Database-Centric Programming for Wide-Area Sensor Systems

Shimin Chen† Phillip B. Gibbons‡ Suman Nath†‡

†Carnegie Mellon University
{chensm,sknath}@cs.cmu.edu

‡Intel Research Pittsburgh
phillip.b.gibbons@intel.com

Abstract

A wide-area sensor system is a complex, dynamic, resource-rich cloud of Internet-connected sensing devices. In this paper, we propose *X-Tree Programming*, a novel database-centric programming model for wide-area sensor systems designed to achieve the seemingly conflicting goals of expressiveness, ease of programming, and efficient distributed execution. To demonstrate the effectiveness of X-Tree Programming in achieving these goals, we have incorporated the model into IrisNet, a shared infrastructure for wide-area sensing, and developed three widely different applications, including a distributed infrastructure monitor running on 473 machines worldwide.

I. Introduction

A wide-area sensor system [3, 13, 14] is a complex, dynamic, resource-rich cloud of Internet-connected *sensing devices*. These devices are capable of collecting high bit-rate data from powerful sensors such as cameras, microphones, infrared detectors, RFID tags, and vibration sensors, and performing collaborative computation on the data. A sensor system can be programmed to provide useful *sensing services* that combine tens to millions of live sensor feeds, as well as traditional data sources. An example of such a service is a Person Finder, which uses cameras or smart badges to track people and supports queries for a person's current location. A desirable approach to developing such a service is programming the collection of sensors as a whole, rather than writing software to drive individual devices. This provides a high level abstraction over the underlying complex system, thus facilitating the development of new sensing services.

Recent studies [6, 10, 15, 16, 28] have shown that declarative programming through a query language provides an effective abstraction for accessing, filtering, and processing sensor data. While their query interface is valuable, we argue that these programming models are too restrictive for wide-area sensing. Specifically, their focus is on standard database operations (SQL queries using aggregates such as *sum* and *average*), and not the more general functionality that wide-area sensing services require. Thus, it is cumbersome, if not impossible, to implement many important aggregation and filtering operators required by wide-area sensing services (such as complex image processing operators—see Section III for further examples) using these models.

In this paper, we present a novel database-centric approach to easily programming a large collection of sensing devices. The general idea is to augment the valuable declarative interface of traditional database-centric solutions with the ability to perform more general purpose computations. Application developers can write code for accessing, filtering, and processing sensor data, define on-demand (snapshot) and continuous (long-running) states derived from sensor data, and seamlessly combine their code and derived states with a standard database interface. Developers can concentrate on the implementation of core application functionalities without worrying about the physical locations of data, network communications, and basic database operations. Unlike many wireless sensor networks [6, 10, 15, 16, 28] that use a flat relational database model and the SQL query language, we instead use the XML hierarchical database model. Our experience in building wide-area sensing services shows that it is natural to organize the data hierarchically based on geographic/political boundaries (at least at higher levels of the hierarchy), because each sensor device takes readings from a particular physical location and queries tend to be scoped by such boundaries [11]. A hierarchy also provides a natural way to name the sensors and to efficiently aggregate sensor readings [10]. Moreover, we envision that sensing services will need a heterogeneous and evolving set of data types, aggregate fields, etc. that are best captured using a more flexible data model, such as XML.

We call our programming model *X-Tree Programming* (or *X-Tree* in short) because of its visual analogy to an Xmas tree: The tree represents the data hierarchy of the wide-area sensor system, and the Xmas tree ornaments and lights represent derived states and application-specific codes that are executed in different parts of the hierarchy. In X-Tree, sensor data of a sensing service is stored in a single XML document which is fragmented and distributed over a potentially large number of machines. Xpath (a standard query language for XML) is used to access the document as a single queriable unit. With X-Tree, user-provided codes and derived states can be seamlessly incorporated into Xpath queries.

Our X-Tree solution addresses the challenge of finding a sweet spot among three important, yet often conflicting, design goals: expressiveness, ease of programming, and efficient distributed execution. As we will show in Section III, achieving these three goals in the same design is very difficult. X-Tree's novelty comes from achieving a practical balance between these design goals. X-Tree is more expressive than traditional database-centric models—it allows developers to incorporate code into queries, in order to perform complex sensor feed filtering and processing. X-Tree is easier to use than a general purpose distributed programming model—developers benefit from the declarative query interface and need to write code only for new application-specific functionalities. X-Tree hides the underlying distributed nature of the sensor data. Finally, X-Tree enables an efficient distributed execution of user-defined code near the sources of sensor data. In this way, it exploits concurrency in the distributed system and saves network bandwidth by transferring only processed results over the network.

In summary, there are three main contributions of this paper. First, we propose X-Tree Programming, a novel database-centric programming model for wide-area sensor systems, which simultaneously achieves expressiveness, ease of programming, and efficient distributed execution. Second, we present several important optimizations within the context of supporting X-Tree that reduce the computation and communication overheads of sensing services. Our caching technique, for example, provably achieves a total network cost no worse than twice the cost incurred by an optimal algorithm with perfect knowledge of the future. Third, we have implemented X-Tree within IrisNet [3, 10, 11], a shared infrastructure for wide-area sensing we developed previously. We demonstrate the effectiveness of our solution through both controlled experiments and real-world applications on IrisNet, including a publicly available distributed infrastructure monitor application that runs on 473 machines worldwide. A rich collection of application-specific tasks were implemented quickly and execute efficiently, highlighting the expressibility, ease of programming, and efficient distributed execution that X-Tree provides.

The rest of the paper is organized as follows. Section II discusses related work. Section III examines three application examples, describes the challenges and overviews our solution. After Section IV provides background information, Section V illustrates the programming interface, Section VI describes our system support for distributed execution, and Section VII discusses several optimizations. Our experimental evaluation is in Section VIII. Finally, Section IX discusses additional issues and Section X concludes the paper.

II. Related Work

Sensor network programming. A number of programming models have been proposed to program resource-constrained wireless sensor networks. The database-centric programming models (e.g., TinyDB [15, 16], Cougar [6, 28]) provide a declarative interface with a standard query processing language, but the resource constraints of the target domain have forced these models to be simple and efficient, at the cost of generality (e.g., TinyDB's SQL is a subset of standard SQL). In contrast, X-Tree supports a general query processing language with arbitrary aggregation functions. The functional programming models (e.g., programming with abstract regions [19, 27]) are tailored to supporting useful primitives that arise naturally in the context of wireless sensor network communication and deployment models. For example, the *abstract region* primitive captures the details of low-level radio communication and addressing within the sensor network. However, the requirements of a programming model for wide-area sensing are

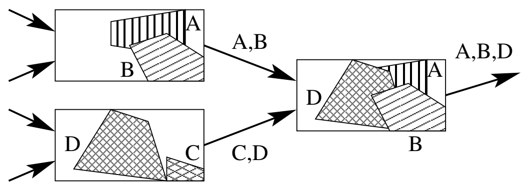


Fig. 1. Merging images in CoastCam

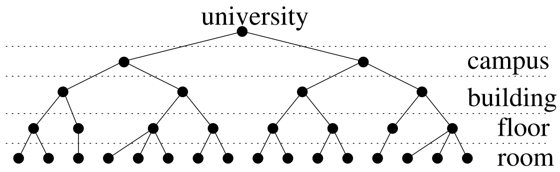


Fig. 2. Logical hierarchy of a person finder service

different—generality of the model is more important than providing efficient wireless communication primitives. Moreover, it is more natural to address wide-area sensors through logical hierarchy rather than physical regions. X-Tree is tailored to achieve these requirements. Proposals for programming with economic models (e.g., market-based micro-programming's pricing [18]) are orthogonal to our model. We believe that X-Tree can be used with such economic models, especially within a shared infrastructure (e.g., IrisNet [3, 11]) where multiple competing services can run concurrently.

Active Networking. X-Tree shares the same general goal with active networking [24] of dynamically programming devices within the Internet. Moreover, some of the design decisions X-Tree makes (e.g., using a Java Virtual Machine) are similar to those used by several active networking systems. However, X-Tree differs from these systems in two important ways. First, X-Tree takes a database-centric view. Second, it presents the whole system as a single queriable unit and hides from the user the physical distribution of both the sensor data and the user-defined code operating on that data.

Distributed Databases. Compared to the existing distributed XML query processing techniques [10, 23] that only support standard XML queries, X-Tree's query processing component is more general in that it supports user-defined operations. Moreover, X-Tree leverages the accessor function approach for executing numeric aggregation functions [5, 15]. However, X-Tree presents a more general model, supporting a richer set of application tasks. Furthermore, X-Tree's stored query construct (defined in Section V) has a similar spirit as the proposal for database fields to contain a collection of query commands in relational environments [22]. While the original proposal aims to support clean definitions of objects with unpredictable composition in a centralized environment, X-Tree aims (1) to support application-specific states, and (2) to enable developers in a dynamic distributed environment to guide the distribution of application code to machines without knowing what data resides on what machine.

III. Example Applications, Challenges, and Our Solution

This section describes a number of wide-area sensing services (we use the terms *service* and *application* interchangeably) that we aim to enable. We also highlight the desirable properties of an enabling programming model, challenges in achieving them, and our solution.

A. Applications

Oceanographic Image Stitching. The oceanographers of the Argus project [1, 12] have deployed a collection of cameras along various coastlines, with each camera taking periodic snapshots of the near-shore ocean surface. The cameras at a coastline can have overlapping views, and the oceanographers would like to combine the snapshots from individual cameras to produce a single panoramic image. If the same physical region is visible from multiple cameras, the resulting panoramic image should have the corresponding pixels from the snapshot having the highest resolution of that particular region. This requires complex image processing algorithms, details of which are omitted here for brevity (please see [12] for details). At a high level, it requires projecting source images into overhead views and then merging them into a single overhead view by taking the highest resolution pixel at every coordinate position from multiple images.

Instead of first collecting snapshots from all the cameras and then generating the panoramic image in a central server, a scalable approach is to combine the snapshots "in-network", as they are sent toward the central server. Figure 1 illustrates one step of this process, where the composite images on the left are merged to form the composite image on the right. This in-network approach enables concurrently running CPU-intensive image processing tasks across many machines, significantly improving system throughput.

Person Finder. A person finder application keeps track of the people in a campus-like environment (e.g., a university campus) and supports queries for the current location of a person. The application uses sensors like cameras, microphones, smart-badges, etc. along with sophisticated software (e.g., for face or voice recognition) to detect the current location of a person.

For scalability, it is important that the sensor data is stored near their sources (i.e., stored by location) and is retrieved only on-demand. One way to implement this application is to maintain a distributed database of all the people currently at each location. A query for some person would then perform a brute force search of the entire database; such a query would suffer from slow response time and high network overhead. A far more efficient implementation would organize the distributed database as a location hierarchy (Figure 2) and prune searches by using approximate knowledge of people's current locations. Such pruning can be implemented by maintaining a Bloom filter (similar to [20]) at every intermediate node of the hierarchy, representing the people currently within that part of the location hierarchy.

Infrastructure Monitor. A distributed infrastructure monitor [2] uses *software sensors* [21] to collect useful statistics (e.g., CPU load, available network bandwidth) on the infrastructure's host machines and communication network, and supports queries on that data. One way to scale such an application to a large number of hosts is to hierarchically organize the data. Figure 3 shows part of the hierarchy used by IrisLog, an infrastructure monitoring service deployed on 473 hosts in PlanetLab [4]. Infrastructure administrators would like to use such an application to support advanced database operations like continuous queries and distributed triggers. Moreover, they would like to dynamically extend the application by incorporating new sensors, new sensor feed processing, and new aggregation functions, as needs arise.

B. Design Goals and Challenges

A programming model suitable for all the above applications should have the following properties. First, it should have **sufficient expressive power** so that application code can use arbitrary sensor data and perform a rich set of combinations and computations on that data. For example, applications may perform complex tasks (e.g., image stitching) on complex data types (e.g., images), and/or combine application-specific states (e.g., Bloom filter) with standard database queries. Second, the model should support **efficient distributed execution** of application code, executing a piece of computation close to the source of the relevant data items. This exploits concurrency in the distributed environment, and saves network bandwidth because intermediate results (e.g., the location of a person) tend to be much smaller than raw inputs (e.g., images of the person). Finally, the model should be **easy to use**, minimizing the effort of application developers. Ideally, a developer needs to write code only for the core application functions. For example, suppose she wants to periodically collect a histogram of the resource usage of different system components, but the infrastructure monitor currently does not support computing histograms. In such a case, it is desirable that she needs to write only the histogram computing function, which can be easily incorporated within the existing monitor.

While achieving any one of the above goals is easy, it is challenging to achieve all three in a single design. For example, one way to provide sufficient expressive power is to enable collecting all relevant data items in order to perform centralized processing, and using application code to maintain states (e.g., Bloom filters) outside of the database. However, this approach not only disables distributed execution, but it requires developers to integrate *outside* states into query processing—a difficult task. To understand the difficulty, consider the Bloom filters in the person finder application. To employ pruning of unnecessary searches, application developers would have to write code to break a search query into three steps: selecting the roots of the hierarchy subtrees for which Bloom filters are stored using the database, checking the search key against the Bloom filters outside of the database, and then recursively searching any qualified subtrees using the database. This simply overburdens the developers.

Similarly, consider the goal of efficient distributed execution. Distributed execution of numeric aggregation functions (mainly SQL aggregates) has been studied in the literature [5, 15]. The approach is

```
<PlanetLab>
  <Country id="USA">
    <Region id="East">
      <Site id="CMU">
        <Host id="CMU2">
          <User id="user1" memUsage="20MB" cpuUsage="5340sec"/>
          ... ...
        </Host>    ... ...
      </Site>    ... ...
    </Region>  ... ...
  </Country>  ... ...
</PlanetLab>
```

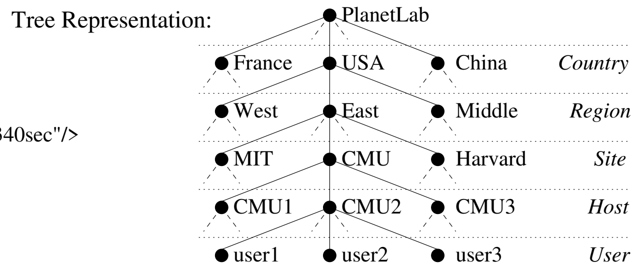


Fig. 3. An XML document representing IrisLog's logical hierarchy

to implement an aggregation function as a set of accessor functions and to automatically distribute the accessor functions. However, it is not clear how to distribute application code for a large variety of possible application tasks that may access and combine arbitrary data items and application-specific states. One may argue that application developers should implement all aspects of the distributed execution of their code. However, this approach requires developers to track the physical locations of stored sensor data and manage network communications, thus violating the goal of ease of programming.

C. *Our Solution*

We observe that although there are a large variety of possible application tasks, many tasks perform similar kinds of computations. For example, a common type of computation is to combine a list of sensor feedings of the same nature (e.g., numeric values or images) to generate a single result (e.g., a histogram or a panoramic image). Other common computation types include (1) computing multiple aggregates from the same set of data sources, and (2) performing group-by before aggregate computations (e.g., computing the total bandwidth consumed by each application on a shared infrastructure). Therefore, our strategy is to treat common computation types and other specific application tasks differently, and provide a higher level of automation for the former.

As mentioned in Section I, X-Tree employs the XML hierarchical database model to organize data in a logical hierarchy, and the Xpath query language for querying the distributed XML database. To enable user-defined computation with XML and Xpath, it provides two components. First, a *stored function* component provides a simple Java programming interface and extends the Xpath function call syntax for implementing and invoking application code. Our implementation of X-Tree (denoted the X-Tree system) automatically distributes the execution of this application code, for the common types of computations. Second, a *stored query* component allows application developers to define derived states and associate Xpath queries with XML elements that invoke their code. In this way, developers can guide the distribution of their code in the logical hierarchy of an XML document, while the X-Tree system hides the details of physical locations of sensor data and network communications. Note that the physical locations of the sensor data can change over time (e.g., for the underlying system's load balancing, caching, etc.). Our implementation of X-Tree works regardless of these dynamics and hides them from developers.

IV. Background: XML Data Model and Distributed Query Processing in IrisNet

An XML document defines a tree: XML elements (tag-pairs) are tree nodes, and their nested structures specify parent-child relationships in the tree. Every XML element has zero or more attributes, which are name-value pairs. Figure 3 illustrates the XML document representing the logical hierarchy in IrisLog. The root node of the tree is PlanetLab. It has multiple country elements as child nodes, which in turn are parents of multiple region elements, and so on. The leaf nodes represent user instances on every machine.

We can use Xpath path expressions to select XML elements and attributes. In Xpath, "/" denotes a parent-child relationship, and "//" an ancestor-descendant relationship. "@" denotes an attribute name instead of an XML element name. For example, `/PlanetLab/Country[@id="USA"]` selects the USA subtree. `//User[@id="Bob"]/@memUsage` returns Bob's memory usage on every machine that he is using, as a list of string values. In order to compute the total memory usage of Bob, we can use the Xpath built-in function "sum": `sum(//User[@id="Bob"]/@memUsage)`. However, the

(a) User query interface to invoke a stored function. (Input_XPATH query selects the set of values to perform computation. Variable number of arguments may be required by the stored function as parameters.)

myClass:mySF (Input_XPATH, arg0, arg1, . . .)

(b) Developers implement three methods for the stored function

```
class myClass {
  String mySF_init      (TYPE val,          String[] args)
  // convert a value of the XPATH output into an intermediate format.

  String mySF_compute (String[] midVals, String[] args)
  // perform computation on a set of intermediate values, merging
  // them into a single intermediate value.

  String mySF_final     (String midVal,     String[] args)
  // generate the final query result from an intermediate value.
}
// if Input_XPATH selects attributes, then TYPE is String
// if Input_XPATH selects nodes, then TYPE is Node
```

(c) Full picture

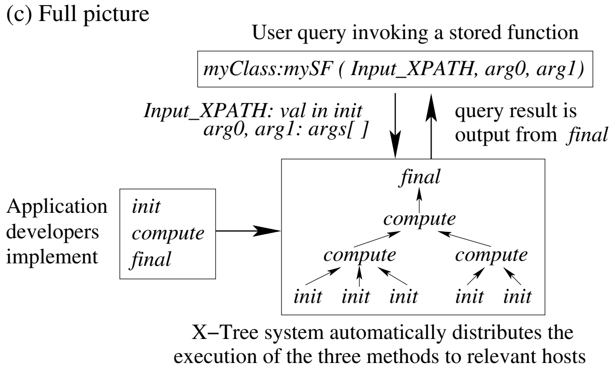


Fig. 4. Stored function programming interface

handful of built-in functions hardly satisfy all application needs, and the original centralized execution mode suggested in the Xpath standard is not efficient for wide-area sensor systems.

IrisNet [3, 10, 11] supports distributed execution of Xpath queries, excluding functions. Sensor data of a service is conceptually organized into a single XML document, which is distributed across a number of host machines, with each host holding some fragment of the overall document. Sensing devices, which may also be hosts of XML fragments, process/filter sensor feedings to extract desired data, and send update queries to hosts that own the data. Each fragment contains specially marked dummy elements, called *boundary elements*, which indicate that the true element (and typically its descendant elements) reside on a different host. IrisNet requires an XML element to have an ID attribute unique among its siblings. Therefore, an element can be uniquely identified by the sequence of ID attributes along the path from itself to the document root. This sequence is registered as a DNS domain entry, for routing queries.

To process an XML query, IrisNet extracts the longest query prefix with ID attribute values specified and constructs an ID sequence. Then, it performs a DNS lookup and sends the query to the host containing the element specified by the prefix; this host is called the *first-stop host*. The query is evaluated against the host's local fragment, taking into account boundary elements. In particular, if evaluating the query requires visiting an element x corresponding to a boundary element, then a subquery is formed and sent to a host storing x. Each host receiving a subquery performs the same local query evaluation process, and returns the results back to the first-stop host. When all the subquery results have been incorporated into the host's answer, this answer is returned.

In the following, we describe X-Tree Programming within the context of IrisNet. However, we point out that our solution is applicable in any XML-based database-centric approach that supports in-network query processing.

V. X-Tree Programming

In this section, we describe the two components of X-Tree Programming, stored functions and stored queries, for efficiently supporting application-specific functionalities in wide-area sensor systems.

A. Stored Functions

The stored function component incorporates application-specific code. Its programming interface is shown in Figure 4. A stored function can be invoked the same way as a built-in function in a user query, as shown in Figure 4(a). The colon separated function name specifies the Java class and the method major name of the application code. The semantics is that the *Input_XPATH* expression selects a list of values from the XML document, the values are passed to the stored function as inputs, and the function output is the result of the invocation. The stored function can require optional arguments, which are typically constant parameters, but can be any Xpath expressions returning a single string value.

```
myClass:histogram ( Input_XPATH, "bucket boundary 0", "bucket boundary 1", ... )   // A set of numeric values is selected to compute the histogram.
class myClass {  // args[ ] specifies the histogram bucket boundaries.
    String histogram_init  (String val,   String[] args )
    // determine the bucket B for a selected value, create an intermediate histogram with B's count set to 1 and all other counts set to 0.
    String histogram_compute  (String[] midVals,   String[] args )
    // merge multiple intermediate histograms given by midVals[ ] by summing up the counts of corresponding buckets.
    String histogram_final (String midVal,    String[] args )
    // generate the query result from the final intermediate histogram.
}
```

Fig. 5. Implementation of a histogram aggregate

As shown in Figure 4(b), application developers implement three Java methods for a stored function: *init*, *compute*, and *final*, which enable the decomposition of the stored function computation into a series of calls to the three methods. For each output value of the *Input_XPATH* expression, the *init* method is called to generate an intermediate value. Intermediate values are merged by the *compute* method until a single intermediate value is left, which is then converted to the query result by the *final* method. The *args* array contains the values of the arguments in the query. As shown in Figure 4(c), our X-Tree system automatically performs this decomposition and distributes the execution of the methods to relevant hosts where the data is located.[1]

Stored functions directly support the ability to perform computation on a single list of values selected by an Xpath attribute query. Examples are numeric aggregation functions, such as sum, histogram, and variance, and more complex functions, such as stitching a set of camera images into a panoramic image. Figure 5 illustrates the implementation of a histogram aggregate. Here, the *Input_XPATH* query is an attribute selection query and therefore the *init* method uses String as the type of its first parameter. However, since arbitrary data structures can be encoded as *String*'s, the interface is able to handle complex inputs, such as images.

Compared to previous approaches for decomposing numeric aggregation functions [5, 15], our approach not only supports string values as inputs, but also allows a list of XML nodes to be selected (Node as input type). The latter enables efficient support for several common computation types, such as computing multiple aggregates from the same set of data sources, and performing group-by operations. These advanced features will be described in Section VII.

B. Stored Queries

Stored queries allow application developers to associate derived states with XML elements in a logical hierarchy. These derived states can be either maintained automatically by our system or computed on demand when used in queries. As shown in Figure 6(a), application developers define a stored query by inserting a stored query sub-node into an XML element. The stored query has a name unique within the XML element. The query string can be any Xpath query. In particular, the query string can be a stored function invocation, and therefore developers can associate application code with logical XML elements.

Figure 6(b) shows how to invoke an on-demand stored query. For each foo element, our system retrieves the stored query string. Then it executes the specified query within the context of the subtree rooted at the parent XML element.

For a continuous stored query, several additional attributes need to be specified, as shown in Figure 6(a). The query can be either in the polling mode or in the triggered mode. When the query is in the polling mode, our system runs the query periodically regardless of whether or not there is a data update relevant to the query. When the query is in the triggered mode, our system only recomputes the query result when a relevant XML attribute is updated. As shown in Figure 6(c), the result of a continuous query is stored as a computed attribute in the database, whose name is the same as the stored query name. A computed attribute can be used in exactly the same way as a standard XML attribute. When adding a stored query, developers can specify a duration argument. Our system will automatically remove expired stored queries.

[1]For those stored functions (e.g. median) that are difficult to decompose, application developers can instead implement a *local* method which performs centralized computation.

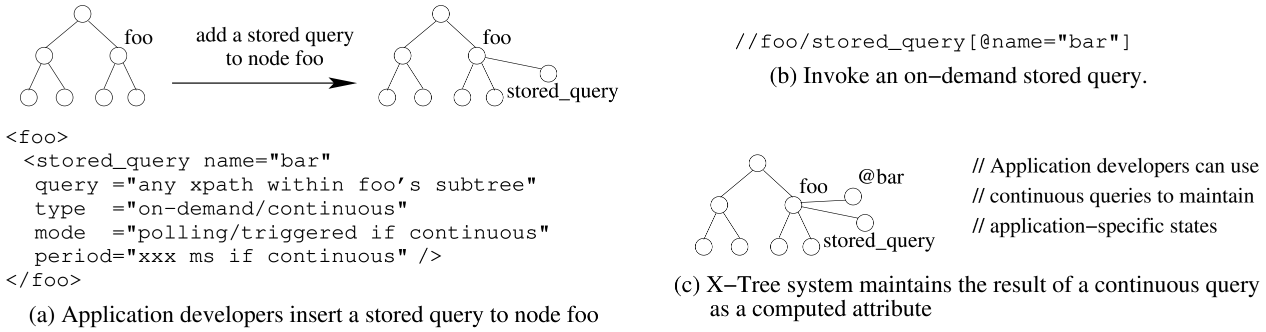


Fig. 6. Defining and using stored queries

Our continuous query implementation is similar to those in Tapestry [25], NiagaraCQ [7], and Telegraph CACQ [17]. Please see [8] for details.

C. Bottom-up Composition of Application Tasks

Combining stored functions and stored queries, our solution supports bottom-up composition of application tasks that may combine arbitrary data items and application-specific states. This is because:

- Application-specific states can be implemented as computed attributes and used in exactly the same way as standard XML attributes.
- The *Input_XPATH* of a stored function may be an on-demand stored query invocation; in other words, an on-demand stored query higher in the XML hierarchy may process results of other on-demand stored queries defined lower in the XML hierarchy. This gives application developers the power to express arbitrary bottom-up computations using any data items in an XML document.

X-Tree Programming saves developers a lot of effort. Developers need not worry about unnecessary details, such as the physical locations of XML fragments, network communications, and standard database operations. They can simply write code as stored functions and invoke stored functions in any user query. They can also associate derived states with any logical XML elements without worrying about the computation and/or maintenance of the states.

VI. Automatically Distributed Execution of Application Code

Stored functions and stored queries can be dynamically added into applications. Developers upload their compiled Java code to a well-known location, such as a web directory. When a stored function invocation (or subquery) is received at a host machine, our X-Tree system will load the code from the well-known location to the host.

Given a stored function invocation, our system automatically distributes the execution of the *init*, *compute*, and *final* methods to the relevant hosts where the data is located. The idea is to call the accessor methods along with the evaluation of the *Input_XPATH* query, which selects the input data.

As shown in Figure 7(a), the stored function invocation is sent to the first-stop host of the *Input_XPATH* query. Our system employs the standard query processing facility (in our case, provided by IrisNet) to evaluate the *Input_XPATH* query against the local XML fragment. As shown in Figure 7(b), the results of querying the local fragment mainly consist of two parts: i) local input data items (squares in the figure), and ii) boundary elements (triangles in the figure) representing remote fragments that may contain additional input data.

Next, our system composes a remote subquery for every boundary element, as shown in the shaded triangles in Figure 7(b). There are two differences between a remote subquery and the original stored function invocation. First, the function name is appended with a special suffix to indicate that an intermediate value should be returned. Second, a sub *Xpath* query is used for the remote fragment. Note that the latter can be obtained with the standard distributed query facility. Our system then sends the subqueries to the remote fragments. On the other hand, our system calls the *init* method for every selected local data item, and the *compute* method to merge multiple local intermediate values (if any).

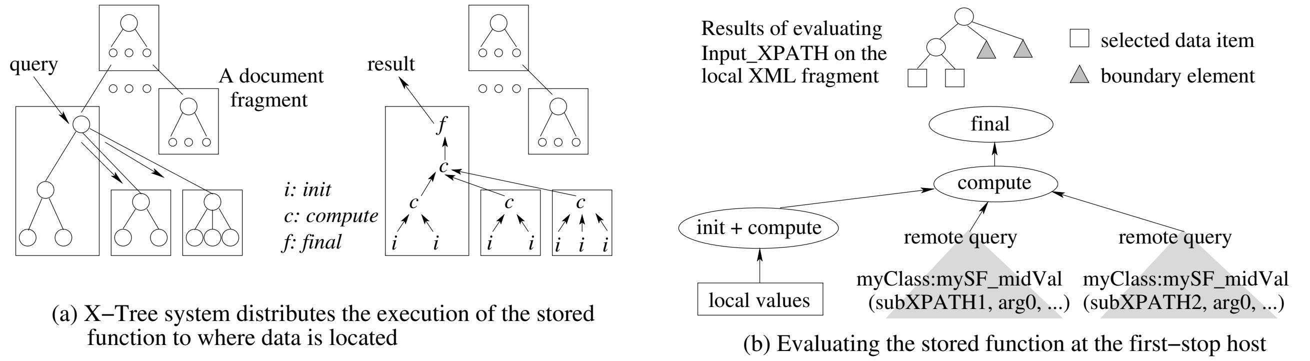


Fig. 7. Automatically distributed execution for *myClass:mySF(Input_XPATH, arg0, ...)*

Finally, when the intermediate results of all the remote subqueries are received, our system calls the *compute* method to merge the local result and all the remote results into a final intermediate value, and calls the *final* method to obtain the query result. Upon receiving a subquery of a stored function invocation at a non-first-stop host, our system performs the same operations except that the *final* method is not invoked.

In summary, our system automatically distributes the execution of stored functions to where the data is located for good performance. This scheme works regardless of the (dynamic) fragmentation of an XML document among physical machines.

In addition to the above mechanism, application developers can use stored queries to guide the distributed execution of their code. They can define stored queries that invoke stored functions at arbitrary logical XML nodes. Upon a reference of a stored query, our system will execute the application code (*final* method) at the host where the associated logical XML node is located.

VII. Optimizations

In this section, we first exploit the stored function Node interface to combine the computation of multiple aggregates efficiently and to support group-by. Then we describe a caching scheme that always achieves a total cost within twice the optimal cost.

A. Computing Multiple Aggregates Together

In IrisLog, administrators often want to compute multiple aggregates (e.g., total CPU usage and maximal memory usage) of the same set of hosts at the same time. A naive approach would issue a separate stored function invocation for every aggregate. However, this approach uses the same set of XML nodes multiple times, performing a lot of duplicate operations and network communications. Therefore, it is desirable to compute all the aggregates together in a single query.

Interestingly, we can enable this optimization through a special stored function that computes an arbitrary number of aggregates at the same time, as shown in Figure 8. Note that there is no need to change the underlying X-Tree system. An example query is as follows:

```
myOpt:multi(//User, "sum", "cpuUsage", "max", "memUsage")
```

The *init*, *compute*, and *final* methods are wrappers of the corresponding methods of the aggregate functions. A *multi*'s intermediate value contains the intermediate value for every aggregate function (for the same subset of XML nodes). Note that *multi* can be used directly in any application to compute any aggregates.

B. Group-By

Using techniques similar to the first optimization, we have also implemented an efficient group-by mechanism, which provides automatic decomposition and distribution for grouping as well as aggregate computations for each group. It supports a set of frequent queries in IrisLog. See [8] for details.

```
myOpt:multi ( Input_XPATH, "function1", "attr1", "function2", "attr2", "function3", "attr3")
// Input_XPATH selects a set of nodes that contain all the specified attributes. For this set of nodes, multi computes "function1"
// on "attr1", "function2" on "attr2", and "function 3" on "attr3".     (Multi supports arbitrary number of functions.)

class myOpt {  // args[2K] is function K, args[2K+1] is attribute K.
    String multi_init  (Node node, String[] args)
    // For every function−attribute pair in args[ ], extract the attribute from the node and apply the function to get an intermediate value.
    // Generate a concatenated intermediate value.

    String multi_compute (String[] midVals, String[] args)
    // For every function−attribute pair in args[ ], extract the intermediate values from midVals[], and merge them into a single value.
    // Generate a concatenated intermediate value from all the computed intermediate values.

    String multi_final (String midVal, String[] args)
    // For every function−attribute pair in args[ ], extract the intermediate value and compute the final result.
}
```

Fig. 8. A stored function *multi* that computes multiple aggregates at the same time

C. Caching for Stored Functions

In the original IrisNet [3], XML elements selected by an Xpath query are cached at the first-stop host in hopes that subsequent queries can be answered directly from the cached data. To exploit caching for stored queries, we slightly modify our previous scheme for executing stored functions. As shown in Figure 9(a), at the first-stop host, the system now has three strategies.[2] The first strategy is the distributed execution scheme as before. Because IrisNet can not exploit cached intermediate values, this strategy does not cache data. The second strategy is to execute the *Input_XPATH* query, cache all the selected XML elements locally, and execute the stored function in a centralized manner by invoking all the methods locally. The third strategy is to utilize existing cached data if it is not stale, and perform centralized execution without sending sub-queries, thus saving network and computation costs.

Cached data becomes stale because of updates. In IrisNet, a user query may specify a tolerance time to limit the staleness of cached data. A piece of cached data is time stamped at the caching time and is only used for queries if its time stamp is within the query tolerance time.

Our system has to choose one of the strategies for an incoming query. For simplicity, we shall focus on improving network cost. Assume for a given stored function invocation, the centralized strategy costs K times as much as the distributed strategy, and the cost of a cache hit is 0. Moreover, we assume all queries have the same tolerance time T. Now we have an optimization problem as shown in Figure 9(a).

To solve this optimization problem, we propose the algorithm in Figure 9(b). This algorithm does not require any future knowledge. It only requires our system to keep per-query statistics so that the Y value can be determined. The typical query patterns are shown in Figure 9(c). The algorithm chooses distributed execution for the first K queries, then it chooses centralized execution for query $K+1$, followed by a period of time T during which all queries are cache hits. This pattern repeats. It is possible that the number of "n" queries in a pattern is more than K. The algorithm ensures that any $K+1$ subsequent "n" queries occur sparsely in a longer period of time than T.

Theorem 1: The algorithm in Figure 9(b) guarantees that the total cost is within twice the optimal cost.

Proof: Consider the query pattern in Figure 9(c). Let us denote the regions containing a series of "n" queries followed by a single "c" query an "nc" region, and denote the period of time T following an "nc" region a "T" region.

Consider an "nc" region containing $M (M \geq K)$ "n" queries and a single "c" query. The cost of our algorithm for this region is $M + K$. Note that M can be greater than K, but the algorithm ensures that no $K + 1$ subsequent queries in these M queries may occur in a period of time T. Therefore, switching an "n" query into a "c" query will have equal or larger cost. Any algorithm must have a cost at least M to handle these queries. The ratio of the cost of our algorithm to the optimal cost is within $(M+K)/M \leq 2$.

Since our algorithm costs 0 for "T" regions, its total cost is always within twice the optimal cost. ■

[2]We focus on the first stop host, but the techniques in this section can be applied at any host processing a subquery.

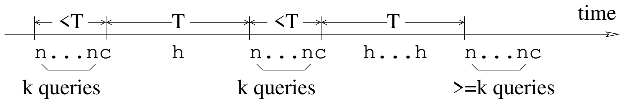
Execution strategy	Abbreviation	Query cost
distributed execution, no caching	n	1
centralized execution to cache subtree	c	K(K>1)
cache hit (within the tolerance time T of the last centralized execution)	h	0

Find an algorithm for choosing the strategy to evaluate an incoming stored function so that the total query cost is minimized.

(a) Optimization problem

```
if (tolerance time has passed) {
  Y = number of distributed executions
      in the previous T time
  if (Y < K)
       distributed execution
  else centralized execution
}
else  cache hit
```

(b) Our algorithm assuming no future knowledge



(c) Typical query patterns of our algorithm

Fig. 9. Optimization for caching

VIII. Evaluation

We have incorporated X-Tree Programming into IrisNet, and then used it to implement the three applications discussed in Section III. Although the Person Finder application is only toy prototype, the other two applications are real-world deployments: The Oceanographic Image Stitching application, called CoastCam, has been deployed with real cameras at the Oregon coast. The Infrastructure Monitor application, called IrisLog, has been deployed on 473 hosts in PlanetLab and has been publicly available (and in-use) since September 2003. The rich and diverse set of application-specific functions and states used by these applications supports the expressive power of X-Tree. The ease of programming using X-Tree is supported by the small number of lines of code needed to implement these applications on our system: 439 lines of code for supporting Bloom filters in Person Finder, 316 lines of code for wrapping CoastCam's image stitching code in stored functions to enable distributed execution, and 84 lines of code for communicating with software sensors in IrisLog.

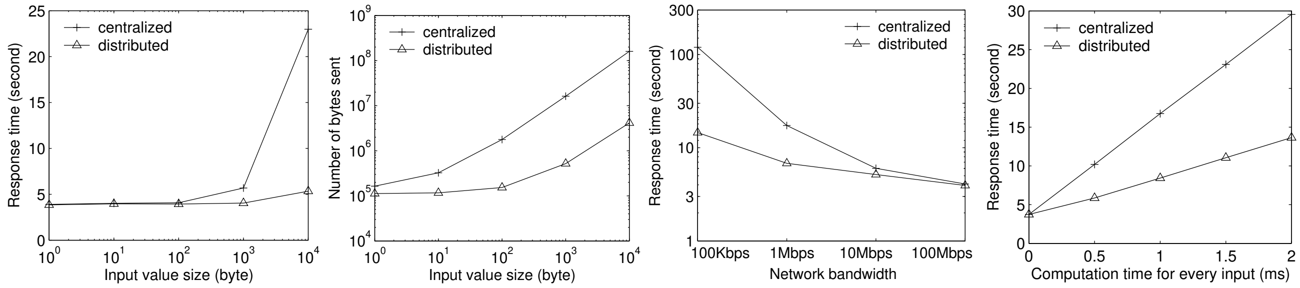
In the following, we present experimental results for both controlled and real-world experiments with these applications. We perform comparative performance studies for various techniques proposed in the paper, including stored function, stored query, and the *multi* stored function. Our approach is to compare the performance of an application when our solution is used against the performance without our techniques. Our aim is to demonstrate the performance benefit achievable by using our solution and thus the effectiveness of our solution. We highlight our findings in Section VIII-C.

A. Controlled Experiments with Person Finder

We perform controlled experiments using the person finder application, as shown previously in Figure 2. We set up 4 campuses in a university, 20 buildings per campus, 5 floors per building, 20 rooms per floor, and on average 2 people per room. Every room element contains a `name_list` attribute listing all people names in the room. We distribute the database across a homogeneous cluster of seven 2.66GHz Pentium 4 machines running Redhat Linux 8.0 connected by a 100Mbps local area network. The machines are organized into a three-level complete binary tree. The root machine owns the university element. Each of the two middle machines owns the campus and building elements for two campuses. Each of the four leaf machines owns the floor and room elements for a campus.

In our experiments, we issue queries one at a time from a 550MHz Pentium III machine. We measure query response time on this machine and the number of bytes sent by the seven server machines. Every result point reported is the average of 100 measurements.

1) Stored Functions: In order to study the benefit of distributed execution of stored functions for saving network cost and exploiting parallelism, we compare the performance of computing an aggregate function using two different approaches. The first approach implements the aggregate using the *init/compute/final* programming interface as a stored function. Therefore, the aggregate computation is automatically executed in a distributed fashion. The second approach extracts all the input values from the database and performs



(a) Response time varying input value size

(b) Communication cost varying input value size

(c) Response time varying network bandwidth

(d) Response time varying computation time

Fig. 10. Distributed vs. centralized execution

centralized execution[3].

The aggregation function we use models the common behavior of numeric aggregates (such as sum and avg), i.e. combining multiple input values into a single output value of similar size. For these experiments, every room element in the database contains a dummy attribute and the aggregations use this attribute. In order to make the size of input values a meaningful parameter to change, we choose to compute bit-by-bit binary OR on all the dummy attributes in the database and update every dummy attribute with a string of a given size before each experiment.

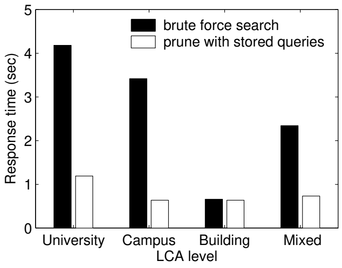
Figure 10(a) and (b) report the response time and communication cost of the two approaches while varying the length of every input value from 1 byte to 10,000 bytes. As the input value size increases, the communication cost of the centralized approach increases dramatically (noting the log scale of y axis in Figure 10(b)), incurring large response time increases beyond 1000B. In contrast, distributed execution only suffers from minor performance degradations. This is because centralized execution requires all input values to be transferred, while distributed execution only transfers intermediate results.

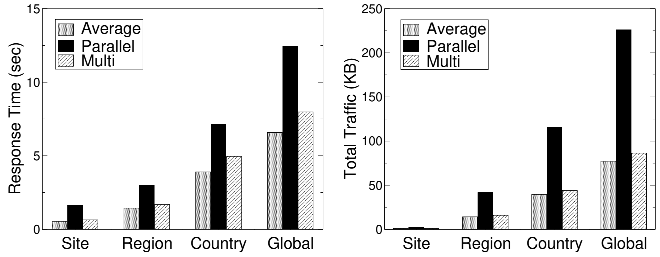
Figure 10(c) varies network bandwidth for the 100B points in Figure 10(a) in order to capture a large range of possible network bandwidth conditions in real use. The true (nominal) network bandwidth is 100Mbps. To emulate 10Mbps network, we change the IrisNet network communication code to send a packet 10 times so that the effective bandwidth seen by the application is 1/10 of the true bandwidth. Similarly we send a packet 100 and 1000 times to emulate 1Mbps and 100Kbps networks. Admittedly, this emulation may not be 100% accurate since the TCP and IP layers still see 100Mbps bandwidth for protocol packets. Nevertheless, we expect the experimental results to reflect similar trends. As shown in Figure 10(c), when network bandwidth decreases, the performance gap between distributed and centralized execution increases dramatically. When network bandwidth is 1Mbps or lower, which is quite likely in a wide area network, distributed execution achieves more than 2.5X speedups over the centralized approach.

Figure 10(d) studies the performance for computation-intensive aggregation functions. To model such a function, we insert a time-consuming loop into our aggregation function so that this loop is executed once for every input value in both the distributed and the centralized approaches. Then we vary the total number of loop iterations so that the whole loop takes 0, 0.5ms, 1ms, 1.5ms and 2ms, respectively, which models increasingly computationally intensive aggregation functions. As shown in Figure 10(d), distributed execution achieves over 1.7X speedups when the computation time is at least 0.5ms per input. This is because distributed execution exploits the concurrency in the distributed database and uses all the seven machines, while the centralized approach performs all the computation on a single machine.

2) Stored Queries: To study the benefit of stored queries, we compare the performance of the brute force search approach and the Bloom filter pruning approach enabled by stored queries. For the latter, we use continuous stored queries to maintain Bloom filters for the building elements, and user queries refer to the Bloom filters using Xpath predicates. At every building element, queries (sub-queries) that do not

[3]We actually implemented this approach using the alternative *local* method in our Java programming interface, which is equivalent to an implementation outside of the XML database system that runs on the root machine.





(a) Response time (b) Network traffic

Fig. 11. Pruning vs. brute force search

Fig. 12. Calling multiple aggregates

satisfy the corresponding Bloom filter are pruned.

We use a 550MHz Pentium III machine for generating background update requests that model the movements of people. In the model, a person stays in a room for time T, which is uniformly distributed between 1 second and 30 minutes, then moves to another room. When making a move, the person will go to a room on the same floor, in the same building, in different buildings of the same campus, and in different campuses, with probabilities 0.5, 0.3, 0.1, and 0.1, respectively.

Figure 11 shows the performance comparison. We measure response times for queries that look for a person in the entire university, in a particular campus, or in a particular building. The mixed workload is composed of 20% university level queries, 30% campus level queries, and 50% building level queries. Since the scope (university, campus, or building) of a query is usually a good guess by the end user, we set up the queries so that a query would succeed and find a person 80% of the times in the given scope.

As shown in Figure 11, the Bloom filter approach achieves dramatically better performance than the brute force approach for queries involving campus or university level elements, demonstrating the importance of stored queries. The building level results are quite close because pruning is less effective in a smaller scope and additional stored procedure overhead almost offsets the limited benefit of pruning.

B. Real World Experiments with IrisLog

Our workload consists of queries with four different scopes. The *global* queries ask for information about all the PlanetLab hosts (total 473 hosts).[4] The *country* queries ask about the hosts (total 290 hosts) within USA. The *region* queries randomly pick one of the regions (*usa-east*, *usa-mid*, or *usa-west*), and refer to all the hosts (around 95 hosts per region) in that region. Finally, the *site* queries ask information about the hosts (around 4 hosts per site) within a randomly chosen site in USA.

PlanetLab is a shared infrastructure; therefore all the experiments we report here were run together with other experiments sharing the infrastructure. We do not have any control over the types of machines, network connections, and loads of the hosts. Thus our experiments experienced all of the behaviors of the real Internet where the only thing predictable is unpredictability (latency, bandwidth, paths taken). To cope with this unpredictability, we ran each experiment once every hour of a day, and for each run we issued the same query 20 times. The response times reported here are the averages over all these measurements. We also report the aggregate network traffic which is the total traffic created by the queries, subqueries and the corresponding responses between all the hosts involved.

Calling Multiple Aggregates Using Multi. Figure 12 shows the performance of computing a simple aggregate (average), computing four different aggregates (average, sum, max, and min) using four parallel queries, and computing the same four aggregates using a single Multi query. For parallel queries, all the queries were issued at the same time, and we report the longest query response time.[5]

[4]Although IrisLog is deployed on 473 PlanetLab hosts, 373 of them were up during our experiments. The query latency reported here includes the timeout period IrisLog experiences while contacting currently down hosts.

[5]Note that IrisNet's query processing module is single-threaded and therefore parallel queries are processed sequentially at every machine. However, different stages of different queries can be executed in parallel on multiple machines and therefore parallel queries are still faster than issuing queries one by one.

From the figures, we see that the average query response time is small considering the number and the geographic distribution of the PlanetLab hosts. There exists a distributed tool based on Sophia [26] that can collect information about all PlanetLab hosts. Sophia takes the order of minutes to query all the PlanetLab nodes [9]. In contrast, IrisLog executes the same query in less than 10 seconds.

Moreover, both the response time and the network overhead of the Multi operation are very close to those of a simple aggregation and are dramatically better than the parallel query approach. The Multi operation avoids the overhead of sending multiple queries (sub-queries), and the packet header and other common metadata in responses. It also avoids redundant selection of the same set of elements again.

C. Summary of Performance Study

Our performance study demonstrates that i) our stored function technique significantly improves application performance, in both response time and network overhead, by providing automatic distributed execution of application code; ii) continuous stored queries enable application-specific states (e.g., Bloom filters for pruning search) that provide significant performance benefit; iii) our techniques are scalable, as shown by IrisLog that has been running on 473 machines worldwide since September 2003.

In addition, we report more results in our full paper [8]. For a group-by query over all the nodes, our scheme achieves 25% speedups in response time and saves 81% network bandwidth in IrisLog compared to the naive approach of extracting all data and computing group-by results in a centralized way. This is because our group-by implementation using the stored function Node interface enables automatic decomposition and distribution of group-by computation. Moreover, we evaluate our solution for stitching images against the centralized approach. We find that image stitching is highly computation intensive. Because distributed execution exploits the parallelism of multiple hosts, our solution increases the throughput of image stitching by 33-286%.

IX. Discussion

X-Tree Programming can be used in more general situations than the three application examples we described. For example, the three application examples do not share sensor data across multiple services; i.e. an XML document is only accessed by a single application. Therefore, our current implementation allows application developers of a service to see the entire XML document. However, in some situations, multiple sensing services may be built on top of the same sensor data. Access control for different XML elements might be important in such situations. One possible approach is to associate access control lists to every XML element and check every operation against the access control lists. Another possible approach is to perform query rewriting similar to that in relational databases to append additional predicates to queries that limit accesses only to the authorized part of a document.

The application tasks described so far are all read-only in nature with respect to the sensor data. However, our system also supports typical XML update queries (XUpdate) to update data items in an XML document. Application developers are free to send update queries from within stored functions, or anywhere else.

X. Conclusion

In this paper, we present *X-Tree Programming*, a novel database-centric approach to easily programming a large collection of Internet-connected sensing devices. Our solution augments the valuable declarative interface of traditional database-centric approaches with the ability to seamlessly incorporate user-provided code for accessing, filtering, and processing sensor data. We demonstrate the effectiveness of our solution through both controlled experiments and real-world applications, including an infrastructure monitor application on a 473 machine worldwide deployment. Using X-Tree Programming, a rich collection of application-specific tasks were implemented quickly and execute efficiently, simultaneously achieving the goals of expressibility, ease of programming, and efficient distributed execution. We believe that X-Tree Programming will enable and stimulate a large number of wide-area sensing services.

REFERENCES

[1] The Argus program. http://cil-www.oce.orst.edu:8080/.

[2] Irislog; a distributed syslog. http://www.intel-iris.net/irislog.

[3] IrisNet (Internet-scale Resource-Intensive Sensor Network Service). http://www.intel-iris.net/.

[4] PlanetLab. http://www.planet-lab.org/.

[5] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. FAD, a Powerful and Simple Database Language. In *VLDB 1987*.

[6] P. Bonnet, J. E. Gehrke, and P. Seshadri. Towards sensor database systems. In *IEEE Mobile Data Management*, 2001.

[7] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD 2000*.

[8] S. Chen, P. B. Gibbons, and S. Nath. Stored Procedures for Distributed XML Databases. Technical report, Intel Research Pittsburgh.

[9] B. Chun. PlanetLab researcher and administrator, http://berkeley.intel-research.net/bnc/, Personal communication, November, 2003.

[10] A. Deshpande, S. K. Nath, P. B. Gibbons, and S. Seshan. Cache-and-Query for Wide Area Sensor Databases. In *SIGMOD 2003*.

[11] P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. Irisnet: An architecture for world-wide sensor web. *IEEE Pervasive Computing*, 2(4), 2003.

[12] R. Holman, J. Stanely, and T. Ozkan-Haller. Applying video sensor networks to nearshore environment monitoring. *IEEE Pervasive Computing*, 2(4), 2003.

[13] P. R. Kumar. Information processing, architecture, and abstractions in sensor networks. SenSys'04 Invited Talk.

[14] J. Kurose. Collaborative adaptive sensing of the atmosphere. SenSys'04 Invited Talk. http://www.casa.umass.edu/.

[15] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *OSDI 2002*.

[16] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *ACM SIGMOD*, 2003.

[17] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously Adaptive Continuous Queries over Streams. In *SIGMOD 2002*.

[18] G. Mainland, L. Kang, S. Lahaie, D. C. Parkes, and M. Welsh. Using virtual markets to program global behavior in sensor networks. In *Proceedings of the 11th ACM SIGOPS European Workshop*, 2004.

[19] R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. In *Proceedings of the First International Workshop on Data Management for Sensor Networks (DMSN)*, 2004.

[20] S. Rhea and J. Kubiatowicz. Probabilistic location and routing. In *INFOCOM*, 2002.

[21] T. Roscoe, L. Peterson, S. Karlin, and M. Wawrzoniak. A simple common sensor interface for planetlab. PlanetLab Design Notes PDN-03-010.

[22] M. Stonebraker, J. Anton, and E. N. Hanson. Extending a Database System with Procedures. *TODS*, 12(3), 1987.

[23] D. Suciu. Distributed Query Evaluation on Semistructured Data. *TODS*, 27(1), 2002.

[24] D. L. Tennenhouse and D. J. Wetherall. Towards an active network architecture. *SIGCOMM Comput. Commun. Rev.*, 26(2):5–17, 1996.

[25] D. B. Terry, D. Goldberg, D. Nichols, and B. M. Oki. Continuous Queries over Append-Only Databases. In *SIGMOD 1992*.

[26] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: An information plane for networked systems. In *Hotnets-II*, 2003.

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

[28] Y. Yao and J. Gehrke. Query processing in sensor networks. In *CIDR*, 2003.