Context Aggregation and Dissemination in Ubiquitous Computing Systems

Guanling Chen and David Kotz
Dartmouth College
Hanover, NH, USA 03755
{*glchen, dfk*}*@cs.dartmouth.edu*

Abstract

Many "ubiquitous computing" applications need a constant flow of information about their environment to be able to adapt to their changing context. To support these "context-aware" applications we propose a graph-based abstraction for collecting, aggregating, and disseminating context information. The abstraction models context information as events*, produced by* sources *and flowing through a directed acyclic graph of event-processing* operators*, then delivered to subscribing applications. Applications describe their desired event stream as a tree of operators that aggregate low-level context information published by existing sources into the high-level context information needed by the application. The operator graph is thus the dynamic combination of all applications' subscription trees.*

In this paper, we motivate and describe our graph abstraction, and discuss a variety of critical design issues. We also sketch our Solar system, an implementation that represents one point in the design space for our graph abstraction, and present the specific choices made by Solar and how its future evolution will be guided by the design discussion.

1. Introduction

In a ubiquitous computing environment (sometimes called pervasive computing), in which a user may interact with dozens or hundreds of computationally enhanced devices, *user attention* becomes a scarce resource. It is unreasonable to expect a user to configure and manage these devices, particularly when the devices and their interactions change as the environment changes around them. "Ubicomp" applications must be aware of the context in which they run [27]. These *context-aware* applications can reduce user distraction by dynamically adjusting their behaviors to the current context, that is, the current state of the user, the current computational environment, and the current physical environment [28].

Context information is derived from an array of diverse information sources, such as location sensors, weather or traffic sensors, computer-network monitors, and the status of computational or human services. A fundamental challenge in ubiquitous computing, then, is to *collect* raw data from thousands of diverse sensors, *process* the data into context information, and *disseminate* the information to hundreds of diverse applications running on thousands of devices, while *scaling* to large numbers of sources, applications, and users, *securing* context information from unauthorized uses, and respecting individuals' *privacy*. In this paper we address this fundamental challenge by proposing a graph abstraction for context information collection, aggregation, and dissemination, and show how it meets the flexibility and scalability challenges. We discuss its security and privacy features in another paper [23].

We discuss the motivation and justification of the graph abstraction in Section 2. In Section 3 we describe the specifics of the graph abstraction. Section 4 overviews several design issues involved in realizing the graph abstraction, by discussing possible approaches and the specific choices we made in our prototype "Solar system"; the details of Solar's architecture and implementation are presented in other papers [7, 8]. We mention related work in Section 5 and summarize in Section 6.

2. Motivation

We arrived at our graph abstraction by considering the structure of applications that consume context infor-

This research has been supported by DARPA contract F30602-98-2-0107, by DoD MURI contract F49620-97-1-03821, by Microsoft Research, by the Cisco Systems University Research Program, and by the USENIX Scholars Program.

Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'02)
0-7695-1647-5/02 $17.00 © 2002 IEEE

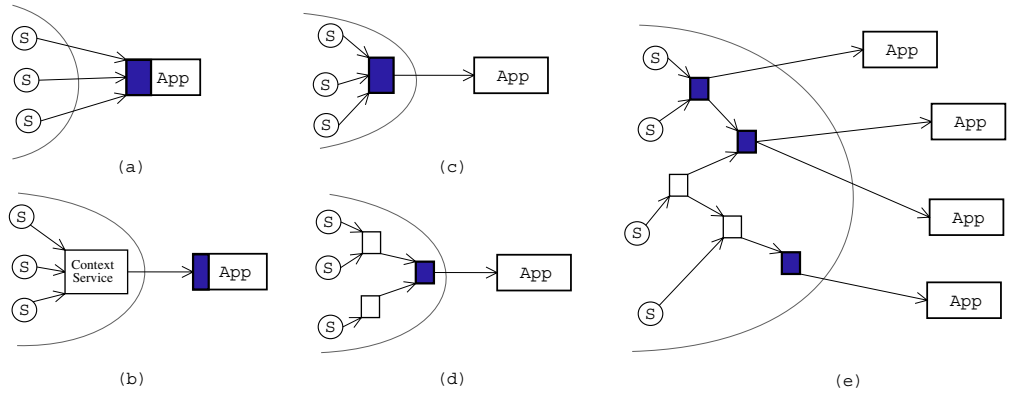

Figure 1. The evolution of data-flow structures.

mation, the proper location for processing sensor data into context information, and structures that can encourage re-use of code and of derived context information. The structure must be flexible and extensible to meet the fundamental challenge of diversity. The structure must also be scalable. Figure 1 sketches an evolution of alternative structures. The circles are data sources, the white squares are operators, and the dark rectangles represent application-specific processing.

A context-aware application attempts to adapt to its changing context by monitoring a variety of sensors. Figure 1a depicts an application receiving sensor data from three sources. The application runs on one platform, commonly a mobile or embedded host. The sensors are located in the infrastructure. This arrangement sends all of the sensor data across the network link to the application platform, and expects the application and its platform to be capable of transforming the raw data into the desired context information. In a situation with slow or unreliable networks, and low-capability mobile platforms, this arrangement is unworkable. With hundreds or thousands of applications and platforms sharing a network connection, it is impossible.

A common approach is to construct a "context service" that receives all of the raw source data, and supplies information about the current context, and changes to the context, to interested applications. (The "location service" seen in many systems is a special case of this approach.) Figure 1b shows that much of the processing has been moved off of the application platform, and may be shared by multiple applications. The context service provider defines the semantics of the context information it provides. While it is possible that the information meets the needs of some applications, in general the applications must process the output of the context service.

Alternatively, the application could push its application-specific processing into the network as a proxy, essentially an application-specific context service. Figure 1c demonstrates this approach. Note, however, that there will be one application-specific proxy for each application, which does not scale well.

We need a compromise that encourages sharing of fundamental aggregation of sensor data into context information, but allows application-specific operations in the network. One possibility (not shown) is to supply a shared context service and install a proxy for each application. Figure 1d takes this approach one step further, decomposing the context service into smaller modules that produce context information of various types and forms. Application-specific proxies may now select the most appropriate inputs to begin their processing.

Finally, in Figure 1e we see that when there are many applications needing context information, they may be able to share both the application-specific as well as the generic processing steps. In the next section, we call this abstraction an *operator graph*. The burden of converting source data into context information is on servers in the network, not on application platforms. The decomposed graph structure improves flexibility, compared to a monolithic context service, and improves scalability, by avoiding a centralized context service, by avoiding the transmission of unnecessary data to application platforms, and by sharing context processing across applications wherever possible.

3. The abstraction

Ubicomp researchers have long recognized the need for context collection, aggregation, and dissemination [10, 12, 30]. The challenge is to allow applications to define their own operations, to describe flexible compositions of operations, and to support many such applications with scalable performance. Given our observations in the preceding section, we propose an abstraction for context collection, aggregation, and dissemination based on a directed acyclic graph. This abstraction can meet

Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'02)
0-7695-1647-5/02 $17.00 © 2002 IEEE

the fundamental challenges of flexibility, scalability, and security, although the discussion of its security aspects can be found in a separate paper [23].

In this section, we introduce the operator-graph abstraction. We explain the specifics about the events, event flows, and operators in the graph. We then sketch an example operator graph for an office scenario. Finally, we discuss the subtle semantics of operator state and "one-time" subscription requests.

3.1. Events, operators, and graphs

Context-aware applications respond to context changes by adapting to the new context. These applications are likely to have an "event-driven" structure, where context changes are represented as *events*. In our graph abstraction, then, we represent context information as events.

We treat sensors of contextual data as *information sources*, whether they sense physical properties such as location, or computational properties such as network bandwidth. Information sources produce their data as events. The sequence of events produced are an *event stream*. An event *publisher* produces an event stream, and an event *subscriber* consumes an event stream.[1]

The event streams represent the data collected by applications from sensors, instead of the queries or control messages sent to sensors, so the event streams are unidirectional in our model. The sources choose when and what events to publish based on the sensor nature, while the downstream subscribers have no control on publishers except using operators to process the published events.

An *operator* is an object that subscribes to and processes one or more input event streams, and publishes another event stream. Since the inputs and output of an operator are all event streams, the operators can be connected recursively to form a directed acyclic graph, an event-flow graph that we call the *operator graph*.

Our operator graph consists of three kinds of nodes: sources, operators, and applications. The *sources* have no subscriptions. They are wrappers for context sensors. *Operators* are functions of their input events and only publish events when they receive an input event. *Applications* are sinks of the graph. They subscribe to one or more event streams and react to incoming events (and possibly other stimuli, such as interactions with the user).

In our operator graph, a directed edge from node A to node B represents that node B subscribes to the event stream published by node A. The operator graph may not be a tree because an operator may subscribe to multiple streams, and its published output stream may have more than one subscriber. In summary, the *publishers* in the graph are the sources and operators, and the *subscribers* in the graph are the operators and applications.

[1] Notice that there is a one-to-one relationship between publishers and event streams. In some other event systems, more than one entity may publish events into an event stream.

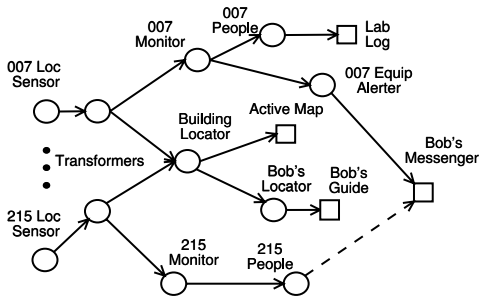

Figure 2. An example operator graph.

There are various types of operators that may perform arbitrary processing functions over incoming events. Some of them are *stateless*, such as a filter selecting location events only about a particular person or a transformer converting the GPS coordinates in the location events into ZIP codes. Other operators are *stateful* and may perform the processing based on accumulated state; for example, a "max-min thermometer" operator outputs an event when it detects a new maximum or new minimum on its input stream of current temperature readings. We discuss operator state further in Section 3.2.

Example. Figure 2 illustrates how the raw events from information sources flow through the operators to become directly usable by the applications. Circles represent event publishers; Squares represent applications that consume the events.

Suppose we have location-tracking sensors installed in each room and badges attached to people and devices. Each time a sensor detects a signal from a badge, it sends out an event containing the badge ID and the timestamp. In the figure these sources are labeled "Loc Sensor" with a room number; each has a transforming operator to map the badge ID to the person or device's name, and to map the sensor ID of the publisher to the location covered by that sensor. The transforming operator then sends out events containing the translated name, the location, and the timestamp.

The *Building Locator* operator subscribes to the current location of every badge, based on the transformed and merged events that originate from the location sensors. It records the current location in its internal state. It generates a "location change" event whenever it sees a badge change location. This output event stream can be used by the *Active Map* application to display the badges' current location in real time. Another sub-

scriber, *Bob's Locator*, filters for changes in Bob's location. Using this information, a *Guide* application [1, 11] running on Bob's PDA can display information related to his current location.

Another reasonable structure, not shown, is to first merge the events from all location sensors and then transform them using only one transformer, to which the Building Locator subscribes. That design is appropriate when using a sensor system that has a centralized architecture, such as Versus.[2]

We would derive a different graph structure if using a different type of location system, such as the Cricket location system, which allows each device to report its own location in a distributed fashion [26].

Returning to our example, the operator *007 Monitor* tracks the set of badges currently in the lab. When a new badge is sensed, it generates a "badge entering" event. When a badge has not been sensed in the past few sensor reports, this operator outputs a "badge leaving" event. The filter *007 People* emits events about people only, not devices. The application *Lab Log* subscribes to that event stream and records the events with timestamp for future reference.

If the *007 Equipment Alerter* receives a "leaving" event for certain equipment, without receiving a "leaving" event for authorized personnel at about the same time, it publishes an alarm event that should be sent to the lab administrator (Bob), whose *Messenger* application displays these alarms on his PDA. If there is nobody in the room with Bob, the Messenger beeps and displays the message. If there are other people in the room, the Messenger vibrates instead. Notice the Messenger subscribes to "215 People" operator (the dashed arrow) because Bob is in room 215 now. This subscription is dynamic and will change as Bob moves around. We discuss the concept of context-sensitive subscriptions in Section 3.4.

There are several advantages of the operator graph abstraction. First, applications receive events semantically closer to their needs than those produced by the sources. Second, due to the modular, object-oriented design we benefit from operator reusability, data abstraction, and maintainability. Third, due to the modular design this operator graph can be deployed across a network and achieve the benefits of parallelism and distribution. Fourth, since filters and aggregators can dramatically reduce traffic along the graph edges, they reduce inter-process (and often inter-host) communication requirements. Finally, by sharing the common operators and event streams the system can support more such applications and more users.

[2] http://www.versustech.com/

3.2. Operator state

Many operators need to keep internal state information to be used when processing events. The state may be simple, as in an aggregator that simply records the previous event to detect changes. The state may be complex, as in an operator that tracks the current location of many users or the current value of every stock on the market.

Our graph abstraction allows the subscriber to choose one of two possible semantics for a new subscription to a stateful operator: 1) the subscription is treated as for stateless operators, or 2) the operator should "push" its current state to the subscriber before any new events are published. In the latter semantics the operator publishes a special sequence of events to the new subscriber only, events that are marked as "state-pushing events" and when considered together represent the current state of the operator. (This feature is reminiscent of the Gryphon expansion operation [3].)

Consider Figure 2. The 007 Monitor maintains a list of badges currently in the lab and publishes changes to this list. The Lab Log logs all the change events, and never needs the original state. The Active Map, on the other hand, needs a "state push" when it first subscribes to the Building Locator, so it can properly locate slow-moving devices like printers.

3.3. One-time subscription requests

The operator graph is an event-oriented abstraction that has publish-and-subscribe interfaces for disseminating information to applications. Occasionally an application may not need the ongoing event stream, but simply needs to obtain the current value. In another system, the application might query the information source. In the operator graph we retain the publish-and-subscribe abstraction by permitting "one-time" subscriptions of stateful operators. An application that needs to obtain the current value of the information published by an operator makes a one-time subscription to that operator. The operator "pushes" its state, as described above, and then cancels the subscription.

The one-time subscription approach has several advantages, largely resulting from its simplicity. There is only one abstraction: publish and subscribe, which streams events from publisher to subscriber. This simplicity avoids the need for additional interfaces and maintains the unidirectional data flow. The programmer of the subscriber can choose one-time or permanent subscriptions based on their needs. The programmer of the publisher need not know anything about queries or one-time subscriptions, only about state push.

3.4. Context-sensitive subscriptions

Many context-aware applications use one aspect of the context (such as the user's location) to subscribe to other information about that context (such as the set of people, devices, or sensors in that location). As the user changes location, the application must cancel its subscriptions, then locate and subscribe to appropriate sources for the new location. These location-sensitive subscriptions are a specific case of what we call *context-sensitive subscriptions*.

It is possible for the application to actively monitor the user's location (for example), and when the user moves to manually adjust its other subscriptions. To reduce programmer effort and to avoid redundant monitoring of the same context by many applications, we aim to support context-sensitive subscriptions directly in the infrastructure. From the viewpoint of the graph, the links representing a context-sensitive subscription are dynamic and the events may flow through different paths as the context changes.

In particular, the context-sensitive subscription supports *physical mobility* of the data sources in the system level. The runtime system that implements the operator-graph abstraction actively adjusts the subscriptions (either to add a new subscription to that sensor because the context matches after movement or to cancel the subscription if the context no longer matches). Any *network mobility* of sources and sinks can be supported by Mobile IP [25] or an application-specific hand-off protocol.

4. Design space and the Solar system

There are many design issues involved in realizing the operator-graph abstraction. In this section we consider the representation of events, how to name operators and event streams, how to route events from publishers to subscribers, and the operator programming model. We first give an overview of the "Solar" system, which implements the graph abstraction. Then for each design issue, we discuss possible approaches, the specific choices made by Solar, and lessons learned from our first Solar prototype.

4.1. Solar overview

Here we present enough background about the Solar architecture to support the following discussion of general design issues. The details of the specific Solar implementation, experimental results, security and privacy model, and applications are not the focus of this paper and interested readers are referred to our technical reports [7, 8, 22, 23].

At the center of any Solar system is a *Star*, which maintains the operator graph and services requests for new subscriptions. When the Star receives a new subscription description from an application, it parses the description, checks the name space, and matches the subscription against its internal data structure representing the operator graph. When it decides to deploy an operator, it instantiates the operator's object on one of many *Planets*. Each Planet is an execution platform for Solar sources and operators. Applications run outside the Solar system and use a small Solar library that allows them to send requests to the Star, and to manage their subscriptions, over standard network protocols. A single Solar system is designed to be deployed in one administrative domain, although in the future we expect to examine wide-area Solar systems.

All event flow is from Planet to Planet, and never through the Star, for scalability. Planets also play a key role in the subscriptions of resident operators. When deploying new subscriptions, the Star tells the Planets to record a subscription from one of its operators to an operator in another Planet. In our implementation there is at most one network (TCP/IP) connection between any two Planets, regardless of the number of operators on or subscriptions between the two Planets.

We now consider the four general design issues.

4.2. Event representation

Perhaps the most fundamental design issue is the choice of a representation for events. Events are passed from publisher to subscriber, typically across a network connection of some kind. Ultimately, any representation agreeable to both publisher and subscriber will work, but there are four typical approaches used by event-distribution systems. The event may be a structure encoded in binary, a typed object appropriate to a particular object-oriented language, a set of attribute-value pairs (usually represented as lines in an ASCII string), or more recently a small XML document. Each representation has advantages and disadvantages.

Events represented as a simple data structure (e.g., a C "struct") and encoded in a binary format, such as XDR [33], usually have smaller size than in the other three formats. While it is small and can be portable, the strict encoding order of the fields makes it difficult for later extension or modification of the existing event structure.

An object representation allows the event to include a complex data structure, if desired. The inherent type hierarchy of an object-oriented language can be used for type-checking when matching subscribers to publishers. Furthermore, type inheritance allows subscribers to process a general class of events even when publishers may send events of more refined subclasses. For example,

a location aggregator can receive any location event regardless of whether it is a GPSLocationEvent, ActiveBadgeLocationEvent, or CricketLocationEvent, if they are all subclasses of LocationEvent. On the other hand, most events are trivial in content and only need a simple common operation (get the values), which makes an object representation overly complex for most event types.

An attribute-value representation is typically more limited than an object representation, although some hierarchical representations (such as INS [2]) do provide structure, and the use of wildcards and implicit fields provide a limited form of inheritance. The simpler representation greatly facilitates content-based event routing (see Section 4.4), and is language- and platform-independent.

Although XML encoding provides more opportunities for structure, XML adds substantial overhead. Parsing every incoming event, constructing every outgoing event, and transmitting information in the verbose XML format, reduces event throughput and adds latency along the event flow. While compressing the XML reduces bandwidth consumption [31], it adds more processing overhead. XML, however, may enhance interoperability and extensibility.

Solar. Our prototype is implemented in Java, so we chose to model events as Java objects and to use Java serialization for event transmission. While an object representation for events hides the internal data encoding and is convenient for operator programmers, our early experiments show that Java serialization posed non-trivial overhead on the latency of event transfer, and reduces overall throughput greatly [7]. Since most events have a simple structure, we plan to use an attribute- or XML-based event representation in our next-generation Solar system.

Another reason to encode events in a simple form is to support context-sensitive subscriptions (CSS; see Section 3.4). Applications specify the conditions when the context information published by some sources should change the application's subscription. Although we show below how Solar's current namespace supports location-sensitive subscriptions (a subset of CSS) by embedding location-related context in the hierarchical namespace, it is awkward to include other types of context in the namespace. An open attribute-based event representation will allow us to support a general CSS abstraction, by enabling applications to encode conditions on the attributes of events.

4.3. Operator naming

We need to name all the data sources so applications can select them as the start of their data flow. A name for a source should be a descriptive handle, including all information necessary to distinguish it from other sources. An alternative approach is for sources to describe the type of events they publish, and for subscribers to choose sources based on the desired type of events. Often, however, much of the descriptive information about a data source naturally resides outside of its event-type space, and it is more flexible and expressive to allow applications to select sources based on such meta information. For example all of the Loc Sensors in Figure 2 output the same event type, but must be distinguished by the *location* of the sensor.

A primary feature of the graph abstraction is the opportunity to re-use the event streams between applications and between users. It is always possible for an application to construct its event flow by describing a tree of generic and custom operators that derive context from a given set of sources. When an application describes such a tree and asks the infrastructure to deploy the operators, the supporting infrastructure can match the new description against the existing graph to identify whether any existing event streams can be used to satisfy all or part of the new request. To make life easier for application programmers, however, it would be helpful if common event streams could be constructed and named by an administrator, or by other users, and then new applications can subscribe to these event streams by name. So we should also allow naming of the event streams (or equivalently, of operators).

The name space (of the sources and operators) could be organized as a tree, as in many file systems. For those publishers given names, the name describes a path from the root to a leaf in the tree. For example, a temperature sensor in Sudikoff room 215 might be named [/Sudikoff/2F/215/temp-sensor]. To enhance scalability, multiple name trees may be federated; perhaps the most common example is the two-level name (hostname:filename) used in URLs.

There are alternative naming architectures with less structure than a tree. Each named publisher could be given a set of descriptive attribute-value pairs [17]. The above temperature sensor might be named [sensor=temperature, room=215, floor=2, building=Sudikoff].

It is arguable whether one approach has clear advantages over the other. In either case the name should be a descriptive handle. In one case the description is a tuple of attributes and values, and in the other case the same attributes may be implicit in the structure of the tree. Both depend heavily on conventions that define the names of the attributes (or structure of the tree) and the range of values (or names of tree links). While the hierarchical tree structure produces concise names and is easy to traverse and explore, the conventions used to

structure the tree are likely stricter and harder to extend than those in a set of attributes, which may make the tree less attractive in a dynamic ubicomp environment.

Another important role for naming is to facilitate resource discovery. In tree-based names a wildcard allows an application to easily describe a large set of publishers, e.g., [/Sudikoff/*/*/temp-sensor/]. The same effect might be obtained in an attribute-based system that allows partial matches, e.g., [sensor=temperature, building=Sudikoff]. While some other operations (such as range selection) might be achievable using tree-based names, the syntax could be awkward.

Sometimes it is natural to use a hybrid approach. Some attributes may depend on another to form a set of attribute trees (as in [2]), or the attribute name itself can have a hierarchical structure (as in the descriptions for X-windows resources). Such hybrid approaches balance the expressiveness of name selection with the efficiency of namespace exploration.

Solar. Solar names sources and operators in a tree-structured name space. Thus, operators have path names like [/Sudikoff/2F/215/temp-sensor]. We extend the tree abstraction with two types of *soft links*. *Alias* nodes bind one name to another. Unlike a Unix symbolic link, however, most alias nodes are designed to change their binding when certain context changes. Thus, the alias [/people/profs/Bob/location] may be bound to [/Sudikoff/2F/215] now, then later to [/Sudikoff/0F/007] when Bob walks to room 007. *Dynamic directories* dynamically compute their set of children based on context. For example, [/Sudikoff/0F/007/people] is a directory whose children are other nodes in the tree, nodes that represent people. The combination can be quite powerful; for example, [/people/profs/Bob/location/people], is a directory containing a list of people co-located with Bob. The list changes when Bob moves or when people enter or leave Bob's current room.

Solar's naming conventions thus encode location context in the name space. Dynamic context is captured with the dynamic soft links. To enable applications to monitor changes to the namespace, and hence changes to the encoded context, all namespace nodes are publishers. Directories publish changes to their set of children, and aliases publish changes to their binding. Aliases and dynamic directories are operators that subscribe to the context information necessary to change their binding or set of children.

The proposed context-sensitive namespace directly supports context-sensitive subscriptions and allows distributed monitoring when combined with our operator-graph abstraction. It is, however, derived incrementally in an ad-hoc way, and strict conventions on the structure may make re-use difficult. We also have found it hard to embed context other than location into the namespace. Realizing these limitations of a tree structure, we plan to use an attribute-based or a hybrid naming approach to improve the expressiveness of data-source selection and to support general context-sensitive subscriptions.

4.4. Event routing

Although the graph abstraction links publishers directly to subscribers, routing events from a publisher to its set of subscribers is essentially a multicast problem that may be implemented in many ways. The simplest approach is to use unicast to send a copy of the event to each subscriber. This approach will not scale when there are many subscribers. Where IP multicast is supported, applications might subscribe and unsubscribe to event streams by joining or leaving particular multicast groups. This approach requires one IP-multicast group for each publisher, however, which is not scalable. An overlay network can use an application-level multicast protocol among a set of servers acting as multicast routers, based either on traditional multicast groups [19] or based on groups defined by names [2].

We can take the multicast concept one step farther to content-based event routing [3, 6, 31]. These systems also use an overlay network of servers, often called *brokers*, which route events to subscribers based on the content (attributes) of the events, not on a destination group or name. In effect, all publishers send events to the global event stream, and subscribers describe the events they want to receive as filters. Siena filters can even recognize event sequences [6]. Some of the systems can encode simple transform operations.

Since event brokers implement simple merge, filter, and transform functionalities, it is tempting to add complex operators like stateful aggregators. Since the brokers are essentially pattern-matching engines, it is unclear whether they might be extended to implement complex operators. All such systems must balance expressiveness with scalability [5].

An interesting compromise is to use content-based routing as the routing substrate of the operator graph, implementing merge and simple filter operations in the event-broker layer. The more complicated operators remain independent operators that subscribe to the routing system and publish events back into the routing system.

Solar. We use a hybrid approach. Solar unicasts events among Planets and multicasts events inside a Planet. In other words, the Planet sends only one copy of each published event to each other Planet where subscribers reside. While this approach uses more network bandwidth than would IP multicast, or even than would an

overlay network implementing multicast, we believe our approach achieves a good balance between bandwidth consumption and delivery latency (no extra processing along the path), and is easily deployed.

Solar has a centralized Star, which is responsible for parsing subscription requests and establishing operator-graph links. The event data-flow traffic, however, never goes through the Star. There may be situations where the Star can become a potential performance bottleneck when subscription and unsubscription happen frequently, for example, an application makes subscriptions whenever some particular context changes. We believe the support for context-sensitive subscriptions in the distributed Planet network will remedy such situations and reduce the load of the Star greatly.

4.5. Programming model

What abstraction do we provide to the application programmer? The programmer may deal explicitly with the operator-graph abstraction, composing an event stream by describing an operator tree based in named sources, and using generic and custom operator classes. Or, the programmer may describe the desired events from named sources using a descriptive higher-level language, which is translated by a compiler into the appropriate operator tree.

The *explicit* approach exposes the operator-tree abstraction to the programmer. The programmer manually derives an event flow that produces context information from named event streams using generic and custom operator classes. Then she uses a subscription language to describe the structure of the tree. She may optionally name the intermediate or final event stream, for others to use.

We speculate that the act of manually deriving an event flow, and the temptation to use existing operator classes where available, will encourage programmers to derive similar trees in similar situations, increasing the opportunities for re-use of event streams. Similarly, programmers will be likely to name event streams for use by other users or in other applications.

With a more complex subscription language, the operator-tree abstraction may be *transparent* to the programmer. Using the language to describe the aggregation of events from named publishers into the desired context, the programmer encodes all of the necessary logic in one program. A compiler translates the subscription into a tree of operators, which is deployed in the same way as in the explicit programming model. The challenge is to invent a subscription language sufficiently powerful to encode complicated aggregations, and yet simple enough to efficiently parse into an operator tree. A language like Java is highly expressive, but a language like SQL, XQuery, or iQL [9] may offer more structure. The language needs a structure that encourages the programmer to describe the event flow in a way that is easily decomposable and likely to match other applications' operator trees. With sophisticated compiler analysis, it may be possible to define finer-grain operators and support finer-grain sharing within the graph. With knowledge of the semantics, the compiler may also be able to rearrange operator trees to allow matches that would otherwise not have occurred. To reduce the complexity of analysis, however, the language would likely limit the expressiveness of operator functionality.

Solar. We use an explicit programming model, in which an application uses an XML-based language to describe its subscription request as a tree of operators and sources. Operators and sources are programmed as Java classes. While the Star is able to match the subscription requests against each other to share operators implicitly, the language also allows application to directly import (explicitly re-use) existing event streams by name. For all other operators in the subscription tree, the application specifies the class and the parameters to initialize the instance. The Solar framework provides a base Operator class from which the programmer derives a subclass implementing three abstract methods: initialization with parameters, event handling, and state push. The application can also optionally name its event streams so other applications can re-use them by name.

We used our Solar prototype in a class, where a group of students designed and developed several context-aware applications [7]. The students found that the explicit programming model was easy to master; ultimately the application developers know the steps necessary to aggregate context data and can decompose the data flow efficiently. We saw little re-use across applications, however, perhaps because the applications were developed simultaneously by independent developers. We are exploring a new language for composing Java operators that may better encourage re-use.

5. Related work

Many have studied context-aware applications and their support systems. In Xerox Parc's distributed architecture each user's "agent" collects context (location) about that user, and decides to whom the context can be delivered based on that user's policy [29, 32]. AT&T Laboratories at Cambridge built a dense network of location sensors to maintain a world model shared between users and applications [16]. Location context can be used to select on-the-spot information for tourist guide applications [1, 11]. HP's Cooltown

project adds Web context to the environment by allowing mobile users to receive URLs sent by ubiquitous beacons [21]. Microsoft Research's Easyliving focuses on a smart space that is aware of the user's presence and adjusts environmental settings to suit her needs [4].

A few projects specifically address the flexibility and scalability of context aggregation and dissemination. Like Solar, the Context Toolkit is also a distributed architecture supporting context fusion and delivery [12]. It uses a *widget* to wrap a sensor, through which the sensor can be queried about its state or activated. Applications can subscribe to pre-defined aggregators that compute commonly used context. These aggregators are deployed as services while Solar allows applications to dynamically insert operators into the infrastructure. Solar dynamically re-uses the operators across applications to reduce redundant event processing and transmission, for scalability. Solar's operator-graph abstraction adapts to changes using context-sensitive subscriptions, which is critical for dynamic ubicomp environments.

Given the type of desired data, some systems automatically construct a data-flow path from sources to requesting applications, by selecting and chaining appropriate components from a system repository [18, 20]. CANS can further replace or rearrange the components to adapt to changes in resource usage [15]. To apply this approach to support context-aware applications, the system manager must foresee the necessary event transformations and install them in the component repository. These systems offer no specific support for applications to dynamically inject custom operators or to share the instances of the components across the applications.

Inserting customized functionalities into data flow paths is not a new idea. Active networks allow programmable switches to host user code customizing the incoming packets [34]. Active Names allow clients to supply a chain of generic or custom components through which the data from a service must pass [35]. Direct diffusion in a sensor network pre-installs application-specific filters on the nodes to reduce traffic [17]. Active Streams support event-oriented inter-process communication, and allow application-supplied *streamlets* to be dynamically inserted into the data path [13].

All of these approaches use a per-user (or per-application) data-flow structure and encourage the re-use of standard components to construct custom event flows. None, to our knowledge, specifically encourage the dynamic and transparent re-use of event streams across applications and users. Solar's re-use of operator instances, and their event streams, avoids redundant computation and data transmission, and improves scalability.

A non-procedural language, iQL, can specify the logic for composing pervasive data [9]. The model supports both requested and triggered evaluation. For one composer, iQL allows the inputs to be continually rebound to appropriate data sources as the environment changes. The language iQL complements Solar in two ways: iQL can be the programming language for individual operators, or iQL can be the high-level subscription language the compiler can decompose into a data-flow tree description used by Solar.

Solar is designed to support a wide variety of sensor data, including computational as well as physical parameters. Solar may then be the delivery mechanism for systems that allow mobile applications to adapt to changes in computational resources. For example, Odyssey applications are aware of the state of resources and can adapt to variations in bandwidth [24] and battery power [14].

As we discuss in Section 4.4, there are many options for event routing. Solar currently uses point-to-point links between a publisher and its subscribers. Although the implementation multiplexes links on Planet-to-Planet socket connections, and implements multicast within a Planet, we may eventually construct an overlay multicast network on the Planets. We may also consider using content-based event routing [3, 6, 31] to support the operator graph. Ultimately, we need to evaluate whether Solar's explicit filter operators will be more or less efficient than the implicit filtering in a content-based event routing system like Siena.

6. Summary

To support context-aware mobile applications, we propose a graph-based abstraction for context aggregation and dissemination. The abstraction models the contextual information sources as event publishers. The events flow through a graph of event-processing operators and become customized context for individual applications. This graph-based structure is motivated by the observation that context-aware applications have diverse needs, requiring application-specific production of context information from source data. On the other hand, applications do not have *unique* needs, so we expect there is substantial opportunity to share some of the processing between applications or users. The situation calls for both flexibility and scalability, and our proposed operator-graph abstraction meets both challenges. It allows the flexible construction of event streams through composition of generic and custom operators. It encourages scalability through re-use of event streams across applications and users wherever possible, by migrating the load off weak mobile application platforms and into powerful network servers, and by distributing that load among many network servers.

We discuss the details of the operator graph abstraction, the semantics of stateful operators and one-time subscriptions, and the context-sensitive subscriptions that make the graph dynamic. There are many ways to realize the graph abstraction in a supporting system, and we discuss design issues involved with event representation, operator naming, event routing, and programming models. We give an overview of our Solar system that implements the graph abstraction. Interested readers can find more details in technical reports about Solar [7, 8, 23] and about a smart reminder application built on top of Solar [22].

References

[1] G. D. Abowd, C. G. Atkeson, J. Hong, S. Long, R. Kooper, and M. Pinkerton. Cyberguide: A mobile context-aware tour guide. *Wireless Networks*, 3(5).

[2] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *SOSP 1999*, South Carolina.

[3] G. Banavar, M. Kaplan, K. Shaw, R. E. Strom, D. C. Sturman, and W. Tao. Information flow based event distribution middleware. In *the Middleware Workshop at the ICDCS 1999*, Austin, Texas.

[4] B. Brumitt, B. Meyers, J. Krumm, A. Kern, and S. Shafer. EasyLiving: Technologies for intelligent environments. In *HUK 2000*, pages 12–29, Bristol, UK.

[5] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving scalability and expressiveness in an Internet-scale event notification service. In *PODC 2000*, Portland, OR.

[6] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3).

[7] G. Chen and D. Kotz. Solar: A pervasive-computing infrastructure for context-aware mobile applications. Technical Report TR2002-421, Dartmouth College.

[8] G. Chen and D. Kotz. Supporting adaptive ubiquitous applications with the SOLAR system. Technical Report TR2001-397, Dartmouth College.

[9] N. H. Cohen, H. Lei, P. Castro, J. S. Davis II, and A. Purakayastha. Composing pervasive data using iQL. In *WMCSA 2002*, Callicoon, New York.

[10] N. H. Cohen, A. Purakayastha, J. Turek, L. Wong, and D. Yeh. Challenges in flexible aggregation of pervasive data. Technical Report RC21942, IBM Research.

[11] N. Davies, K. Cheverst, K. Mitchell, and A. Friday. Caches in the air: Disseminating tourist information in the GUIDE system. In *WMCSA 1999*, Louisiana.

[12] A. K. Dey. *Providing Architectural Support for Building Context-Aware Applications*. PhD thesis, College of Computing, Georgia Institute of Technology, 2000.

[13] G. Eisenhauer, F. E. Bustamante, and K. Schwan. A middleware toolkit for client-initiated service specialization. *Operating Systems Review*, 35(2):7–20, April 2001.

[14] J. Flinn and M. Satyanarayanan. PowerScope: A tool for profiling the energy usage of mobile applications. In *WMCSA 1999*, New Orleans, Louisiana.

[15] X. Fu, W. Shi, A. Akkerman, and V. Karamcheti. CANS: Composable, adaptive network services infrastructure. In *USITS 2001*, San Francisco, California.

[16] A. Harter, A. Hopper, P. Steggles, A. Ward, and P. Webster. The anatomy of a context-aware application. In *MobiCom 1999*, pages 59–68, Seattle, WA.

[17] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan. Building efficient wireless sensor networks with low-level naming. In *SOSP 2001*, pages 146–159, Chateau Lake Louise, Canada.

[18] J. I. Hong and J. A. Landay. An infrastructure approach to context-aware computing. *Human-Computer Interaction*, 16(2&3), 2001.

[19] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole, Jr. Overcast: Reliable multicasting with an overlay network. In *OSDI 2000*.

[20] E. Kiciman and A. Fox. Using dynamic mediation to integrate COTS entities in a ubiquitous computing environment. In *HUC 2000*, Bristol, UK.

[21] T. Kindberg, J. Barton, J. Morgan, G. Becker, D. Caswell, P. Debaty, G. Gopal, M. Frid, V. Krishnan, H. Morris, J. Schettino, and B. Se. People, places, things: Web presence for the real world. In *WMCSA 2000*, pages 19–28, Monterey, CA.

[22] A. Mathias. SmartReminder: A case study on context-sensitive applications. Technical Report TR2001-392, Dartmouth College, June 2001.

[23] K. Minami and D. Kotz. Controlling access to pervasive information in the "Solar" system. Technical Report TR2002-422, Dartmouth College.

[24] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *SOSP 1997*.

[25] C. Perkins. IP mobility support. RFC2002.

[26] N. B. Priyantha, A. Chakraborty, and H. Balakrishnan. The Cricket location-support system. In *MobiCom 2000*.

[27] M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, 8(4).

[28] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *WMCSA 1994*, Santa Cruz, CA.

[29] W. N. Schilit. *A System Architecture for Context-Aware Mobile Computing*. PhD thesis, Columbia University.

[30] A. Schmidt, K. A. Aidoo, A. Takaluoma, U. Tuomela, K. V. Laerhoven, and W. V. de Velde. Advanced interaction in context. In *HUC 1999*, Karlsruhe, Germany.

[31] A. C. Snoeren, K. Conley, and D. K. Gifford. Mesh based content routing using XML. In *SOSP 2001*.

[32] M. Spreitzer and M. Theimer. Providing location information in a ubiquitous computing environment. In *SOSP 1993*, pages 270–283, Asheville, NC.

[33] R. Srinivasan. XDR: External data representation standard. RFC1832.

[34] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Commnunication*, 35(1), January 1997.

[35] A. Vahdat, M. Dahlin, T. Anderson, and A. Aggarwal. Active Names: Flexible location and transport of wide-area resources. In *USITS 1999*, Boulder, Colorado.