Event-based Constraints for Sensornet Programming

Jie Mao
Brown University
jmao@cs.brown.edu

John Jannotti
Brown University
jj@cs.brown.edu

Mert Akdere
Brown University
makdere@cs.brown.edu

Ugur Cetintemel
Brown University
ugur@cs.brown.edu

ABSTRACT

We propose a sensornet programming model based on declarative spatio-temporal constraints on events only, *not* sensors. Where previous approaches conflate events and sensors because they are often colocated, a focus on events allows programmers to specify their intent more directly, and better supports remote sensing devices such as cameras, microphones, and rangefinders. In our model, complex events are specified as aggregations of events in time or space, without regard to sensor locations or communication paths. New techniques are required to aggregate events based on these constraints without knowledge of nearby nodes.

We present a decentralized, scalable event detection framework that allows for efficient in-network aggregation without coupling events and sensors. First, we describe a SQL-style declarative language with spatio-temporal constraints between events that can be used to express complex events. Next, we show how these complex events can be assembled efficiently. The distributed event detection mechanism scales to very large networks, load balances work across sensors, and is fault tolerant to network partitions and node failure.

Categories and Subject Descriptors

D.3.3 [**Software Engineering**]: Language Classifications—*Constraint and logic languages*; C.2.4 [**Computer-communication Networks**]: Distributed Systems—*Distributed applications*

General Terms

Design Languages Performance

Keywords

Programming Model, Sensor Networks, Complex Event, Constraints

1. INTRODUCTION

Many wireless sensor network applications require the fusion of sensor readings from individual sensors into meaningful *events*. These events draw the attention of human operators, activate actuators, or contibute to the construction of even higher-level events.

The events of interest vary greatly based on different application requirements. Sometimes, a single sensor reading is significant to the application. In other cases, applications are concerned with *complex events* which are the aggregations of geographically and temporally related sensor data. In these applications, sensor data from several different sensor nodes sensed at different moments and places must be fused to create application-specified events.

To aggregate geographically and temporally distributed sensor data, sensor nodes could send all reading to a single rendezvous where they could be aggregated into application-specific events. There are several obvious drawbacks with this approach. First, sending all sensor data to a single place requires complete connectivity, and creates congestion near the base station. Second, sensor data are often redundant for complex event detection. Sending all data indiscriminately wastes bandwidth and power and thus shortens the system lifetime. Existing data aggregation algorithms generally address these problems by aggregating sensor readings into complex events at join points while propagating toward a collection point. Unfortunately, this form of aggregation is greatly complicated by remote sensing. It is is difficult to determine that a subevent need not be propagated further up the tree when any given subtree might report an event from a distant location. Cameras may observe events quite far from their location, for example.

Beyond tree aggregation, some systems aggregate among sensor *neighborhoods* [7, 14]. These neighborhoods are based on sensor locations (*i.e.* nodes with 10m of a given sensor), or communication details (*i.e.* nodes within two hops of a given sensor). These definitions are sometimes called *data-centric* because they abstract away the details of node identity, and focus on the location of sensor readings.

However, truly data-centric applications will not specify their operation in terms of sensor locations and communication paths. Notions of locality, in space or time, are best tied to events, not sensors or pathways. This separation is critical to supporting long-range sensors (*i.e.* cameras), or complex events that may be deduced to occur at a locations far from any single sensor (*i.e.* triangulated sounds).

For example, consider three successively more complex applications in a network of acoustic and visual sensors deployed in an urban area.

1. Detect gunfire. The application is interested in acoustic data matching the gunshot sound pattern. A single match indicates that gunfire is present. No data aggregation is neces-

sary.

2. Locate gunfire. The arrival time of the sound of a gunshot at multiple sensor nodes must be compared in order to triangulate and locate the point of fire[12]. Triangulation requires the exchange of acoustic data and processing among several sensor nodes.

3. Locate suspects near gunfire. The location and time of gunfire must be compared to the locations of people detected by surveillance cameras. The locations of the gunfire may be arbitrarily far from the acoustic and visual sensors. Further, the microphones that detect the gunfire may not be colocated with the cameras that observe people. Coordination must occur with constraints expressed on the times and locations of events, not of sensors.

The third application motivates our work to allow complex event detection based on event constraints rather than sensor-based neighborhoods. In this paper, we present a complex event detection framework that uses geographic addressing to decouple event locations from sensor locations while allowing maximum flexibility in choosing aggregation nodes.

Our mechanism, *Distributed Constraint Processing* (DCP), performs distributed hierarchical event processing. We have extended geographic hash tables (GHTs) [11] in several ways to help implement DCP. DCP allows the specification of events at increasing levels of complexity. The constraints used to specify event computation are used to perform optimizations that avoid global collection of low-level events. The building blocks of DCP are *event processors*, which take in simpler, lower-level events and compute more complex, higher-level events. DCP places event processors within the network at the locations where their input events will be routed. The coordinates of these locations are computed by *Regional GHTs*. These event processors operate on input data when they become available, and send their output events to higher-level event processors for further computation. DCP leverages the properties of GHTs to provide load-balancing and fault-tolerance. Additionally, fault-tolerance is enhanced by our Regional GHTs which store geographic events locally, allowing continued operation during network partitions.

DCP is particularly suited to sensor network applications in which the subevents of a complex event happen close to each other in space or time (even if the sensors that detect the events are not nearby). DCP leverages this proximity to facilitate efficient local aggregation. However, DCP degrades gracefully to act as a general collection tree for applications that require global fusion, such as the detections of *max*, *min*, and *average* sensor readings.

In the rest of the paper, we describe our constraint framework and discuss its performance. In Section 2, we use our event specification language to demonstrate complex event detection with an example application. We show how complex events are built up from primitive events and describe the implementation of DCP based on the example application in Section 3. Section 4 evaluates DCP's performance compared to simpler approaches. The paper closes after a discussion of related work.

2. COMPLEX EVENT DETECTION

In this section, we describe the concept of *complex events* and present an event specification language. We describe how com-

plex events are formed in a hierarchical way from simpler events and demonstrate our language using an application scenario. There have been proposals for event languages both in the active database community and in the complex event processing community. Our main contribution does not lie in the language itself. We use it only to demonstrate the importance of constraints to the specification of complex events. Throughout our example application, note that events are naturally specified by declaring the constraints that must be met in order to produce complex events from simpler events.

2.1 Complex and Primitive Events

Events are defined as occurrences of interest in a system. A person in a room, high temperature in an area, or the theft of a book might all be events in different applications. However, unlike a single sensor observation, a "book theft" event requires many observations and computation over those events. Accordingly, we divide events into two categories: primitive events and complex events.

Raw sensor readings are primitive events. A primitive event consists of the reading itself annotated with metadata, such as a time and location.

Complex Events are derived from simpler events. They are produced by *Event Processors* or *Event Detectors*, rather than individual sensors. An event processor creates complex events when it observes the appropriate constituent events. For example, a fire detector may require the observation of high temperature and smoke sensor readings. Further, the detector computes a fire event only if high temperature and smoke events occur in close proximity, in time and space.

In DCP, there are multiple event processors for every event type. Event processors of the same type are distributed throughout the network using Regional GHTs to facilitate distributed event aggregations. The efficient evaluation of constraints to produce complex events will be discussed in detail in Section 3.

2.2 Event Specification Language

Our event specification language borrows event operators from active database research where event operators were used in specifying triggers in database systems. We have also incorporated windowing constructs from stream processing and complex event processing research.

2.2.1 Sensor Specification

All events are derived in some way from raw sensor readings. The output of each sensor is declared in order to reference their raw readings in derived events. The sensor specification conforms to the following template:

sensor   *name*
schema   *attr_list*

   attr_list → *attr* | *attr_list, attr*
      attr → *attr_type name*
attr_type → *double* | *int* | *string*

Figure 1: A sensor is given a name, and any number of named and typed attributes. These attributes are referenced to create *events*.

Here are two example sensor specifications:

 sensor *temperature* **schema** *double temp*

sensor barometer **schema** *double pressure*

We also assume that a pseudo-sensor named *node* exists in all sensor platforms. *Node* provides the spatial and temporal context information used during the construction of both complex and primitive event. It has the following specification:

sensor   *node*
schema   *string node_id,*
         double[2] loc,
         double time

node_id is the unique id of the device the sensor is located on. *loc* specifies the platform's location and the *time* attribute is used to access the platform clock. We assume that sensors are sufficiently synchronized, in time and space, to use these values in calculating constraint matches.

2.2.2   Base Event Schema

Complex and primitive events are both represented as attribute collections. All event of the same type have the same set of attributes which is called the event schema. The schema for each event type is specified in the event type declaration.

Some attributes, such as timestamp and location, are required in both complex and primitive events. They constitute the *base event schema*. The base schema includes the attributes *event_id*, *loc*, *start_time*, *end_time* and *node_id*. The base event schema facilitates the use of standardized event operators that evaluate common spatio-temporal relationships.

event_id is the identifier that identifies an instance of an event type. This identifier can be made unique by generating a fresh identifier for each complex event instantiation, or it can be created based on a subset of the attributes of an event instance. In the latter case, logically duplicate event instances will have the same identifier and may be suppressed during later processing. The *loc* attribute stores the location assigned to the event instance. *start_time* and *end_time* represent the occurrence interval of the event. Finally, *node_id* identifies the node that generates the event instance.

2.2.3   Primitive Event Declaration

Primitive event declarations specify the transformation of sensor readings into primitive events. A primitive event can be regarded as a sensor reading annotated with metadata information. Primitive event declarations are made using the template in Figure 2.

primitive name
     on *sensor_list*
  schema *base_schema, attribute_list*

Figure 2: A primitive event is created by combining attributes from one or more sensors. The pseudosensor "node" is often used to provide the time and location required by the base event schema.

The *name* symbol stands for the name assigned to the primitive event type such as *person_detected*, or *barometer_reading*. *Sensor_list* contains the sensors the primitive event is defined upon. It may contain multiple sensors, but they must be located on the same node. Sensor fusion across nodes is described by complex events. Finally *schema* specifies the attributes of this primitive event type and the way they are assigned values. Here we provide an example primitive event specification for a temperature reading event common in many sensor network scenarios.

primitive temp
     on *temperature, node*
  schema *event_id as hash(node.node_id,*
                node.node_time),
        loc as node.loc,
        start_time as node.time,
        end_time as node.time,
        temp as temperature.temp

Figure 3: The *temp* primitive event consists of the temperature reading from the sensor named *temperature* along with time and location information to populate the base event scheme from the *node* pseudosensor.

2.2.4   Complex Event Declaration

Complex events are combinations of simpler events, each of which may be primitive or complex. For most applications, users are interested in specifying complex events which impose spatial, temporal or attribute-based constraints on their subevents. We take a SQL-like approach to complex event specification and extend it with spatial/temporal constructs such as time windows to support these constraints. Our complex event specification template is given in Figure 4.

complex name
     on *source_list*
  schema *base_schema, attribute_list*
    where *constraint_list*

Figure 4: complex event declaration template

Every complex event type is assigned a unique name with the *name* attribute. The *source_list* is used to specify the subevents of a complex event type. The source list may also contain the *node* pseudosensor. As in primitive event specifications, *schema* specifies the attributes of the complex event type and also defines the transformation from subevents and their attributes into the attributes of the complex event. The *constraint_list* in the *where* clause specifies a logical expression on the subevents that must be fulfilled to construct the complex event. Constraints can be defined over subevent attributes, and can specify temporal or spatial patterns over subevents. We provide event operators for easy specification of constraints over subevents. Existential constraints are also available through subqueries. These features are described in Section 2.2.5.

As a simple example of a complex event consider the high temperature event. We define the high temperature complex event using the previously defined *temp* primitive event as follows:

complex hitemp
     on *temp T, node*
  schema *event_id as hash(node.node_id,*
                node.node_time),
        loc as T.loc,
        start_time as T.start_time,
        end_time as T.end_.time,
        temp as T.temp
    where *T.temp > 70*

Figure 5: A hitemp event is constructed from a single subevent when a temp event, T, meets the constraint: T.temp > 70.

2.2.5   Constraint Specification

Temporal, spatial, attribute-based and existential constraints can be specified in the where clause of a complex event specification.

Each constraint returns a boolean result. For easy specification of event constraints we provide event operators, as introduced in the event languages developed in active database research. We have borrowed the event operators *and*, *or*, and *sequence* from existing work in that area [2, 3, 9]. All event operators are n-ary operators. The last argument of each event operator is the time window argument, w, which specifies the maximum time between any two subevents of the complex event. Subevents which are separated by more than w time units cannot be part of the same complex event instance. When an event operator produces output on a given set of subevents we say the corresponding event constraint is satisfied.

We also provide the SQL construct *exists (subquery)* for the specification of existential constraints. The result of the *exists* clause is true if the subquery returns any result. An example event specification to detect unattended luggage is given in Figure 6.

complex unattended_bag
     on BagDetector B, node
 schema event_id as hash(node.node_id,
             node.node_time, B.bagid),
      loc as B.loc,
      start_time as B.start_time,
      end_time as B.end_.time,
      bagid as B.bagid
  where not exists (select * from person_detected P
   where and(P,B;60) and distance(P.loc, B.loc) < 5)

Figure 6: An *unattended_bag* event when a bag is detected, but no person is detected within 5 meters for one minute. The base event schema is populated from the base schema in the *BagDetector* event.

The unattended bag complex event is an example for a security monitoring scenario. We assume that there is a detector for bags already implemented and we can access a bag detection event through BagDetector. The event specification is made such that a bag is considered unattended when no person is detected within 5 meters of the bag for 60 seconds.

2.3 Example Application

Consider an example object tracking application using calibrated stereo cameras. Stereo cameras can localize the 3D positions of the objects in their frustums, and can identify different objects using techniques, such as histogram comparison [1]. Such a camera network can be used to monitor behavior of people and to detect abnormal activities in an area. Here, we present an example scenario where the monitored event is a *person chasing another person*. We use our event specification language to declare the events involved in the application.

In order to detect complex events, we break them down into simpler, lower-level events and repeat this process until all events are primitive. For our example chase scenario, this process is illustrated in Figure 7. We can think of the *people_chasing* complex event as two people running close to each other (*e.g.* 10 meters) for a certain amount of time (*e.g.* 5 seconds). Below is the specification for the *people_chasing* complex event based on this idea.
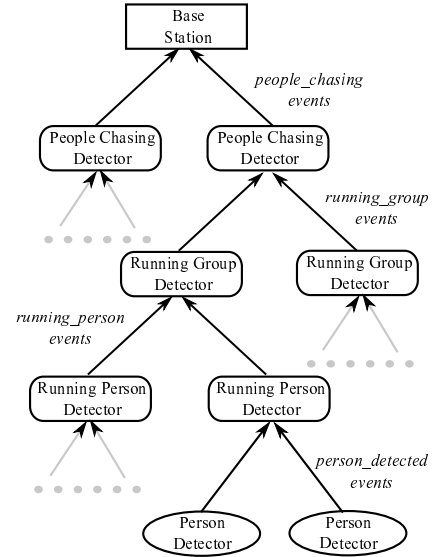
Figure 7: The process to detect the *people_chasing* event is decomposed into the following steps: 1. Detect a person using the person detector on the stereo camera sensor; output a *person_detected* event; 2. Detect a running person by calculating the person's moving speed using two consecutive *person_detected* events of the same person; output a *running_person* event; 3. Detect people running together by calculating the distance between two different running people; 4. Detect people chasing each other by looking for two people keeping running closely for a period of time, examining continuous *running_group* events with same person IDs; send *people_chasing* events back to the base station.

complex people_chasing
     on running_group as G1, running_group as G2,
        node
  schema event_id as hash(node.node_id, node.time,
        G1.person1_id, G1.person2_id),
      loc as avg(G1.loc, G2.loc),
      start_time as G1.start_time,
      end_time as G2.end_time,
      node_id as node.node_id,
      person1_id as G1.person1_id,
      person2_id as G1.person2_id,
  where seq(G1, G2; SRC_PERIOD_RG) and
      G1.person2_id = G2.person2_id and
      G1.person1_id = G2.person1_id and
      distance(G1.loc, G2.loc) <= CHASING_DIST

people_chasing complex event is defined using the *running_group* complex event. *running_group* complex event detects two people running in close proximity. Its specification is given below.

```
complex running_group
      on running_person as R1, running_person as R2,
            node
 schema event_id as hash(node.node_id, node.time,
            R1.person_id, R2.person_id),
            loc as avg(R1.loc, R2.loc),
            start_time as min(R1.start_time, R2.start_time)
            end_time as max(R1.end_time, R2.end_time),
            node_id as node.node_id,
            person1_id as R1.person_id,
            person2_id as R2.person_id,
   where and(R1, R2; SRC_PERIOD_PR) and
            R1.person_id != R2.person_id and
            distance(R1.loc, R2.loc) <= GROUP_DIST
```

The *running_group* complex event depends on the complex event *running_person*. The *running_person* complex event, which is used to find a running person, can be detected by computing a person's moving speed and comparing it to a threshold speed. This involves the comparison of two *person_detected* events of a person with different location and timestamps. The specification of the *running_person* complex event is given below.

```
complex running_person
      on person_detected as P1, person_detected as P2,
            node
 schema event_id as hash(node.node_id, node.time,
            P1.person_id),
            loc as P2.loc,
            start_time as P1.start_time,
            end_time as P2.end_time,
            node_id as node.node_id,
            person_id as P1.person_id,
            speed as distance(P1.loc, P2.loc)
                  /(P2.end_time-P1.end_time)
   where seq(P1, P2; SRC_PERIOD_PD) and
            P1.person_id = P2.person_id and
            distance(P1.loc, P2.loc) <= RUNNING_DIST and
            distance(P1.loc, P2.loc)
                  /(P2.end_time-P1.end_time)
                  > SOURCE_PERIOD_PD*MAX_SPEED
```

These specifications are naturally expressed with both spatial and temporal constraints that limit the distance and interval between subevents. People can only run so fast, so a spatio-temporal constraint prevents spurious matches from distant, unrelated events. Furthermore, these constraints allow DCP to operate efficiently, disseminating subevents only far enough to meet other relevant events. Without such constraints, a global event detection process would have to occur which would reduce the performance of the system.

Finally, the *person_detected* events can be generated by the *person_detector* on each sensor node, which constantly analyzes the stereo images taken by the stereo camera.

```
 sensor  person_detector
 schema int person_id,
            double[2] loc
primitive person_detected
      on person_detector as PD, node
 schema event_id as hash(node.node_id, node.time,
            PD.person_id),
            loc as PD.loc,
            start_time as node.time,
            end_time as node.time,
            node_id as node.node_id,
            person_id as PD.person_id
```

Although complex events are decomposed in a top-down manner, DCP uses a proactive approach for event processing. Events are constantly generated by lower-level event processors and pushed into higher-level event processors. Whenever an event processor produces an event, it looks up the system configuration to find high-level event processors that operate on this type of subevent, then sends the event to their location. In such a way, events of all complexities can be detected with low delay.

In the rest of the paper, we will use this example application to illustrate how our constraint processing framework works and evaluate its performance with simulation.

3. DISTRIBUTED CONSTRAINT PROCESSING

In this section, we describe how complex events are detected in the Distributed Constraint Processing framework. DCP detects events in a decentralized manner, avoiding global collection trees, and balancing the computational and network load across participating nodes.

We build our constraint matching on top of geographic hash tables because they are a natural fit for our needs: they use geographic, rather than node-based addressing and they provide a matching mechanism that is scalable and fault-tolerant. We extend GHTs to provide a local matching service.

3.1 Regional GHTs

In applications that detect events in a spatial area covered with wireless sensors, we expect that most events contain regional or temporal constraints because they are triggered by related phenomena, perhaps detected by various nearby sensor types. In our example application, people chasing may only be considered a suspicious behavior when it happens in a certain high security area, and people will only be considered to be chasing if they are running in close proximity. DCP leverages these constraints to obtain significant performance improvements without compromising correctness.

Data-Centric Storage [10] introduced the Geographic Hash Table (GHT) for wireless sensor networks. GHTs hash keys into geographic coordinates within the network topology, and store key-value pairs at the sensor node geographically closest to the hashed location. The canonical form of the GHT hash function is *coordinates = hash(key)*.

To preserve spatial locality of the events, we extend the canonical GHT hash function to create *Regional GHTs* which take `keys` and `regions` into account during hashing. The `region` defines the boundary of a geographic area. The extended regional hash function returns coordinates within the specified region. The regional hash function is *coordinates = hash_r(key, region)*.

Figure 8 shows the difference between a normal GHT and Regional GHT. Note that the event is stored much closer to its original location in a Regional GHT. In a Regional GHT, lookups must specify a matching region to find a particular event.

3.2 Hierarchical Event Processing

To use Regional GHT for spatially constrained event detection, we first divide the sensor network field into a grid of tiles. The size of the tiles can be determined by the constraints expressed in event
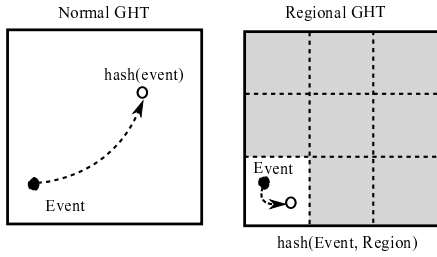
Figure 8: Normal GHT hashes events to global coordinates. Regional GHT hashes events to coordinates within a given region which is the lower left tile.

composition specifications, the resolution of sensor readings, the density of the event detectors, or simply picked arbitrarily. When an event occurs, it is stored in the tile covering the event location.

When processing a regional query for a particular area, the query will be sent to *all* tiles overlapping the queried area. Each of these sub-queries will use the Regional GHT to find the location where the interested data would be stored in each tile. A Regional GHT avoids the need to store events at arbitrary locations in the (potentially large) sensor field, though lookups may need to explore a few tiles if the queried area is large, or falls on a tile border.

Moving beyond support for pull-based queries, we can extend Regional GHTs to detect complex events, as specified by the language of Section 2. Here, events are not only stored at the location they are created, they are also pushed to a rendezvous point determined by the hash of the event type of any complex event specification for which they may be a part. Event processors at that location attempt to construct complex events that meet the event specification.

Sensor nodes constantly produce primitive events with metadata such as timestamp and location information. When a lower-level event is detected, it is sent to all higher-level event processors that need the lower-level event as input. Due to the locality preserving effect of the Regional GHT, the lower-level events only need to be sent to the higher-level event processors in tiles that contain the lower-level event's location (with occasional additions, described in Section 3.4).

As higher-level events are computed, these events may be sent to the locations of even higher level event processors. When low-level events are combined into a complex event, redundant data is removed, and only the attributes attached to the new event are pushed to higher-level processors, usually at a lower rate than the lower-level events.

Hierarchical event processing is performed efficiently from bottom up. At each level, events are hashed and distributed evenly within the tiles due to the advantage of GHTs, and can be directly accessed by ad-hoc queries. A Regional GHT is basically a spatial index making spatial queries efficient.

In the application of Section 2.3, there is a Person Detector on each stereo camera node, so the *person_detected* primitive events are stored at the nodes where they are detected, and propagated to *running_person* detectors. The *running_person* and *running_group* also propagated to the processors for the specifications in which they are referenced. Finally, *people_chasing* events are sent to a

base station for human attention. Figure 9 shows how the events are processed in a hierarchical order. Note the *people_chasing* events are stored at the same node where the dependent *running_group* events are produced. This is an optimization decision explained in Section 3.6.
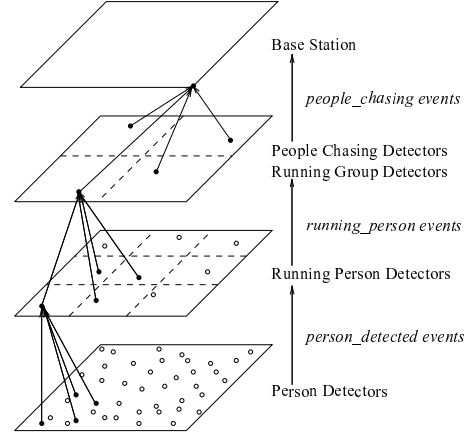
Figure 9: The hierarchical event processing for people chasing detection. Each event detector accepts multiple lower-level events as input and produces higher-level events. Events are pushed from bottom up to the base station.

In DCP, the original GHT's `put()` function is used for sending lower-level events to higher-level event processors, rather than storing directly. So the `key` in the `put()` function is key of high-level events, while the `value` is the lower-level event. However, the `get()` function still works the same way, returning the event data associated with the `key`. This is because we use the proactive approach to propagate events. Lower-level events are pushed to higher level event processors using the `put()` function, rather than the event processors fetching the lower-level events using the `get()` function. Instead, the `get()` function is used only to perform a regional query. After a higher-level event processor receives lower-level events as input, it may store the lower-level events locally for the purpose of time-related event aggregation, but these lower-level events are not returned from ad-hoc query executions.

3.3 Temporal Rehashing

To further balance the transmission and computation load in the network, *Temporal Rehashing* periodically changes the hash location of a given key to eliminate hot spots in the network. With *Temporal Rehashing*, the locations of the complex event processors will be periodically changed, altering the nodes which receive and store the events. Therefore, *Temporal Rehashing* load balances the bandwidth, CPU, and power usage among nodes. The form of the hash function for Regional GHT with Temporal Rehashing is *hash_rt(key, region, time)*.

`time` is the timestamp of the event. Just as DCP divides the sensor region into regularly spaced grids, time is divided into periods of known length. Two events with times in the same period (and equal keys and regions) will be hashed together. If the period differs, the events will be hashed independently, though the returned coordinates will still fall within the same geographic region.

3.4 Interest Area and Interest Interval

In space-related event aggregations, higher-level event processors often express constraints between their lower-level events, rather than absolute constraints. "Find to two people with 3 meters of one another." rather than, "Find any people in the auditorium." This implies that lower-level events may require forwarding to tiles besides the ones they are located in, so that they may be matched and high-level events can be computed.

Taking the *running_group* event for example, when two people are running near an edge shared by two tiles, they may be running close to each other but on different sides of the edge. If the *running_person* events are only sent to tiles containing their locations, this *running_group* event will not be detected. To detect the *running_group* event, the Running Group Detectors in both tiles should be able to observe both of the two *running_person* events. We introduce the notion of *Interest Area*, which represents the area around a lower-level event's location that may impact a higher-level event processor. The size of the Interest Area is determined by the spatial constraints that the higher-level events place on the lower-level events. Figure 10 shows how Interest Area causes events to be sent to multiple tiles.
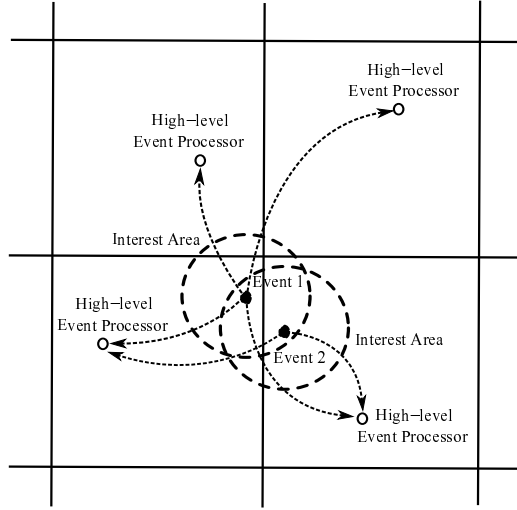
Figure 10: Interest Area causes events to be sent to multiple higher-level event tiles to allow space-related event aggregations. Event 1's Interest Area overlaps with 4 tiles, so it will be sent into all 4 tiles. Event 2's Interest Area only overlaps with 2 tiles, so it will be sent into 2 tiles.

For the same reason, in time-related event aggregations, when temporal rehashing is used, events must be sent to the hash locations of different time periods to allow time-related aggregations. Taking the *running_person* event for example, when a person starts running right before the time of rehashing, and stops running running immediately after the time of rehashing, the two *person_detected* events happen before and after the rehashing time need to be sent to the hash locations in both time periods. Analogously to the Interest Area, we introduce the notion of *Interest Interval* which is the time interval around an event time that may affect higher-level event processors. The length of the Interest Interval is determined by the temporal constraints the higher-level events place on the lower-level events. Figure 11 shows how Interest Intervals cause events to be sent to the hashed locations for multiple time periods.

Interest Area and Interest Interval are used to guarantee that no
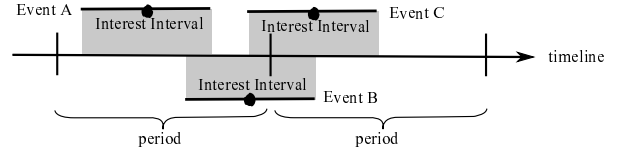
Figure 11: Interest Interval causes events to be sent to event processors of multiple time periods to allow time-related event aggregations. The Interest Intervals for Event B and C overlap with two time periods, so they will be sent to hash locations for both periods. Event A's Interest Interval only overlaps with its own time period, so it will only be sent to the hash location of its own time period.

events are missed because of the usage of Regional GHT and Temporal Rehashing. Whenever a higher-level event has spatial or temporal constraints on its dependent lower-level events, an Interest Area or Interest Interval will be applied to the lower-level events. When a higher-level event depends on more than one type of lower-level events, each type of lower-level events can have different Interest Area sizes and Interest Interval lengths. The effects of Interest Area and Interest Interval may be compounded. For instance, if Event 1 in Figure 10 and Event B in Figure 11 are the same event, it will be sent to $4 * 2 = 8$ different hashed locations.

We now show how to map the event specification to the size of Interest Area and the length of Interest Interval, taking the *running_person* event as an example. Referring to the specification of the *running_person* event in Section 2.3, if the *MAX_SPEED* a person can run equals to 10 m/s, and *SRC_PERIOD_PD* = 0.5s, the Interest Area for the dependent *person_detected* events will be a circle around the person's location with a radius of *SOURCE_PERIOD_PD* * *MAX_SPEED* = 0.5*10 = 5 meters, which means the locations of two consecutive *person_detected* events that can trigger a *running_person* event can be at most 5 meters apart. Here we assume all the stereo camera sensors are synchronized in time. When a person is running closer than 5 meters to the edge of a tile, this person will also be reported to the Running Person Detectors in the other tiles within 5 meters range. Therefore, when the person runs into an adjacent tile, he/she will be immediately detected running by the Running Person Detector in the that tile. The interest interval can be easily picked as two times the period of the *person_detected* events being pushed to the Running Person Detectors, which is 2*SOURCE_PERIOD_PD* = 2*0.5 = 1 second.

In queries that match disparate events, such as "find a blue ball within 10m of a red ball," the sum of the Interest Areas for each event must be 10m. Any appropriate combination may be selected, with the expected rarity of each event and the reuse of each event in other queries playing a role in selecting an appropriate trade-off.

3.5   Event Implementation
In order to realize the Hierarchical Event Processing and perform complex event detection in practice, the base implementation of *Event* has the following important fields: *event_id*, *event_type*, and *target_type*. *event_id* is the event identifier. Events are identified by the event name, or a system-wide unique identifier. *event_type* indicates whether this event is primitive or complex. *target_type* tells how the event processors are located in the network. Its value can be `self`, `ght` or `base`. When *target_type* is `self`, there is an event processor on each node, processing the lower-level events generated on the current node. When *target_type* is `ght`, the net-

work field is divided into tiles. The field *tile_size* indicates the size of the tiles. There is an event processor in each tile with its location computed by the Regional GHT. If Temporal Rehashing is used for this event, there will also be a *rehash_period* field. When *target_type* is base, the event processor is on a base station, all lower-level events are sent to the base station. There are additional *target_id* and *target_loc* fields to provide the node_id and geographic coordinates of the base station. This target_type is used to simulate a global query issued from a base station for data collection purpose.

Each event has a list of *source_event_ids*, which are all the lower-level events that make up this event. The subevents can have different spatio-temporal constraints, so each lower-level event can have a differently sized *interest_area* and different length of *interest_interval*.

We assume that for any specific application, all events and their dependencies are specified *a priori*. The DCP framework leverages this information and forms a hierarchically connected event processing map as discussed in Section 3.2.

Each event processor has a *process*() function. This function is performed whenever a lower-level event is received by the event processor. This function aggregates lower-level events into higher-level events, using local storage to temporarily store lower-level events for temporal aggregation. When a new complex event is produced, it is forwarded to the higher-level event processors that are dependent on this event type.

3.6 Optimization

If the characteristics of the queries in the application is known *a priori*, such as the distribution of the query regions, the events being queried, and the frequency of the queries, the tile size for Regional GHTs can be optimized to minimize the network utilization.

Supposing an Interest Area of radius R for the input events, there exists an optimal tile size with side length L which minimizes (on average) the total distance D that an event must be transmitted to reach all relevant event processors. When L is small, the event location has a higher chance of being near a tile edge and the event will be sent into multiple tiles which increases D. When L is large, the random location of the event processor may be far away from the event location, which also increases D. Our simulation shows the experimental relation between D/R and L/R, as shown in Section 4.2.

Another optimization can be done when the source events of a complex event only come from the same node. In this case, the complex event processor can be located at the exact node where the source events are produced. This optimization eliminates the unnecessary event delivery. For example, the People Chasing Detectors detect *people_chasing* events by comparing two *running_group* events with same person ids, so the *people_chasing* events can be stored at the same node where the dependent *running_group* events are produced, as shown in Figure 9. This optimization requires prior knowledge of the queries in order to choose a hash function that hashes the two queries together.

4. EVALUATION

We show the the advantages of in-network processing allow distributed constraint processing to produce *more* efficient sensor networks while simultaneously decreasing their complexity. We show how DCP compares to a centralizing algorithm by examining the load distribution and total bandwidth consumed during event collection. We separately evaluate the effectiveness of temporal rehashing by showing how DCP performs without temporal rehashing.

4.1 Experimental Setup

We conduct experiments with the NS2 [8] network simulator. Experiments are run in a 300m by 300m square with 200 randomly placed and oriented stereo cameras, each with an 802.11 network interface of 40m range. Several people are moving in the square using a random way-point model with speeds between 0 and 7m/s and no pause time. Our application seeks to find the runners, which we define to be those persons moving faster than 5m/s. One person is considered to be the *guard*, constantly chasing the closest running person. The *guard* starts idle and looks for anyone else that is running. If there is at least one person running, the *guard* immediately starts chasing (at 10 m/s) the closest runner. When the *guard* catches the running person, it matches the speed of the runner, so they are running close together. After the person being chased changes to a speed lower than 5m/s, the *guard* becomes idle and looks for another running person to chase.

Objects can be seen by a camera if the objects are within 40m of the camera, and the camera is orientated in the proper direction. Cameras are assumed to have a 90 degree viewing angle and take pictures twice per second. There is a Person Detector on each camera node, producing a *person_detected* event whenever the simulated camera sees a person. Running Person Detectors are placed in a grid of 100m by 100m squares with a rehashing period of 30s. Running Person Detectors receive all *person_detected* events with a 2.5m radius Interest Area. Running Group Detectors and People Chasing Detectors are placed in a grid of 150m by 150m squares with rehashing period of 50s. The 10s Interest Interval assumes a *people_chasing* event is detected when two persons are running close to each other for more than 5s. *running_person* events are sent to People Chasing Detectors every 5s and their Interest Area is a circle with 10m radius. Whenever a *people_chasing* event is detected, it is sent to the base-station located at (0,0). Each simulation runs for 1000s. GHT uses the GPSR [4] routing protocol to forward packets to destination locations. We turn off the GPSR's perimeter mode which is used to bypass holes in the network. In our experiments, we chose a dense node deployment to allow better camera coverage, so there are unlikely to be any holes in the topology.

4.2 Tile Size Selection

We first analyze the effect of varied tile sizes on the performance of Regional GHTs. A large tile to interest area ratio (L/R) requires every GHT store to travel further, while a small ratio requires multiple stores due to Interest Area overlap with nearby tile edges. We run the simulation for different ratios for the Running Person Detectors with $R = 2.5$m. Figure 12 shows the simulation result. Since camera range and radio range are similar, *person_detected* events can almost always be sent to hash location in one hop when $L < 40$. The hop value shown in Figure 12 is discrete. Each hop covers at most 40m in distance, the maximum radio range, which is about $16R$. When L is larger than $15R$, the average event delivery distance in hops increases slowly. This is due to the effect of discrete hops. We expect that the a faster increase will be shown when L is large compared to the hop distance, not R.

4.3 Bandwidth Distribution
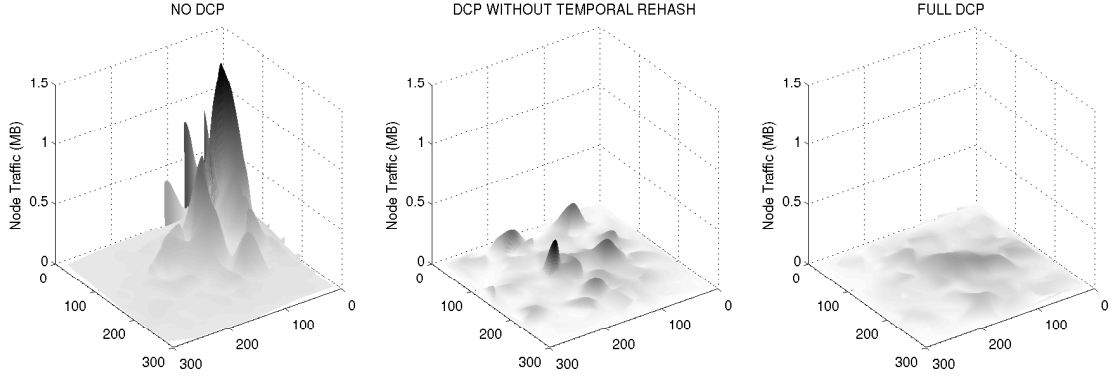
Figure 13: Bandwidth distribution comparison of Centralized Processing (left), DCP (right), and DCP without Temporal Rehashing (middle). Centralized algorithm sends all primitive events back to base-station. Both DCPs send *people_chasing* events back to base-station. Figures show the node traffic distribution created by all event packets.
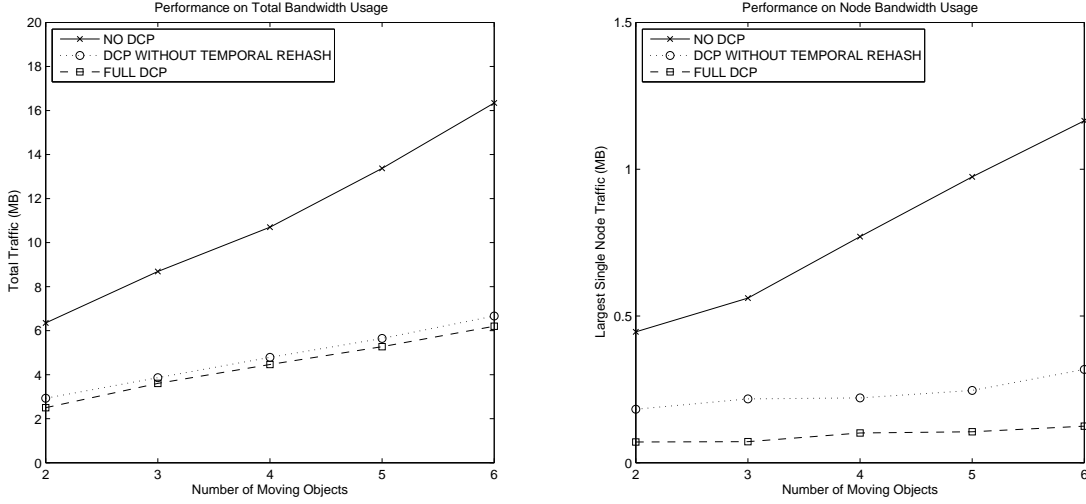

Figure 14: The bandwidth usage comparison of centralized algorithm, DCP without temporal rehashing, and DCP. The number of moving objects is increased from 2 to 6 (including the *guard* object). Left image shows the total traffic in the network. Right image shows the traffic of the busiest node.

DCP distributes complex event processors throughout the network using the Regional GHT. We test the network traffic with three different detection techniques: Centralized Processing (No DCP), DCP without Temporal Rehashing, and FULL DCP (with Temporal Rehashing). All three experiments detect the *people_chasing* events at the base-station. Figure 13 shows the experimental results. The centralized algorithm, sending all primitive events back to base-station without in-network aggregation, creates a serious hot spot near the base-station. When DCP is used, the network traffic is evenly distributed. The network traffic in the center of the sensor field is more than the traffic on the edge of the field. This is because the moving objects tend to move near the center where more events are created. Without Temporal Rehashing, traffic still tends to build up around several locations where the event processors are placed. By using temporal rehashing, DCP further balances the network traffic.

It is worth noting that abstractions that allow neighborhoods or regions of nodes to be defined would not allow purely local aggregation. A technique similar to our Interest Area approach would be required to find matches that span regions. This difficulty is a core reason we avoid the intermediate notion of region and proceed directly to data-centric event constraints.

4.4 Bandwidth Usage

Not only does DCP balance the network traffic, it also reduces the number of radio transmissions because events are usually sent to close destinations and require fewer hops. Figure 14 shows that DCP cuts the total network traffic by about 65%. Equally important, the traffic of the busiest node is much lower under DCP and grows slowly with additional moving objects, prolonging network lifetimes. The adoption of Temporal Rehashing further reduces the requirements on the most heavily loaded node by sharing the work of event processing over time.
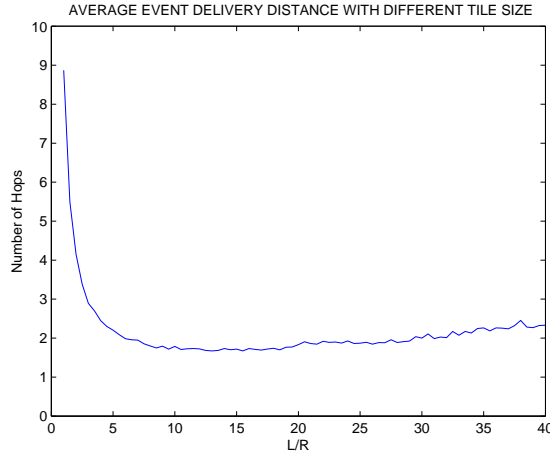
Figure 12: Simulated relation between the average delivery distance (counted in hops) of the *person_detected* events and tile side length L. When the radius of Interest Area R is fixed, the average event delivery distance reaches its minimum of about 1.7 hops when L is approximately $15R$.

4.5 Node Failure

The fault-tolerant aspect of GHTs is discussed in detail in [10]. Here we show the high-level impact of this fault-tolerance. We show how the ability of the sensornet to detect high-level events is affected by the loss of camera and radio nodes. Our example object tracking application depends on radio connectivity, but also on camera coverage to detect the low-level events in the first place. As cameras and radios are lost, detection suffers.

We run the simulations with fewer nodes and report the number of complex *people_chasing* events detected. Dead nodes are unable to generate *person_detected* events nor communicate through radio. We examine the low-level *person_detected* events detected by the live nodes to calculate the ideal number of *people_chasing* events that could possibly be detected in each simulation. We show the actual number of high-level events detected as a measure of performance. We also repeat the simulation with nodes that function as radio nodes, but lack cameras. This separates the influence of network connectivity and sensor availability. Figure 15 shows that at densities over 70% of our baseline, there is little effect on the aggregation abilities of DCP. In all cases, about 5-8% of high-level events are missed, perhaps due to network partition or congestive losses. Below 70%, radio coverage becomes a problem. If the "dead" cameras continue to function as radio nodes, DCP performs well, still finding about 90% of potential complex events. However, if "dead" nodes have neither camera nor radio, the success rate of DCP detecting complex events degrades to about 75%.

5. RELATED WORK

This paper presents a sensornet programming model, including a declarative language to express events with spatio-temporal constraints and an efficient event detection framework to provide runtime support for the programming model. In this section, we compare our work with other existing work in the area.

Many approaches have been proposed to provide programming abstraction and communication models for sensor networks. Our work differentiates itself from existing approaches in its focus on
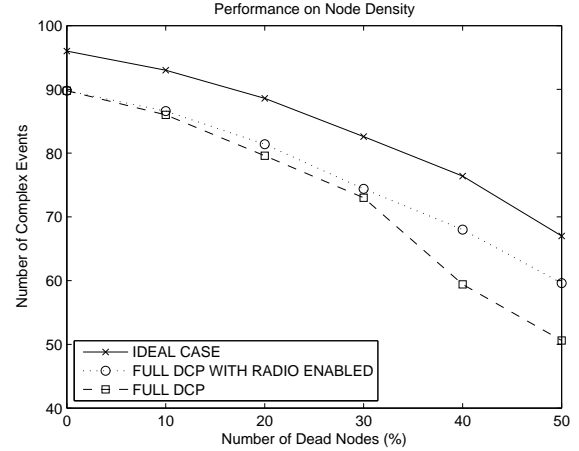
Figure 15: The effect of node density on DCP effectiveness. Simulations are run with different node densities, from 200 cameras in a 300m by 300m field, down to 100 cameras. With fewer cameras, fewer high level events are detected due to lack of camera coverage, and decreased radio connectivity. The top line show how many high-level events *could* be detected if all low-level events were aggregated. The middle line shows how many high-level would be detected if some percentage of cameras, but not radios, are turned off. The bottom line shows the effect of complete outages (camera and radio). Results are the average of five simulations.

removing any sensor-oriented aspects of the programming abstractions.

Our programming model resembles the database-based approaches, such as TinyDB [6] and Cougar [15], which express sensor data of interest in a network-independent way using SQL-style queries. Comparatively, our programming model is built in a similiar way of those in the active database area [2, 3, 9] and is tailored to allow the expression of events with complex spatio-temporal constraints.

EnviroSuite [5] is an environmentally immersive programming framework which uses object-based model to abstract interactions between physical objects and the runtime environment. Our DCP framework is event-based and focuses on expressing hierarchical constraints. Hood [14], Abstract Regions [13], and Regiment [7] present sensor programming models based on groups of nodes defined by their physical proximity or network topology. One may view these groups as sets of nodes that follow constraints that may be laid out in our declarative language. We believe that event constraints represent a similar level of abstraction with the added benefit of removing the need to consider nodes at all when specifying application behavior. Event constraints may express "a red ball within 10 meters of a blue ball," while regions based on node membership cannot, if the node may detect objects at a distance.

The DCP framework decouples event location from node location by extending GHTs [11] as the address mechanism. GHTs were introduced to provide Data-Centric Storage (DCS) [10] for wireless sensor networks. In DCS, events are hashed to geographic locations by event names and stored at the closest node to the hashed location. GHT uses GPSR [4] to route packets to the destination locations. DCP extends GHTs to *Regional GHTs* which preserve

spatial locality in events and allow local operation despite large-scale network partitions.

6. CONCLUSION AND FUTURE WORK

Programming sensornets is well recognized as a hard problem, and data-centric techniques have emerged as a way of taming the associated complexity. In this paper, we have described an area in which existing sensornet programming paradigms have not yet embraced a data-centric approach. We have filled that gap with a distributed constraint processing engine for constructing complex events from subevents. Details of the sensor network are abstracted away so that constraints may be expressed directly between events, rather than through an intermediate abstraction based on the node location or attributes. We believe this separation is particularly important for future sensornets that will integrate more node that sense events at a distant.

Despite the increased level of abstraction, our approach to distributed constraint processing is efficient, scalable, and fault-tolerant because it uses local resources to process local events.

7. REFERENCES

[1] B. Brumitt, B. Meyers, J. Krumm, A. Kern, and S. A. Shafer. Easyliving: Technologies for intelligent environments. In *HUC*, 2000.

[2] S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data Knowledge Engineering*, 14(1), 1994.

[3] S. Gatziu and K. R. Dittrich. Detecting composite events in active database systems using petri nets. In *RIDE-ADS*, 1994.

[4] B. Karp and H. T. Kung. GPSR: greedy perimeter stateless routing for wireless networks. In *MobiCom '00: Proceedings of the 6th annual international conference on Mobile computing and networking*. ACM Press, 2000.

[5] L. Luo, T. F. Abdelzaher, T. He, and J. A. Stankovic. Envirosuite: An environmentally immersive programming framework for sensor networks. *ACM Trans. Embedded Comput. Syst.*, 5(3), 2006.

[6] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: A tiny aggregation service for ad-hoc sensor networks. In *OSDI*, 2002.

[7] R. Newton, G. Morrisett, and M. Welsh. The regiment macroprogramming system. In *IPSN*, 2007.

[8] The Network Simulator - ns-2. `http://www.isi.edu/nsnam/ns/`.

[9] N. Paton and O. Diaz. Active Database Systems. *ACM Computing Surveys*, 1(31):63–103, 1999.

[10] S. Ratnasamy, B. Karp, S. Shenker, D. Estrin, R. Govindan, L. Yin, and F. Yu. Data-centric storage in sensornets with ght, a geographic hash table. *Mob.Netw.Appl.*, 8(4), 2003.

[11] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. GHT: a geographic hash table for data-centric storage. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*. ACM Press, 2002.

[12] G. Simon, M. Maróti, Á. Lédeczi, G. Balogh, B. Kusy, A. Nádas, G. Pap, J. Sallai, and K. Frampton. Sensor network-based countersniper system. In *Proc. SenSys*, 2004.

[13] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *NSDI*, 2004.

[14] K. Whitehouse, C. Sharp, D. E. Culler, and E. A. Brewer. Hood: A neighborhood abstraction for sensor networks. In *MobiSys*, 2004.

[15] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Record*, 31(3), 2002.