}, the reachability probability for S_i, is equal to the occurrence probability of the partial complex event that causes the transition to state S_i. For this partial complex event to occur in this time step, all its constituent events need to occur within the last W time units with the last one occurring in this time step (otherwise the event would have occurred before). Then, P_{S_i} is 1 when *i* is 1 and for $m \geq i > 1$ is given for Poisson processes (i) and Bernoulli distributions (ii) by:

$$\text{(i)} \quad \sum_{e_j \in \bigcup_{k=1}^{i-1} S_k} (1 - e^{-\lambda_{e_j}}) \prod_{\substack{e_t \neq e_j \\ e_t \in \bigcup_{k=1}^{i-1} S_k}} (1 - e^{-\lambda_{e_t} W})$$

$$\text{(ii)} \quad \sum_{e_j \in \bigcup_{k=1}^{i-1} S_k} p_{e_j} \prod_{\substack{e_t \neq e_j \\ e_t \in \bigcup_{k=1}^{i-1} S_k}} (1 - (1 - p_{e_t})^W)$$

Under the identical latency assumption, the latency of a plan for *and* operator is defined by the number of the states in the plan (except the final state). Hence, the latency for the event *and(*$e_1, e_2, \ldots, e_n$*)* can range from Δt to $n\Delta t$.

Sequence Operator. We can consider the same set of plans for *sequence* operator as well. However, sequence has the additional constraint that events have to occur in a specific order and must not overlap. Therefore, the time interval to monitor a subevent depends on the occurrence times of other subevents and is at most W time units.

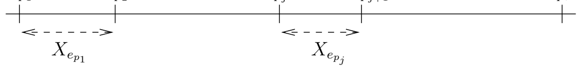


Figure 9. subevents for $seq(e_{p_1}, e_{p_2}, \ldots, e_{p_t})$

Expected cost for monitoring the complex event $seq(e_1, e_2, \ldots, e_n)$ with window size W using a plan with $m+1$ states has the same form $\sum_{i=1}^{m} P_{S_i} * cost_{S_i}$.

Let $seq(e_{p_1}, e_{p_2}, \ldots, e_{p_t})$ with $t \leq n$ and $p_1 < p_2 < \ldots < p_t$ be the partial complex event consisting of the events before state S_i, i.e. $\cup_{k=1}^{i-1} S_k = \{e_{p_1}, e_{p_2}, \ldots, e_{p_t}\}$. Then

1. P_{S_i}, the reachability probability for S_i, is equal to detecting $seq(e_{p_1}, e_{p_2}, \dots, e_{p_t})$ at a time point. For this complex event to occur subevents has to be detected in sequence as in Figure 9 within W time units. We define the random variable $X_{e_{p_j}}$ to be the time between $e_{p_{j+1}}$ and the occurrence of e_{p_j} before $e_{p_{j+1}}$ (see Figure 9). Then, $X_{e_{p_j}}$ is exponentially distributed with $\lambda_{e_{p_j}}$ if we are using Poisson processes, or has geometric distribution with $p_{e_{p_j}}$ when using Bernoulli distributions.

   (i) For the Poisson process case, we have $P_{S_i} = (1 - e^{-\lambda_{e_{p_t}}})(1 - R(W))$ where $R(W) = P(\sum_{j=1}^{t-1} X_{e_{p_j}} \geq W)$. Closed form expressions for sums of exponential random variables are studied in [9]. In the case all exponential variables have distinct parameters $R(W)$ has the following form:

   $$R(W) = \textstyle\sum_{j=1}^{t-1} A_j e^{-\lambda_{e_{p_j}} W} \text{ where } A_j = \prod_{\substack{k=1 \\ k \neq j}}^{t-1} \frac{\lambda_{e_{p_k}}}{\lambda_{e_{p_k}} - \lambda_{e_{p_j}}}.$$

   (ii) For the Bernoulli distribution $P_{S_i} = p_{e_{p_t}}(1 - R(W))$ where $R(W)$ is defined on a sum of geometric random variables. In this case, there is no parametric distribution for $R(W)$ unless the parameters of geometric random variables are identical. Hence, it has to be numerically calculated.

2. Any event e_{i_k} of state S_i should either occur (a) between e_{p_j} and $e_{p_{j+1}}$ for some j or (b) before e_{p_1} or after e_{p_t} depending on the order in $seq(e_1, e_2, \dots, e_n)$. In case *a*, we need to monitor e_{i_k} between e_{p_j} and $e_{p_{j+1}}$ for $X_{e_{p_j}}$ time units (see Figure 9). For case *b* we need to monitor the event for $W - \sum_{j=1}^{t-1} X_{e_{p_j}}$ time units. In the cost estimation, we use the expectation values $E[X_{e_{p_j}} | \sum_{k=1}^{t-1} X_{e_{p_k}} \leq W]$ and $W - E[\sum_{k=1}^{t-1} X_{e_{p_k}} | \sum_{k=1}^{t-1} X_{e_{p_k}} \leq W]$ for estimating $L_{e_{i_k}}$, the monitoring interval. Then $cost_{S_i}$ is $\sum_{e_{i_k} \in S_i} L_{e_{i_k}} \lambda_{e_{i_k}}$.

The latency of a plan for sequence depends on the latency of the last event (e_n) and the events in later states (after e_n) of the plan. If the complex event $seq(e_1, e_2, \dots, e_n)$ is being monitored with an m-step plan where the j^{th} step contains e_n, then its latency is $(m - j + 1)\Delta t$. This latency difference between *and* and *sequence* operators exists because unlike the *sequence*, with *and* operator any of the subevents can be the last event that causes the occurrence.

Negation Operator. In our system, negation can be used on the primitive events inside *and* and *seq* operators. Here, we consider the plans for complex events, with negated terms, specified using *and* operator over primitive events. The plans we consider for such events resemble a filtering approach. First, we detect the partial complex event consisting of non-negated events only. When that complex event is detected, we monitor the negated events. The detection plan for the complex event defined by non-negated events can be any arbitrary plan discussed for *and* operator. Same set of detection plans can be considered for negated events as well. However, the execution has to be changed in a way that the absence of an event is now what is aimed for. The cost estimations discussed for *and* operator can be applied here by changing the occurrence probabilities with nonoccurrence probabilities.

Or Operator. As discussed before, *or* operator generates a complex event for every event instance it receives. Hence, the only detection plan for *or* operator is the *naive* plan. The cost of the naive plan is the sum of the costs of the subevents and its latency is the highest latency among the subevents.

4.2 Optimizing for Shared Subevents

The hierarchical nature of complex event specification may introduce common subevents across complex events. For example, in a network monitoring application we could have the *syn* event indicating the arrival of a TCP *syn* packet. Various complex events could then be specified using the *syn* event such as syn-flood (sending syn packets without matching acks to create half-open connections for overwhelming the receiver), a successfull TCP session and another event detecting port scans where the attacker looks for open ports.

The overall goal of shared optimization is to find the set of plans for which the total cost of monitoring all complex events is minimized. Yet the base algorithm presented in Section 4.1.1 does not consider sharing between event expressions as it runs independently for each expression. Here, we modify our plan generation algorithm for (1) calculating the overall event detection cost correctly when shared subevents exist and (2) choosing plans that facilitate sharing to further reduce cost when available and applicable.

First, we need to identify the expected amount of sharing that will happen on a shared node. However, the degree of sharing depends on the plans selected by the ancestor nodes of the shared node. Since our base algorithm proceeds in a bottom-up fashion, we cannot identify the amount of sharing unless the algorithm completes and the plans for all nodes are selected. Below, we present an iterative version of our algorithm to address these problems (for simplicity, modified algorithm is presented for the case of a single shared complex event):

1. run the base plan generation algorithm
2. find the expected amount of sharing with the current plan selections and recalculate the current plan costs for ancestors of the shared node
3. rerun the base algorithm starting at each parent of shared node utilizing the sharing probabilities
4. if overall cost is reduced then goto 2 else exit with the previous plans

After the first step, every node will have selected its plan but the total cost for the shared node will be incorrect. In the second step, we fix the overall cost by taking sharing into account. This is possible because we can find the amount of sharing after all nodes have selected their plans. We assume that parents of the shared node function independently and find the probability that they will monitor the shared event in overlapping intervals. Third step runs the base algorithm starting at each shared node parent. However, in this execution of the algorithm, the sharing probabilities of generated plans are calculated and utilized as well. Hence, ancestors of shared node may now change their plans since increased sharing may further reduce plan costs. Moreover, the plan changes made in third step are guaranteed to increase the amount of sharing because (1) Cost of the shared node can only decrease due to sharing and (2) Ancestor nodes can only reduce their costs at each step if they choose plans which monitor the shared node in earlier states (and monitoring the shared node earlier means it will be shared more). The algorithm reiterates as long as the overall cost is reduced.

4.3 Constraint Optimization

In this section, we describe how the spatial and attribute-based constraints affect the occurrence probabilities of events and explain the additional optimizations we have made to the plan selection and execution processes for further reducing cost using these constraints. First, we discuss the effects of spatial constraints on the plan generation process. The **spatial constraints** we consider are defined in terms of regional units. The space is divided into regions such that events in a region occur independently from events in other regions. The division of space into such independent regions is typical for some applications. For instance, in a security application we could consider the rooms of a building as independent regions. In addition, it is also easy for users to specify spatial constraints (by combining smaller regions) once regional units are provided.

When the spatial constraints are specified in the described way, their effect on event occurrence probabilities can be incorporated in our system with minor changes. First, we modify our model to keep event occurrence statistics per each independent region of an event type. Then, when a spatial constraint on a complex event is given, we only need to combine the information from corresponding regions to derive the associated event occurrence probability. For example, if we have Poisson processes with parameters λ_1 and λ_2 for two regions, then the Poisson process associated with the combined region has the parameter $\lambda_1 + \lambda_2$. Hence, by combining Poisson processes we can easily construct the Poisson process for any arbitrary combination of independent regions. However, this is only possible because the regions are independent, otherwise we would have to derive joint distributions. Hence, the spatial constraints alter the event detection process, such that different plans may be used for monitoring different spatial regions if doing so reduces the overall cost. A related experiment is available in the experiments section.

Attribute-based constraints have been considered in many query processing systems. The main approach, which we also adapt, has been to keep histograms for attributes. Histograms provide the information for deriving the selectivity probabilities of attribute-based constraints which we can then use to derive the event occurrence probabilities. Moreover, value based attribute constraints can be pushed down to information sources further reducing the number of transmitted events. Parametrized attribute constraints between events can also be pushed down whenever one of the events is monitored earlier than the other one.

5. Experiments

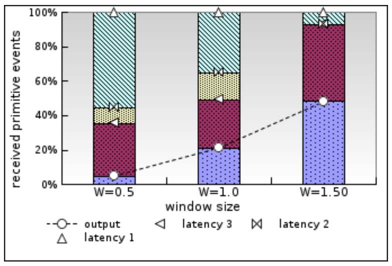
In this section, we analyze the performance of our system and investigate the effects of various parameters through a set of experiments. We have implemented the base node functionality and generated specific event adapters for use in experiments. Our experiments involve both synthetic and real data sets. Unless stated otherwise *Zipfian* distribution has been used in synthetic data generation. Real data set is a collection of network traffic logs obtained from Planetflow web site [12].

5.1 Experiments with Synthetic Data Sets

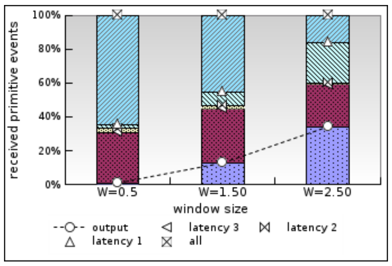
On window size and detection latency: We explore the effects of window size and latency deadline on the event detection cost in this experiment. We defined the complex events *and(*e_1, e_2, e_3*)* and *seq(*e_1, e_2, e_3*)* where e_1, e_2 and e_3 are primitive events. Using Zipfian distribution with skew 0.255 (other skew values are used in the varying skewness experiment) we generated event streams for these primitive types. Then, for different window values and latency deadlines of both complex events we ran our event detection algorithm on the generated streams. The event detection costs, expressed as percentage of the primitive events sent to base, are provided in Figures 10(a) and 10(b).

Both figures show that as the allowed latency for event detection increases (from 1 to 3 in this case) the event detection cost reduces. The lines labeled *output*, which show the percentage of primitive events output as parts of complex events, serve as a lower bound on cost. Because *sequence* operator does not need to monitor all events unless the first events of sequence occur, it can reduce cost even under hard latency constraints. Finally, the event detection cost increases with window size since larger window size means increased event occurrence probability.

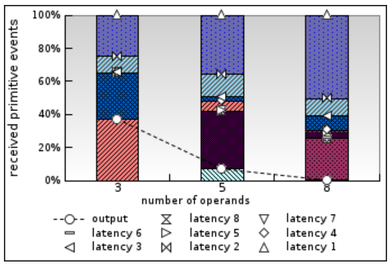
Increasing the number of subevents: In this experiment, we investigate the cost performance under increasing number of subevents through a complex event specified with a single *and* operator. We randomly generated streams using similar event frequencies for all event types (to rule out the effect of frequency in the test). In Figure 10(c), we can see that (1) increasing the number of operands tends to decrease the number of detected complex events and (2) greater num-



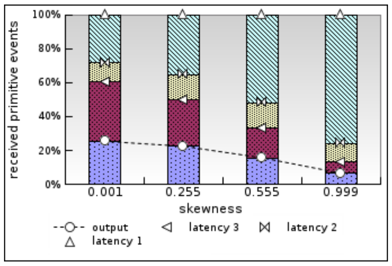
(a) and operator with 3 operands



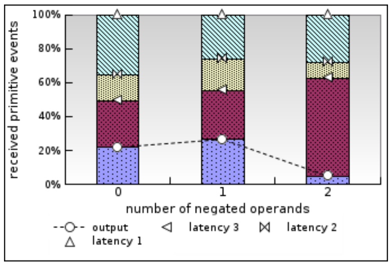
(b) seq operator with 3 operands



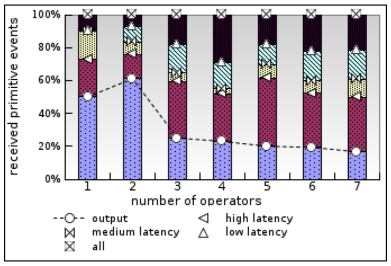
(c) and operator with increasing operands



(d) and operator with increasing skew



(e) and operator with increasing negation



(f) events with increasing number of operators

Figure 10. Operator wise experiments and complex event detection performance

ber of operands means we have a wider latency spectrum (therefore a larger plan space) to reduce cost.

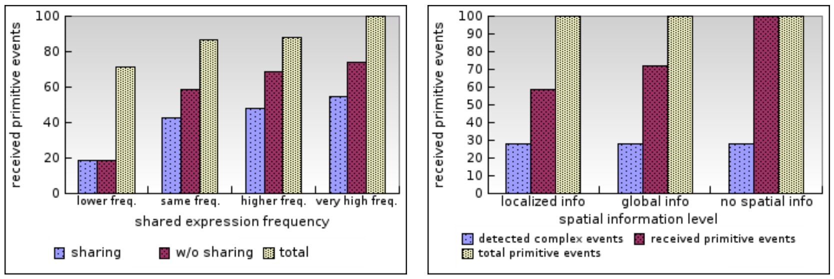
Workloads with varying skewness: In this experiment, we use the complex event *and*(e_1, e_2, e_3) with a fixed window parameter under workloads with varying skewness. Each workload stream is generated with a Zipfian distribution and has around the same number of events. In Figure 10(d), we see that in low skew streams a greater number of complex events is detected and the cost is therefore higher. Increasing the skew generates event types with low frequencies which our system uses to reduce the cost.

Negated subevents: To explore the cost performance for complex events involving negated subevents, we performed an experiment using the *and*(e_1, e_2, e_3) event in which we varied the number of negated subevents. In Figure 10(e), we can see that while the costs for the complex events with single and no negated terms are similar, the cost when two subevents are negated is high even though less complex events are detected. This is mainly because (1) monitoring of negated and non-negated events are not interleaved, that is we monitor the negated subevents after the non-negated subevents occur (see Section 4.1.3) and (2) all the detected non-negated subevents are discarded when a negated subevent that prevents them from forming a complex event is detected.

Increasing the number of operators: In this experiment, we consider the cost performance with increasing number of operators. We varied the number of operators used in complex events from 1 to 7 and for each operator count we generated 10 complex events based on event composition rules. The average event detection cost for each operator count is shown in Figure 10(f). As the number of operators in an expression is increased, generally its occurrence probability decreases. Moreover, for similar event occurrence probabilities the relative cost of event detection is also similar irrespective of the operator number.

Shared subevents: To test the shared event optimization, we specified two complex events with a common subevent tree and compared the performance with and without shared optimization. In the experiment, we varied the frequency of the complex event that corresponds to the shared subtree. In Figure 11(a), we see that when the frequency of the shared part is low, both with and without sharing the system experiences similar cost since the shared part is chosen to be monitored earlier in both cases. When the frequency of the shared part is the same with or slightly higher than other parts, non-shared parts are monitored earlier without sharing optimization. In this case, shared optimization reduces cost by monitoring the shared part first. Finally, when shared part has very high frequency, non-shared parts are monitored first in both cases. Even in this case shared optimization experiences less cost, because it better estimates shared subevent costs which can cause better execution plans to be selected in some cases (since we are using heuristics for plan generation). When we used exhaustive plan generation, both with and without sharing the algorithm chose the exact same plans for this case.

Spatial constraints: In this experiment, we show the utilization of spatial constraints in reducing detection costs through the complex event *and*(e_1, e_2) with constraint *e1.loc = e2.loc*. We assume that there are two regions X and Y with event occurrence rates $\lambda^X_{e_1} = 3\lambda$, $\lambda^Y_{e_1} = 7\lambda$, $\lambda^X_{e_2} = 6\lambda$ and $\lambda^Y_{e_2} = 4\lambda$. When localized information is available, i.e.



(a) shared optimization (b) spatial constraint between two events

Figure 11. Shared optimization and spatial constraints

frequencies of events are known for each region, the cost is lowest (see Figure 11(b)). In this case, the system monitors e_1 in region X and e_2 in region Y. Then when e_1 is detected in X (or e_2 in Y), e_2 is monitored in X (or e_1 in Y). When localized information is not available, but the global selectivity of the spatial constraint is known (global info in Figure 11(b)), either e_1 or e_2 is monitored (both have the same total frequency) in all regions. Finally, when no spatial constraint information is available, the system expects that the complex event will occur every time step and therefore chooses to execute the naive plan.

5.2 Experiments with Planetlab Data Set

The Planetlab data set we have used consists of 5 hours of network logs for 49 Planetlab nodes we have obtained from [12]. The network logs provide aggregated information on network connections between Planetlab nodes and other nodes in the Internet. The provided information includes connection start/end times, amount of generated traffic and used network protocol. We have experimented with various complex events most of which can easily be found on many network monitoring applications. Here, we present the results for three of the complex events.

Change of overall network load: We define a Planetlab node as *idle* if its average network transfer speed (incoming and outgoing total) in the last minute is less than 125KBps and as *active* if the average speed is greater than a threshold T. Given that, the complex event monitors for an overall network load change from a situation where more than half of all nodes are idle to more than half being active within a specified time interval. The complex event is defined as *seq(count(idle)* $>$ *%50 of all nodes, count(active)* $>$ *%50 of all nodes; W=30min)*. The results are provided in Figure 12(a) for $T = 250, 500$, and 1250 KBps.

Diverse clusters: We define a cluster to be a set of machines from the same $/8$ IP class. A diverse cluster is then defined as a cluster with more than C connections to Planetlab nodes in total. To specify this complex event we first define a *locally diverse cluster* event which monitors the event that a Planetlab node has more than $\frac{C}{N=49}$ connections with a cluster. The diverse cluster complex event is specified as *sum(conns)*$>$ C *group by cluster*. Then, it is *and*'ed with the locally diverse cluster event which acts as a prerequisite for the diverse cluster event and helps reduce monitoring cost. The results are given in Figure 12(b) for $C = 250, 500, 1000$, and 2000.

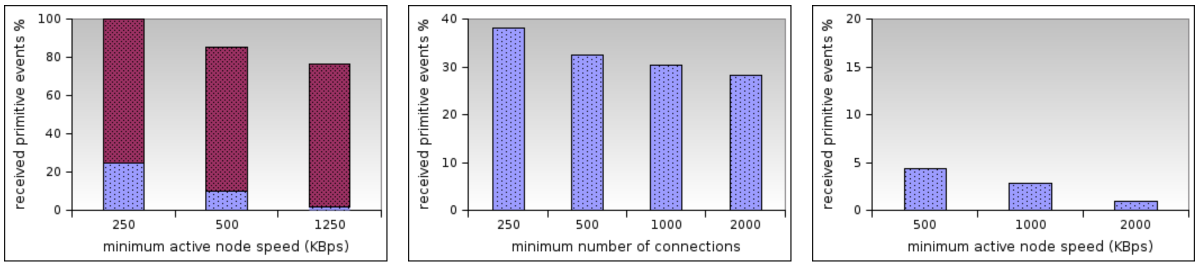
Clusters with multiple active nodes:

We define a node (outside of Planetlab) to be *active* if its aggregate average network transfer rate to Planetlab nodes is more than T in the last minute. In this complex event we are interested in $/8$ clusters with more than one active nodes in the last minute. Similar to the diverse cluster complex event, we first defined a *locally active node* event which monitors a node with an average network speed greater than $\frac{T}{N=49}$ to a Planetlab node. Then the active node complex event is specified as *sum(speed)* $>$ *T group by node_ip* and is *and*'ed with the locally active node event which acts as a prerequisite event. Finally, clusters with multiple active nodes is specified as *count(active node)* $>$ *1 group by cluster*. The results are provided in Figure 12(c) for $T = 500, 1000$, and 2000 KBps.

6. Related Work

In continuous query processing systems such as TinyDB [1] for wireless sensor networks, and Borealis [10] for stream processing applications queries are expected to constantly produce results. Push based data transfer, either to a fixed node or to an arbitrary location in a decentralized structure, is characteristic of such continuous query processing systems. On the other hand, event detection systems are expected to be silent as long as no events of interest occur. The aim in event systems is not continuous monitoring of the data, but is the detection of events of interest.

In the active database community, ECA (event-condition-action) rules have been studied for building triggers [6]. Triggers offer the event detection functionality through which database applications can subscribe to in-database events, e.g. the insertion of a tuple. However, most in-database events are simple whereas more complex events could be defined in the environments we consider. Many active database systems such as Samos [2], Ode Active Database [3], and Sentinel [4] have been produced as the results of the studies in the active database area. Most systems



(a) network load change (b) diverse clusters (c) clusters with multiple active nodes

Figure 12. Experiments with Planetlab data set

provide their own event languages. These languages form the base of the event language in our system. However, our language has additional features such as spatial and temporal constructs which are important for the systems we consider.

In the join ordering problem, database query optimizers try to find ordering of relations for which intermediate result sizes is minimized [13]. Most query optimizers only consider the orders corresponding to left-deep binary trees mainly for two reasons: (1) Available join algorithms such as nested-loop joins tend to work well with left-deep trees, and (2) Number of possible left-deep trees is large but not as large as number of all trees. Our problem of constructing minimum cost monitoring plans is different from the join ordering problem for the following reasons. First, we are not limited to binary trees since multiple event types can be monitored in parallel. Second, our cost metric is the expected number of events sent to base. Finally, we have an additional constraint, i.e. the latency constraint, further limiting the solution space.

In a recent study about high performance complex event processing [5] optimization methods for efficient event processing are described. There the aim is to reduce processing cost at the base station. While our system also helps reduce the processing cost, our main goal is to minimize the network traffic. Moreover, their system does not consider distributed event processing, and simultaneous queries.

7. Conclusions and Future Work

CED is a critical capability for many monitoring applications. While earlier work primarily focused on optimizing processing requirements of complex events, we made an effort towards optimizing communication needs when distributed sources are involved.

The results support our premise that communication requirements can be significantly reduced by exploiting spatiotemporal constraints within the event specification and the frequency skew among the relevant sub-events, at the expense of additional detection delays. Specifically, the main benefit came from a novel multi-step planning technique that combined proactive and retroactive monitoring of events.

This is a rich research area with many open problems. Our immediate future work will explore probabilistic planning for sensor network applications and augmenting manual event specifications with learning-based techniques.

References

[1] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: An acquisitional query processing system for sensor networks. Transactions on Database Systems (TODS), 2005.

[2] S. Gatziu and K. R. Dittrich. Detecting composite events in active database systems using petri nets. In Proc. 4. Intl. Workshop on Research Issues in Data Engineering, pages 2–9, Houston, USA, 1994.

[3] S. Chakravarthy. et al. Composite Events for Active Databases: Semantics, Contexts and Detection, VLDB 1994.

[4] S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. Data and Knowledge Engineering, 14(10):1–26, 1994.

[5] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-Performance Complex Event Processing over Streams. SIGMOD 2006

[6] N. Paton and O. Diaz, ’Active Database Systems’, ACM Comp. Surveys, Vol. 31, No. 1, 1999.

[7] Zimmer, D. and Unland, R. On the Semantics of Complex Events in Active Database Management Systems. p.392, ICDE’99.

[8] The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems, David Luckham, May 2002.

[9] S. V. Amaria and R. B. Misra, Closed-form expressions for distribution of sum of exponential random variables, IEEE Trans. Reliability, vol. 46, no. 4, pp. 519-522, Dec. 1997.

[10] Daniel Abadi, et al. The Design of the Borealis Stream Processing Engine. CIDR’05.

[11] S. Chandrasekaran, et al. TelegraphCQ: Continuous Dataflow Processing. In ACM SIGMOD Conference, June 2003.

[12] http://planetflow.planet-lab.org/

[13] Selinger, P. G., et al. 1979. Access path selection in a relational database management system. SIGMOD ’79.