Solar: A pervasive-computing infrastructure for context-aware mobile applications

Guanling Chen and David Kotz

Department of Computer Science, Dartmouth College
Hanover, NH, USA 03755
{glchen, dfk}@cs.dartmouth.edu
http://www.cs.dartmouth.edu/~solar/

Dartmouth Computer Science Technical Report TR2002-421
February 28, 2002

Abstract. Emerging pervasive computing technologies transform the way we live and work by embedding computation in our surrounding environment. To avoid increasing complexity, and allow the user to concentrate on her tasks, applications must automatically adapt to their changing *context*, the physical and computational environment in which they run. 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*, which are produced by *sources*, flow through a directed acyclic graph of event-processing *operators*, and are 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 our graph abstraction by discussing several applications under development, sketch the architecture of our system ("Solar") that implements our abstraction, report some early experimental results from the prototype, and outline issues for future research.

1 Introduction

Emerging pervasive computing technologies transform the way we live and work by embedding computation in our surrounding environment. Users can manage their information "anywhere and anytime," using portable devices or devices embedded in the environment. To simplify a user's everyday activities, however, users should not be exposed explicitly to the complexity of new technologies. It is unreasonable to expect a user to configure and manage hundreds of computationally enhanced appliances that she may use in a pervasive-computing environment, particularly when the devices and their interactions change as the environment changes around them.

To reduce user distraction, pervasive-computing applications must be aware of the context in which they run. These *context-aware* applications should be able to learn and

We gratefully acknowledge the support of the Cisco Systems University Research Program, Microsoft Research, the USENIX Scholars Program, DARPA contract F30602-98-2-0107, and DoD MURI contract F49620-97-1-03821.

dynamically adjust their behaviors to the current context, that is, the current state of the user, the current computational environment, and the current physical environment [20], so that the user can focus on her current activity.

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. While the raw sensor data may be sufficient for some applications, many require the raw data to be transformed or fused with other sensor data before it is useful. By aggregating many sensor inputs to derive higher-level context, applications can adapt more accurately.

A fundamental challenge in pervasive 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 should be able to meet the flexibility and scalability challenges. Its security and privacy features are addressed in a companion paper [18].

We propose our graph-based abstraction in Section 2. In Section 3 we show an example graph shared by several context-aware applications that are being developed using our Solar prototype, whose architecture and implementation are discussed in Section 4. We present some results of early experiments with Solar in Section 5. We discuss related work in Section 6 and conclude in Section 7.

2 Operator graph

Context-aware applications respond to context changes by adapting to the new context. These applications are active in nature and the their actions are triggered by asynchronous occurrences. Thus they are likely to have an "event-driven" structure, where context changes are represented 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. An information source *publishes* events indicating its current state or changes to its state. The sequence of events produced are an *event stream*. A sensor with only a query interface can be easily wrapped with a proxy publisher. Context-sensitive applications *subscribe* to event streams that interest them, and react to arriving events to adapt to their changing environment.

Few context-aware applications, however, want to work directly with raw data from information sources. It could be that the application only needs portion of the data, the data is in wrong format, the data is inaccurate, or the data is incomplete and not useful without aggregating other sensor inputs. Thus, sensor data typically needs to go through several processing steps before it becomes meaningful contextual knowledge desired by applications.

One naive approach is to implement all these steps as a monolithic component custom-designed for each application. If we place the component on a mobile platform together with the application, the traffic at the edge of the network may be excessive, since the amount of context needed is usually small compared to the amount of raw data the sources may produce. In addition, placing one independent component for each application in the network does not scale well given thousands of potential pervasive applications.

We observe that many adaptive applications ask for similar (though rarely exactly the same) contextual information, from basics (such as location context) to high-level social context (such as user activity). It is then natural to re-use the overlapping context aggregation functions or sub-functions among applications. Our approach is to decompose the context-aggregation process of every application into a series of modular and re-usable *operators*, each of which is an object that subscribes to and processes one or more input event streams and publishes another event stream. We explore the background and justification for this design in an earlier paper [5].

We identify four common categories of operators. A *filter* outputs a subset of its input events. For example, a sensor publishes the temperature every 10 seconds while one application needs alerts only when the reading exceeds 90 degrees. A *transformer* inputs events of type T1 and outputs events of type T2. For example, a location sensor reports coordinates, but the application needs a symbolic value such as "Lobby." T2 may be the same as T1 if the transformer only changes some attribute values. The *merger* simply outputs every event it receives. For example, an active-map application that displays the current location of all employees merges the readings from all location sensors. While mergers are not strictly necessary, since any of the merger's subscribers could directly subscribe to the same inputs, a merger aids re-use of event streams as discussed later. An *aggregator* outputs an arbitrary type event stream based on the events in one or more input event streams. 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.

Since the inputs and output of an operator are all event streams, the applications can use a tree of recursively connected operators (starting from sources) to collect and aggregate desired context. In contrast to the naive monolithic approach, the application now is able to distribute the tree of operators into network, minimizing the traffic across the network edge and allowing the computation to be distributed. While each application can build its own operator tree, to scale to a large number of applications we must take advantage of opportunities to re-use operators between applications' operator trees. Upon arrival of a new subscription tree, our system attempts to identify subtrees that match a subtree of an existing subscription tree. We define subtree match recursively: two operators match if they are objects of the same class, have the same parameters, and have the same subscriptions. When a match is found, the subtree is clipped from the new subscription tree, replacing it with a subscription to the output of the existing subtree; thus the two subscriptions share the subtree.

These inter-connected overlapping operator trees form a directed acyclic graph, which we call the *operator graph*. There are several advantages of the operator-graph abstraction for context collection, aggregation, and dissemination. 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 re-usability, 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.

3 Applications

To learn whether the operator-graph abstraction is helpful to application developers, we built a prototype of the graph abstraction in a system we call "Solar", and intend to develop several context-aware applications. The next section describes Solar. In this section, we describe a few such applications designed for office environments, and we show the resulting operator graph. At this writing we have groups of students writing these applications, and in a future study we will evaluate both user and programmer experiences.

We have installed an IR-based location system, provided by Versus Technology,[1] throughout our CS department building. The location system contains one or more IR sensors in each room and hallway. These sensors pick up the IR signals emitted from personnel badges and asset tags and report the sightings to a central Versus server. Our applications execute on Fujitsu pen-based tablet computers, each of which runs Windows 2000 and is equipped with a Cisco 802.11b wireless card and a Versus asset tag.

We plan to install a tablet computer on the wall outside of a professor's office or other rooms as an interactive "doorpad", and users can also carry the tablets as mobile devices. We focus on a set of applications that manage office hours, meetings, and interaction between the visitors and the resident of the office. In the following application scenarios we assume one tablet doorpad is installed outside David Kotz's office (room 116).

The display of an office doorpad changes according to the resident's current context. Normally there is a "doorbell" button on the touch-sensitive screen, which is green if David is inside and available, turns yellow if he is on the phone, or turns red if he is in the meeting. There is also a line of text showing related information, such as the time the meeting is scheduled to finish according to the calendar. If the visitor Bob chooses to press the button, the tablet on the David's desk chimes "Bob is at the door" or, if David is at a meeting, pops up a text message.

If David is not in the office, the doorbell button is replaced by an "iNote" button. When Bob clicks the button, iNote recognizes Bob by the proximity of his badge and can display a message, such as "Bob, our meeting is moved to room 214," or "My meeting with George is running late, I will be back in 10 minutes," which David sent for Bob using the iNote application on his mobile device. The current location of David, if available, is also shown on the screen. If Bob decides not to wait, he can simply leave a note that is delivered to David's tablet when he is online.

If David is in the office and it is time for "office hours," an open meeting for students in his course, the doorbell is replaced with a sign-up GUI that contains a menu of

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

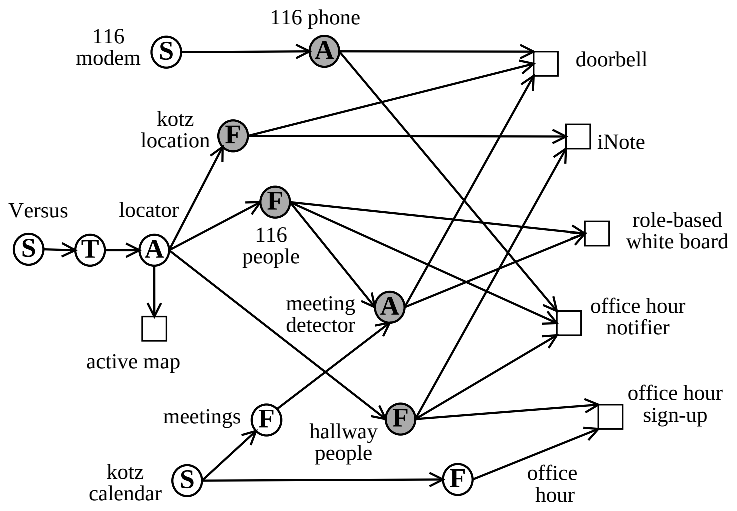

Fig. 1. An example operator graph shared by several context-aware applications. Circles are sources or operators, and rectangles represent applications. Shaded circles represent the operators that are directly re-used by several applications. Letters represent types: S as source, F as filter, T as transformer, M as merger, and A as aggregator.

names (initially including all students in the course), an indication of the number of students already waiting, and the number of minutes left before office hour is scheduled to finish. When the student Chris enters the hallway, the menu refreshes itself and puts Chris on the top. Chris peeks at the doorpad and clicks his name on the menu to sign up; his name is removed from the menu. This context-sensitive menu is similar to the proximate-selection pattern in [20].

Chris does not need to wait in the hallway after signing up. Instead he can go back to the lab for a while, or chat with friends in the lounge. When the previous student is done and leaves David's office, Chris is notified by a pop-up message on his mobile device, or by calling him at the phone in his current room, or (if Chris is still in the hallway) by the doorpad shouting "Chris, it is your turn for office hour". The notification may be delayed if there is an incoming telephone call for David. If Chris is not in the building or is not detected as moving toward David's office, then the next student in line will be notified.

David may hold other meetings in his office or in a conference room. Meeting attendees may interact with a virtual white board using their tablets. While the white board can be accessed remotely, at the discretion of the meeting's chair, only the attendees in the room have write access to the virtual white board. Only the chairperson (David) can delete things from the white board. The white board's content is saved when the meeting is finished, and restored at the start of the next meeting so that they can continue previous discussion.

Consider an operator graph to support these context-aware applications, shown in Figure 1. Circles represent sources and operators, and rectangles represent applications.

The telephone source lives on a host with a modem attached to the phone line, and polls the phone line through the modem interface. On each poll, the source reports that the phone is idle (dial tone detected) or in use (no dial tone detected). The raw event stream produced by the source is then aggregated to provide an event stream indicating only changes in the phone line status.

The Versus source, which monitors all IR sensors in the building, emits an event when a badge's IR signal is picked up by any sensor. The event contains the badge ID and sensor ID, which are transformed into symbolic values using a transformer. The operator labeled as "locator" in the figure aggregates the transformed event stream and produces only location change events. Then various filter operators are applied to its output to produce more specific context for different applications.

The calendar source produces events when a pre-defined activity (such as a meeting, or office hours) is scheduled to start or finish. The calendar information, however, may not always be accurate. Thus the operator "meeting detector" also uses the context about the list of people in room 116 to determine whether there is actually a meeting happening, or whether a meeting is running long.

The event streams produced by the shaded operators in Figure 1 are directly re-used by several different applications without further processing. Many copies of an application run by different users on different machines, such as Active Map, can easily share the operator labeled "locator". Other sharing of operators in this graph is also possible by applications we do not discuss; for example, a tour-guide application on David's mobile device can re-use the operator labeled "kotz location". This sharing of operators dramatically reduces the number of events transmitted across network and avoids redundant computations. This sharing is critical for scalability to support more users and applications.

Even if the existing graph does not provide the exact context needed by some applications, they can start from existing higher-level event streams instead of from scratch. For instance, a reminder on the doorpad notifying David to pick up his printouts needs the context that David is leaving his office (116), which can be derived by applying a filter to either operator "kotz location" or "116 people", while a new source watching print commands on David's workstation provides another piece of context. The new operators contribute to the existing graph and can be re-used by other applications. This flexibility and extensibility makes it easier for developers to program context-aware applications from semantically close event streams, instead of raw sensor data.

Finally, the event-flow and operator-graph abstractions provide a basis for flexible, decentralized access control. Our approach is based on information-flow principles, with relaxation mechanisms to allow flexible user control over access to context about themselves. We describe our security model in a companion paper [18].

4 The Solar system

We are building a prototype infrastructure for context collection, aggregation, and dissemination, based on the operator-graph abstraction. In this section we describe the system architecture of our prototype, the "Solar" system, and discuss some implementation details.

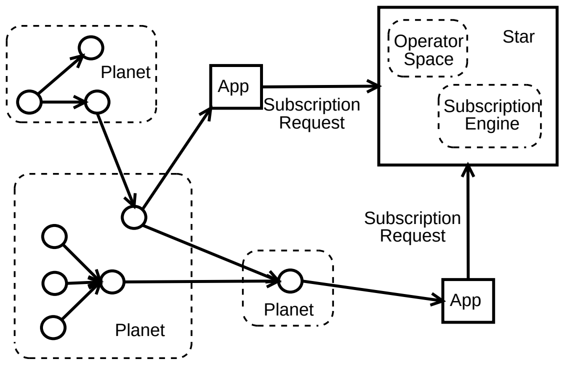

Fig. 2. The architecture of Solar.

4.1 Architecture

The Solar system consists of several components (see Figure 2). A centralized *Star* processes subscription requests from applications and deploys operators onto appropriate *Planets* as necessary. A Planet is an execution platform for Solar sources and operators, and it is responsible for tracking subscriptions and delivering events in the operator graph. We now discuss each component in detail.

The Star maintains a representation of the operator graph and services requests for new subscriptions. When the Star receives a new subscription-tree description, it parses the description, and matches the subscription tree against its internal data structure representing the operator graph to see whether there are some existing operators that can be shared and re-used. When it decides to deploy a new operator, it instantiates the operator's object on one of many *Planets,* which periodically register themselves with the Star. The Star determines which Planet should host the new operator by considering the Planet's load and network traffic between Planets. In essence, it attempts to map the operator graph onto the Planetary network to distribute load and avoid congestion

Planets play a key role in the subscriptions of resident operators. When deploying new subscriptions, the Star tells the Planets to arrange a subscription from one of its operators to another operator, possibly in another Planet. Thus the Planet maintains all the subscriptions for each of the resident operators. When an operator publishes an event, the hosting Planet delivers the event to all the subscribing operators (that may reside on several Planets) and applications. When a Planet receives an event, it dispatches the event to the appropriate resident operator.

Applications run outside the Solar system and use a small Solar library to interface with Solar. The small library allows the applications to send requests to the Star, to manage their subscriptions, and to receive Solar events over standard network protocols.

4.2 Implementation

Our Solar prototype is implemented in Java. It models events as Java objects and uses Java serialization for event transmission. The operators are small Java objects that implement a simple publish/subscribe interface.

A context-aware application uses an XML-based language to describe its subscription request, which is a tree of operators and sources. The language allows the application to directly import (explicitly re-use) existing operator objects by name. For all other operators in the subscription tree, the application specifies the class and the parameters to initialize the instance. The application can optionally name its operators so other applications can re-use them by name.

The Star parses the subscription request, and matches it against an internal data structure to determine whether any existing operators can be shared. Explicit sharing occurs when the subscription mentions an existing operator by name. Implicit sharing occurs when the Star notices that a subtree in the subscription matches one in the existing graph, as described earlier.

When the Star decides it is necessary to deploy a new operator, it randomly chooses a Planet to host the operator. While randomization can achieve some load balancing, we plan to investigate more clever graph-embedding algorithms that attempt to balance the load and minimize network traffic.

When the Star asks a Planet to deploy an operator, the Planet loads the operator's Java class from the local CLASSPATH or remote Web server and initializes a new instance with parameters supplied in the XML subscription request. The Planet maintains one outbound event queue for each resident source or operator, and a dedicated thread takes events from this queue and sends them to Planets hosting the subscribers. We multiplex operator subscriptions onto inter-Planetary TCP/IP sockets, so that there are at most two one-way TCP/IP connections between any two Planets, regardless of the number of operators on or subscriptions between the two Planets. The Planet's Network Manager thread monitors the inbound sockets and fills an inbound event queue; a dispatcher thread removes events from this queue and enters a reference for the event into the incoming event queue for each destination operator. Each operator has a dedicated thread to invoke the operator's event handler as new events arrive.

4.3 Programming model

Solar provides a framework for developers. Developers can write a new source or operator by inheriting from the appropriate base class and implementing a few abstract methods. When a source or operator needs to publish an event, it simply calls an inherited *publish(IEvent)* method. An operator's *handleEvent(IEvent)* is automatically invoked when the Planet receives an event destined to that operator. The application developer uses an XML-based subscription language to construct its event-flow tree, and a small library to interact with Solar as discussed earlier.

To determine the value of the operator-graph abstraction and programming model, and the performance of the Solar system, we are developing several real-world context-sensitive mobile applications, some of which are discussed in Section 3. We installed an IR-based location system to supply location context to our Solar system and its applications. We plan to add more information sources to enrich the context space and to explore the performance and flexibility of the operator-graph abstraction.

5 Experiments and results

In this section we present early results of our experiments with the Solar prototype. Solar's graph-based abstraction decomposes a particular application into several pieces and distributes them across network. From a single application's viewpoint, its computation is scattered, leading to additional latency and overhead. Our expectation is that these inefficiencies can be traded against overall system scalability due to distribution, shared processing, and the migration of computation toward information sources. In this initial round of experiments, we quantify the fundamental overhead of Solar's publish/subscribe implementation.

5.1 Experiments

To measure event-delivery throughput and latency, we set up a single event source and multiple event sinks on different hosts. We compare three approaches to send messages from source to sinks: 1) a message is a byte array and the source sends it directly through sockets, 2) a message is an event object and the source sends serialized events without Solar, and 3) the source sends serialized events through Solar's publish/subscribe interface. This approach allows us to identify the overhead of serialization separately from the overhead of Solar.

In the throughput measurement, the source first sends a message to start the sinks' clocks and then sends another 1000 messages. The sinks compute throughput (messages per second) when the last message arrives. We then plot the average result from all sinks, which are deployed on different hosts. In the latency test, we set up the source and sink on the same host and another "bouncer" on a different host. The sink computes the round-trip delay and averages over 1000 messages.

In the experiments, message content is simply a string and we vary its size N from 0 to 1000 one-byte characters, reflecting our expectation that the payload of most events will be small. The byte-array approach adds 4 bytes (indicating the length) to the outbound data. The approach sending serialized events without Solar sends a MessageEvent that contains the content string and some meta-data, such as timestamp and event-publisher information. The actual size transferred through sockets after serialization is $365 + N$ bytes for an N-character string. Solar further wraps the MessageEvent with additional fields such as the subscriber's ID; Solar sends $463 + N$ bytes through the socket. In each case, the event is serialized separately for each receiver.

We ran the experiments on a set of identical Linux workstations (VA Linux VarStation 28, Model 2871E, 450 mHz Pentium II, 256 MB RAM, 5400 rpm EIDE disk) running the Linux 2.4.2-2 (RedHat 7.1) operating system and Sun's Java VM (1.4.0-rc). We interconnected the computers with a full-duplex 100 mbps Ethernet. The computer and the network were dedicated to our experiments (no other users or traffic).

5.2 Results

Figure 3 shows the results of the latency experiments. It is not surprising to see that Java serialization adds significant overhead, because of the time needed for serialization and because more bytes needs are transmitted through the Socket. The main overhead caused by Solar comes from its queue-based structure. An event passes through several

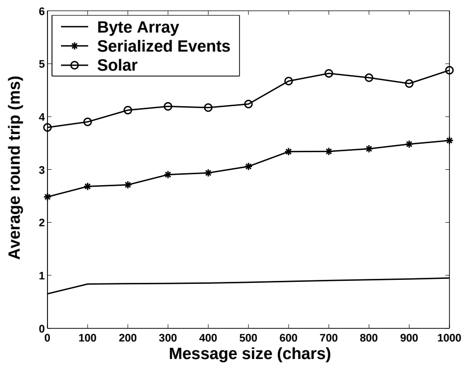

Fig. 3. Latency Tests

queues when traveling from source to a subscribing operator: the `publish()` method places the event in the source's outbound queue (1), the Network Manager thread removes it and places a reference in the outbound queue (2) for each Planet hosting a subscriber, a thread for each connection removes it and pushes it into the socket where it lands in the kernel's TCP buffer (3), in the receiving Planet the event is pulled from the kernel's TCP buffer (4) and placed in the dispatcher's queue (5), the dispatcher thread removes it and places it in the input queue (6) for each local subscriber, and finally the subscriber's thread removes it and calls the event handler. The same process is reversed to bounce an event back to the sink: a total of four system queues and eight Solar queues. The round-trip latency of Solar events is about 1.5 milliseconds more than a similar test that does not use Solar. Thus, Solar adds about .75 milliseconds to each event publication. While acceptable for many applications, we clearly need to optimize this path for better performance and scalability.

Figure 4 shows the results of throughput experiments for 1 sink and 10 sinks. Again, Java serialization causes the throughput to drop significantly compared to sending a byte array, due to the serialization time and additional data to transmit. Solar's throughput is lower still, due to the larger events and additional processing. Solar's throughput is mostly independent of message size, indicating that the fixed overhead for serialization and queue management are dominant. While 1000 events/sec between one source and one sink, or 120 events/sec between one source and ten sinks, is a reasonable performance for many applications, there is clearly room to improve.

In summary, our experimental results indicate that Java serialization overhead was non-trivial, and the overhead would increase for event objects more complex than a simple String. Solar's queuing overhead, on the other hand, was relatively small. Any approach based on object serialization gives the programmer tremendous flexibility, but at the cost of significant serialization overhead.

We are considering several approaches to reduce event-serialization overhead. First, an event should be serialized only once, when sending to multiple subscribers. The latency and 1-sink results show, though, that even one serialization is significant. Special-

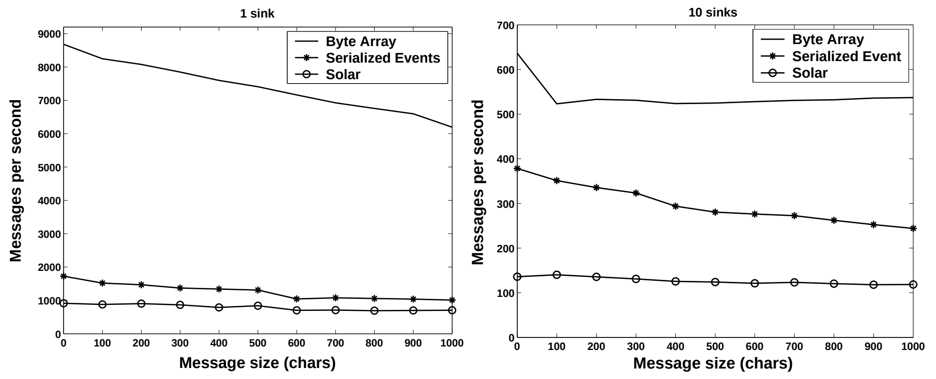

Fig. 4. Throughput Tests

ized Java serialization packages, such as JavaParty,[2] are fast but require each class to implement its own marshaling and de-marshaling methods. Finally, we could represent events as a set of (attribute, value) pairs, rather than as full Java objects; Solar can provide fast generic event serialization, although there is less flexibility in defining event types.

We are also considering ways to improve event distribution in large-scale Solar systems. For large subscriptions, it may be more efficient to deliver events by multicast through an overlay network formed from the Planets, rather than directly to each subscribing Planet.

6 Related work

Many have studied context-aware applications and their supporting 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 [21,23]. AT&T Laboratories at Cambridge built a dense network of location sensors to maintain a world model shared between users and applications [13]. Location context can be used to select on-the-spot information for tourist guide applications [1,6]. HP's Cooltown project adds Web context to the environment by allowing mobile users to receive URLs sent by ubiquitous beacons [16]. Microsoft's Easyliving focuses on a smart space that is aware of the user's presence and adjusts environmental settings to suit her needs [3].

A few projects specifically address the flexibility and scalability of context aggregation and dissemination. Like Solar, the Context Toolkit is a distributed architecture supporting context fusion and delivery [7]. 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. Solar allows applications to dynamically insert operators into the system and compose refined context that

[2] http://wwwipd.ira.uka.de/JavaParty/

can be shared by other applications. The Context Toolkit allows applications to supply filters for their subscriptions, while Solar introduces general filter operators to maintain a simple abstraction. IBM Research's context service, Owl, addresses similar issues such as scalability, extensibility, and privacy [8], but provide no details.

Targeted for distributed sensor networks, Michahelles et al. propose to use context-aware packets to detect desired context [17]. These smart packets contain a retrieval plan that indicates what context sensors to visit to get results. The plan can be updated at run time according the results from certain sensors. The packets may also contain a *context hypothesis*, which can be evaluated at compute-empowered nodes, that derive higher-level context information based on the retrieved raw sensor data. At this point, it is unclear whether these smart packets could be used to deliver notifications about context changes.

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 [15,14]. CANS can further replace or rearrange the components to adapt to changes in resource usage [11]. 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 provide custom operators. Active Names, on the other hand, allow clients to supply a chain of generic components through which the data from a service must pass [24]. Also, Active Streams support event-oriented inter-process communication, and allow application-supplied *streamlets* to be dynamically inserted into the data path [9].

All of these approaches 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.

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 [19] and battery power [10]. Project one.world [12] promotes a new application structure designed to cope with frequent changes in pervasive computing environment. Solar could be a complementary system used by one.world applications to detect the contextual changes.

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 use content-based event dissemination [2,4,22] 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-forwarding system like Siena.

7 Summary

To support context-aware pervasive-computing 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, and we give an example operator graph shared by several context-aware applications that are under development using the Solar infrastructure, which is our prototype system that implements the operator-graph abstraction. We present Solar's architecture and some implementation details, and we show some early results of our experiments with Solar. We describe Solar's access-control model in a companion paper [18].

Acknowledgments

We thank Abram White, for the experimental framework and for assistance with the experiments. We give special thanks to the students of Course CS88/188 of Winter 2002 (Sanket Agrawal, Feng Cao, Guanling Chen, Kazuhiro Minami, LeeAnn Tzeng, Guanhua Yan, Hrithik Govardhan, and Chris Masone), who designed and developed interesting context-aware applications using Solar system. We are also grateful to Kazuhiro Minami, for his work on Solar's security and privacy models.

References

1. Gregory D. Abowd, Christopher G. Atkeson, Jason Hong, Sue Long, Rob Kooper, and Mike Pinkerton. Cyberguide: A mobile context-aware tour guide. *Wireless Networks*, 3(5):421–433, October 1997.
2. Guruduth Banavar, Marc Kaplan, Kelly Shaw, Robert E. Strom, Daniel C. Sturman, and Wei Tao. Information flow based event distribution middleware. In *ICDCS 1999*, Austin, Texas. IEEE Computer Society Press.
3. Barry Brumitt, Brian Meyers, John Krumm, Amanda Kern, and Steven Shafer. EasyLiving: Technologies for intelligent environments. In *HUC 2000*, pages 12–29, Bristol, UK. Springer-Verlag.
4. Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.

5. Guanling Chen and David Kotz. Context aggregation and dissemination in ubiquitous computing systems. Technical Report TR2002-420, Dept. of Computer Science, Dartmouth College, December 2001. Submitted to *WMCSA 2002*.
6. Nigel Davies, Keith Cheverst, Keith Mitchell, and Adrian Friday. Caches in the air: Disseminating tourist information in the GUIDE system. In *WMCSA 1999*, New Orleans, Louisiana. IEEE Computer Society Press.
7. Anind K. Dey. *Providing Architectural Support for Building Context-Aware Applications*. PhD thesis, College of Computing, Georgia Institute of Technology, December 2000.
8. Maria R. Ebling, Guerney D. H. Hunt, and Hui Lei. Issues for context services for pervasive computing. In *Workshop on Middleware for Mobile Computing 2001*, Heidelberg, Germany.
9. Greg Eisenhauer, Fabian E. Bustamante, and Karsten Schwan. A middleware toolkit for client-initiated service specialization. *Operating Systems Review*, 35(2):7–20, April 2001.
10. Jason Flinn and M. Satyanarayanan. PowerScope: A tool for profiling the energy usage of mobile applications. In *WMCSA 1999*, New Orleans, Louisiana.
11. Xiaodong Fu, Weisong Shi, Anatoly Akkerman, and Vijay Karamcheti. CANS: Composable, adaptive network services infrastructure. In *USITS 2001*, San Francisco, California. USENIX.
12. Robert Grimm, Janet Davis, Ben Hendrickson, Eric Lemar, Adam MacBeth, Steven Swanson, Tom Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble, and David Wetherall. Systems directions for pervasive computing. In *HotOS-VIII*, Elmau, Germany.
13. Andy Harter, Andy Hopper, Pete Steggles, Andy Ward, and Paul Webster. The anatomy of a context-aware application. In *MobiCom 1999*, pages 59–68, Seattle, WA.
14. Jason I. Hong and James A. Landay. An infrastructure approach to context-aware computing. *Human-Computer Interaction*, 16(2&3), 2001.
15. Emre Kiciman and Armando Fox. Using dynamic mediation to integrate COTS entities in a ubiquitous computing environment. In *HUC 2000*, pages 211–226, Bristol, UK. Springer-Verlag.
16. Tim Kindberg, John Barton, Jeff Morgan, Gene Becker, Debbie Caswell, Philippe Debaty, Gita Gopal, Marcos Frid, Venky Krishnan, Howard Morris, John Schettino, and Bill Se. People, places, things: Web presence for the real world. In *WMCSA 2000*, pages 19–28, Monterey, California. IEEE Computer Society Press.
17. Florian Michahelles, Michael Samulowitz, and Bernt Schiele. Detecting context in distributed sensor networks by using smart context-aware packets. In *ARCS 2002*, Karlsruhe, Germany. Springer-Verlag.
18. Kazuhiro Minami and David Kotz. Controlling access to pervasive information in the "Solar" system. Technical Report TR2002-422, Dept. of Computer Science, Dartmouth College, February 2002. Submitted to *Pervasive 2002*.
19. Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *SOSP 1997*, pages 276–287, Saint-Malo, France. ACM Press.
20. Bill Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *WMCSA 1994*, pages 85–90, Santa Cruz, California. IEEE Computer Society Press.
21. William Noah Schilit. *A System Architecture for Context-Aware Mobile Computing*. PhD thesis, Columbia University, May 1995.
22. Alex C. Snoeren, Kenneth Conley, and David K. Gifford. Mesh based content routing using XML. In *SOSP 2001*, pages 160–173, Chateau Lake Louise, Canada. ACM Press.
23. Mike Spreitzer and Marvin Theimer. Providing location information in a ubiquitous computing environment. In *SOSP 1993*, pages 270–283, Asheville, NC. ACM Press.
24. Amin Vahdat, Michael Dahlin, Thomas Anderson, and Amit Aggarwal. Active Names: Flexible location and transport of wide-area resources. In *USITS 1999*, Boulder, Colorado. USENIX.