Supporting Adaptive Ubiquitous Applications with the SOLAR System

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

Dartmouth College Computer Science
Technical Report TR2001-397

May 31, 2001

Abstract

As we embed more computers into our daily environment, ubiquitous computing promises to make them less noticeable and help to prevent information overload. We see, however, few ubiquitous applications that are able to adapt to the dynamics of user, physical, and computational context. We believe that there are two challenges causing this lack of ubiquitous applications: there is no flexible and scalable way to support information collection and dissemination in a ubiquitous and mobile environment, and there is no general approach to building adaptive applications given heterogeneous contextual information. We propose a system infrastructure, SOLAR, to meet these challenges. SOLAR uses a subscription-based *operator graph* abstraction and allows dynamic composition of stackable *operators* to manage ubiquitous information sources. After developing a set of diverse adaptive applications, we expect to identify fundamental techniques for context-aware adaptation. Our expectation is that SOLAR's end-to-end support for information *collection*, *dissemination*, and *utilization* will make it easy to build adaptive applications for a ubiquitous mobile environment with many users and devices.

1 Introduction

Ubiquitous (or "pervasive") computing has the vision to enhance computer use by making many computers available throughout the physical environment, but making them effectively invisible to the user [14, 29, 39]. To do so, ubiquitous applications need to adapt to current *context*, defined as the situation of user, physical environment, and computational state. Context is derived from an array of diverse information sources, such as location sensors, weather or traffic sensors, computer-network monitors, the status of computational or human services, and so forth. We have seen, however, no flexible and scalable information-collection and information-dissemination support for ubiquitous applications. Cohen et al. [12] provide an excellent summary of the challenges faced by context-sensitive applications.

Traditional applications are not designed to cope with context changes and a large number of users and devices. While the computational devices and network connections are becoming ubiquitous, adaptive applications are not. The few recent context-aware applications use ad-hoc methods to handle heterogeneous context. Furthermore, the misinterpretation of often-unreliable contextual information may lead to inappropriate application behaviors. The lack of a general approach to use contextual information, we believe, is a key obstacle to building adaptive ubiquitous applications.

How should a context-aware computing environment best support the dissemination of the context information collected by thousands of diverse context sensors and needed by dozens of diverse applications running on thousands of devices? What is the general approach to best use this contextual information in ubiquitous applications? These are the challenges faced by every such system. Several research projects have investigated solutions to related problems, and we discuss them in Section 6 below. We believe that these approaches, however, provide partial solutions

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

without sufficient flexibility or scalability.

1.1 Challenges

It is particularly challenging to build ubiquitous applications that can flexibly collect current context and use it to dynamically adapt to changes in their mobile environment with a large number of users and devices.

Scalability. As more computers and sensors are embedded in our daily life, the information accessible to applications explodes dramatically. The proliferation of computational devices with wireless connectivity also increases the number of potential clients of ubiquitous applications. As a result, context collected from a large number of information sources must be shared with dozens of applications over thousands of devices.

Information quality. The quality of information in a ubiquitous environment, which comes from error-prone sensors and resource-constrained embedded devices, is not suitable for an application's direct use. Also, high-level context might only be derived by combining the results of many information sources. It is unreasonable to ask each individual application to handle this overhead by itself, which restricts greatly the flexible access and usage of the context. On the other hand, it is a challenge for the applications to exhibit appropriate behaviors without 100 percent assurance of the accuracy of derived context.

Physical mobility. In addition to network address changes and weak connectivity, physical mobility leads to changes in context. Location is often difficult to measure, adding unreliability to contextual information. It may be difficult for ubiquitous applications to adapt quickly and accurately to changes in context if the user or device is moving frequently or rapidly.

2 Our approach

We propose a system infrastructure, SOLAR[1], to meet the challenges of supporting end-to-end information collection, dissemination, and utilization for adaptive ubiquitous applications. SOLAR uses a subscription-based *operator graph* abstraction and allows dynamic composition of the stackable operators, which are located in a context-sensitive namespace together with information sources and services. We believe that this novel abstraction is flexible through module composition to derive refined contextual information, and is scalable through re-use of common sub-graphs in the operator graph. We also propose to find fundamental approaches for diverse adaptive applications to use heterogenous context sources, especially when that information is incomplete or only partially accurate.

[1] SOLAR is not an acronym.

2.1 Assumptions

We have chosen to investigate an event-oriented approach that allows dynamic composition of simple stackable modules. Information is collected, processed, and disseminated through a graph of components called *operators*. We make several assumptions to clarify our design space:

- We assume we can deploy most of the operators onto well-connected network nodes, which agree to run our execution environment. The information sources and applications, however, may reside on the network edge or even on mobile nodes, communicating with our infrastructure through low-bandwidth wireless connections.
- Since we treat all information sources as event publishers as we discuss below, we assume it is possible to wrap a sensor that only has a query interface with a proxy that publishes sensed information as an event. Events may be published periodically, or only when the sensor changes its state.
- Our infrastructure is built on top of the TCP protocol, providing reliable packet delivery.
- We assume there will be widely deployed indoor location-sensing systems with room-sized granularity in the near future, based on the proliferation of location-sensing technologies [4, 9, 18, 19, 32, 37, 41] and location models [25, 28, 38].
- The operators may have internal state. They are deterministic functions of their inputs and their state and do not have random behaviors. This assumption is necessary for correct reusability, as we see below.

2.2 Design goals

To tackle the challenges discussed in Section 1.1, SOLAR is designed to meet three key goals.

Flexibility. SOLAR applications must have flexible access to either low-level raw information or high-level refined context, and applications should also have flexible ways to adapt their behaviors to heterogenous contextual information.

Mobility. SOLAR should use location-dependent information sources and operators as the user travels across geographical space, and provide support for handling fast context updates and lossy information due to physical mobility.

Scalability. SOLAR must collect context from many information sources and disseminate it to numerous clients, without becoming a bottleneck in the information flow.

3 The SOLAR system

In this section, we discuss the details of SOLAR's approach for information *collection* and *dissemination* in a ubiquitous and mobile environment. Section 4 talks about our current prototype and implementation status. We present our plans for supporting information *utilization* in Section 5.

SOLAR represents context information as *events*. SOLAR sensors, which we call *sources*, each *publish* a stream of events. SOLAR applications *subscribe* to event streams that interest them, and react to arriving events to adapt to their changing environment. Few applications, however, want to work with the raw event streams published by sensors. Applications need to *filter*, *transform*, *merge*, or *aggregate* event streams. Consider these examples. Filtering: a sensor publishes the temperature every 10 seconds while one application needs alerts only when the reading exceeds 90 degrees. Transformation: a location sensor reports coordinates, but the application needs a symbolic value ("Lobby"). Merging: an active-map application that displays the current location of all employees merges the readings from all location sensors. Aggregation: a reminder application wants to know when a user's calendar indicates it is time for a meeting and the user is not located in the meeting room.

Earlier researchers have recognized the need for filtering, transforming, merging, and aggregating event streams [12, 13, 35]. The challenge in a system like SOLAR is to allow applications to define their own operations, to describe flexible compositions of operations, and to support many such applications with scalable performance. SOLAR provides flexibility by allowing applications to define and interconnect *operator* objects. SOLAR provides scalability by distributing these operators across hosts in the network and by sharing identical event streams across users and applications.

We begin our description of SOLAR in Section 3.1 by discussing our operator-graph abstraction. Section 3.2 presents the context-sensitive naming scheme used to identify the information sources and operators. Then we sketch our small language for building subscriptions in Section 3.3.

3.1 Operator Graph

In this section, we introduce SOLAR's concepts of events, event streams, operators, and operator graphs. Then we classify several types of commonly used operators and sketch an example operator graph for an office scenario. Finally, we discuss the subtle semantics of operator state and "one-time" subscription requests.

3.1.1 Basic concepts

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. An *event* is a typed set of typed attribute-value pairs. SOLAR implements events as objects; thus event objects belong to a class that defines their type. The sequence of events produced are an *event stream*, which is inherently unidirectional. Only one type of events may flow through a given event stream; thus, event streams are also typed. An event *publisher* produces an event stream, and an event *subscriber* consumes an event stream.

An *operator* is an object that subscribes to and processes (filters, transforms, merges, or aggregates) 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*. SOLAR guarantees that the graph is acyclic through its incremental construction (see Section 3.3).

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 deterministic functions of their input events. They only publish an event 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

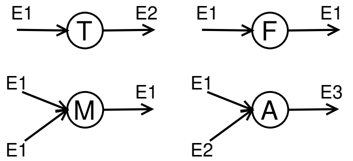

Figure 1: Four types of operators: T as Transformer, F as Filter, M as Merger, and A as Aggregator.

other stimuli, such as interactions with the user).

In our operator graph, a directed edge from node A to 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.

There are four common categories of operators (see Figure 1), related to the four kinds of event-processing functions we discussed earlier. A *filter* outputs a subset of its input events. A *transformer* inputs events of type E1 and outputs events of type E2. E2 may be the same as E1 if the transformer only changes some attribute values. The *merger* simply outputs every event it receives. While mergers are not strictly necessary, since any of the merger's subscribers could directly subscribe to the same inputs, we show below that a merger aids re-use of event streams. An *aggregator* outputs an arbitrary type event stream based on the events in one or more input event streams. 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.

3.1.2 An example operator graph

Figure 2 presents an example operator graph to show how the raw events from information sources flow through the operators to become directly usable by the applications. Circles represent event publishers; the letter inside indicates its category (S stands for source). 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 associated with it.

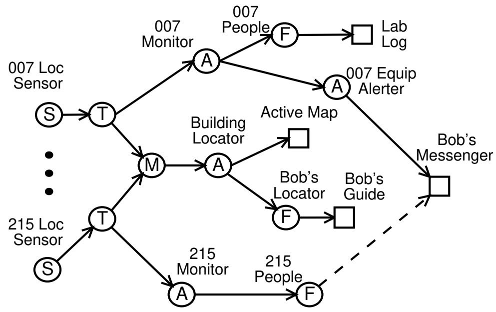

Figure 2: An example operator graph.

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. (We discuss stateful operators below.) 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 (such as [27]) to display the badges' current location in real time. Another subscriber, *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. Any application that cares about location events only in one particular room can filter the Building Locator's output. Although that approach seems awkward, it allows the Building Locator to resolve sensor conflicts (where multiple location sensors report seeing a badge at the same time).

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 how SOLAR supports such context-sensitive subscription requests in Section 3.2.

There are several advantages of the SOLAR approach: 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.

3.1.3 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. Filter, transformation, and merger operators are stateless; aggregators may have state.

SOLAR 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 [5].)

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.

3.1.4 One-time subscription requests

SOLAR is an event-oriented publish-and-subscribe system 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 SOLAR 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.

SOLAR's 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 subscriber's control flow remains event-oriented rather than blocking for the results of a query. 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.

3.2 Context-sensitive names

Although we can build operator graphs from the connections described by a subscribing application, and SOLAR has a small language for that purpose, it is frequently useful to name event publishers so that their streams are easily usable by many subscribers. By naming a publisher, applications can subscribe to its event stream without needing to describe that stream from first principles (sources). It is common to construct named mergers, for example, to make pre-defined combinations of event streams available to many applications.

In addition to information sources, applications also need to discover services. We expect that many of the same objects located in our information-dissemination graph (such as printers that publish their current print queue) may also provide a service (accepting documents to be printed). Thus, the SOLAR name space allows naming of both publishers and services.

A flat name space, such as that used to label publishers in Figure 2, does not scale. Furthermore, given the dynamic nature of an application's context in a pervasive-computing environment, we desire name bindings that change with the changing context. The SOLAR name space is thus hierarchical and dynamic.

3.2.1 Examples

Before we present the details of the name space, consider a few examples of both static names and context-sensitive names (CSNs).

A printer is located in room 007. The printer's name is [/devices/printers/23], a unique name assigned to that printer when it first entered the system. That name is static, assigned by the administrator. The room's name, also statically assigned, is [/places/0F/007], indicating that 007 is a room in the basement floor of the building. The set of devices in 007 are in the directory [/places/0F/007/devices/], which contains a context-sensitive list of devices currently in 007. Similarly, the context-sensitive name [/devices/printers/23/location] is an alias for [/places/0F/007].

A graduate student, Alice, wishes to print a document from her PDA to the nearby printer. Alice's name is [/people/students/Alice]. Her location, [/people/students/Alice/location], is a context-sensitive name. When Alice is in 007, that name is bound to the same object as [/places/0F/007]. Thus the printing application on her PDA can search directory [/people/students/Alice/location/devices/] for printers.

Applications can find services in the name space. The service for the printer [/devices/printers/23] is called [/devices/printers/23/PrinterService]. (In our implementation, services are named by convention after the Java class that implements the service.) Thus, Alice's PDA can find the service for all printers in the room as [/people/students/Alice/location/devices/*/PrinterService].

Context-sensitive names can be used in subscription requests. Consider Bob's Messenger in Figure 2, which requested a subscription to the context-sensitive name [/people/profs/Bob/location/people/]. As a result, this subscription request maps into a subscription to the node in the namespace at [/places/2F/215/people]. The figure shows this subscription as a dashed line to indicate that it is the result of a context-sensitive subscription request, which will change when the name binding changes.

Here it is important to note the difference between a *subscription request*, which refers to a (possibly context-sensitive) name, and a *subscription*, which is an actual connection between a publisher and a subscriber. Subscription requests map a changing set of zero or more subscriptions, as we see below.

The SOLAR approach is different from that in most other pervasive-computing systems. Rather than retrieving Alice's location from a location service, and then using that as a key to search a name space for printers in that location, both concepts are included in the same namespace. "Printers sharing a room with Alice" thus has a single concise name, a name whose binding changes when Alice's room changes.

3.2.2 The naming tree

The name space is organized as a tree of labelled nodes, with the addition of cross edges (see an example in Figure 3). The *name* for a node is the sequence of labels encountered on any path from the root to the node, separated by slashes. Many nodes have an associated list of attribute-value pairs describing the object represented by that node. For example, the printer [/devices/printers/23] may be described as (category=printer, color=true, laser=true, maker=xerox).

There are several types of nodes. Some of the leaves are *alias nodes* that are representatives of another node. By analogy to Unix, we call these cross edges "soft links" and depict them with dashed lines in the figure. Some leaves may be *service nodes*, depicted as squares, which represent available services. Other leaves are the event publishers we discussed in the operator graph.

Internal nodes are *directories* that refer to several child nodes. *Static* directory nodes contain a list of children, a list updated only by explicit requests. *Dynamic* directory nodes generate a list of children dynamically, from internal state that depends on contextual information obtained from subscriptions. Typically the children of dynamic directories are nodes already located elsewhere in the name space, another form of soft link that we depict with dash-dot-dot lines.

Soft links are dynamically generated by a directory or alias node when needed. These directory and alias nodes must, therefore, be operators with appropriate subscriptions and sufficient state to be able to generate the appropriate list of children when asked. For example, the node [/places/2F/215/people] lists Bob as child and [/people/profs/Bob/location] refers to room 215. These links are automatically updated as the user moves across geographical spaces. Since both operators derive the location information from the same source [/places/2F/215/loc-sensor] (refer to the operator graph in Figure 2) these two views will remain consistent.

Nodes in the name space are also publishers of changes to name bindings. More precisely, direc-

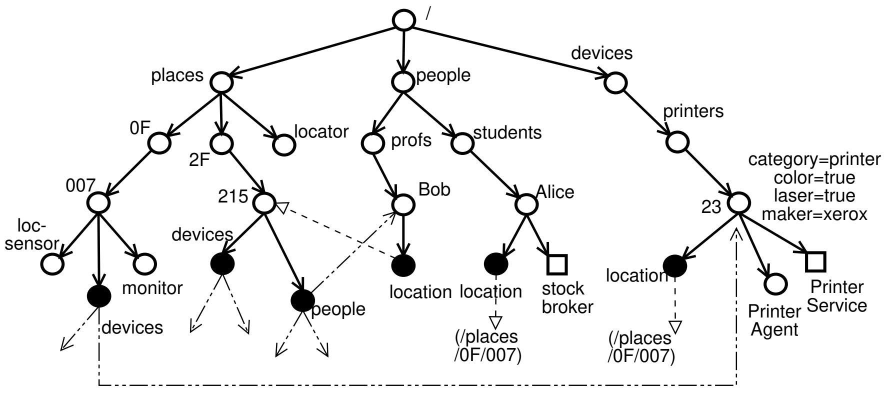

Figure 3: A partial context-sensitive naming tree. Dashed lines represent alias soft links, and dash-dot-dot lines represent child soft links.

tory nodes are publishers that announce additions or deletions of their children by publishing events. (Static directory nodes are sources and dynamic directory nodes are operators.) Alias nodes are publishers that announce changes in their bindings. Interested applications can subscribe to these sources to detect changes in the name space. So if Alice wants to track the location of her advisor Bob, her application subscribes to the operator at [/people/profs/Bob/location], the node identified as "Bob's Locator" in Figure 2.

3.2.3 Name resolution

An application can locate services or publishers by using the Solar name syntax. In this section we show how Solar uses wildcards, attribute matching, and soft links to provide for dynamic discovery.

We support two types of wildcards. A "⋆" can represent any node in one level of the naming tree, while a "..." symbol represent any number of levels in the naming tree. So the name for all the location-sensor sources on ground floor is [/places/0F/⋆/loc-sensor] while the name for all the location sensors in the building is [/places/.../loc-sensor]. This latter form, while equivalent to [/places/⋆/⋆/loc-sensor], is useful if the application is not exactly sure about the naming tree structure.

A name can contain a *selector* to identify names based on attributes. For example, the services for the color printers at Alice's current location can be found using the name [/people/students/Alice/location/devices/⋆(color=true)/PrinterService].

Solar name resolution starts from the root and traverses its children recursively by matching the label with the symbol at the next level in the name. At an alias node, the algorithm continues resolution from the target of the alias. When the algorithm encounters attribute selectors, it checks for matches in the attributes of the nodes. If the wildcard "..." appears in the name, the entire subtree is searched for matches; the algorithm limits the number of soft-link traversals to avoid infinite loops.

3.2.4 Combining the naming tree with the operator graph

Note that the Naming Tree and the Operator Graph are interconnected data structures, containing many of the same nodes. As described above, most nodes in the name tree are event publishers and thus appear in the Operator Graph: directory nodes publish changes to the set of children, alias nodes publish changes in their binding, and of course publisher leaves are publishers. Only service nodes are not publishers. The availability of events about changes to the name space is used by Solar to support context-sensitive subscription requests; we present the details in Section 4.1.2.

Not all publishers in the Operator Graph are in the Naming Tree, however. These unnamed publishers are often created by applications as intermediate filters and transformations with no need to be named in the name space.

In summary, we achieve one abstraction for two concepts. Figure 3 uses filled black nodes to identify those nodes that also appear in Figure 2. The directory node with name [/places/2F/215/people] is the operator labeled "215 People" in Figure 2. As people or devices enter or leave room 215, this operator publishes events as updates to the node's list of children. The alias node [/people/profs/Bob/location] is the operator labeled "Bob's Locator" in Figure 2.

Changes in Bob's location are reflected as events about the change in the binding of that alias.

Note carefully that the links drawn in the Naming Tree represent the children of directories, or the targets of aliases, whereas the links drawn in the Operator Graph represent subscriptions.

3.3 A subscription language

In the preceding two subsections we described the SOLAR operator graph and the SOLAR approach to naming. In this section we show how an application can request a subscription to the desired stream of events, and how that request can grow the operator graph.

In the simplest case, the application wishes to subscribe to a stream of events already published by an existing source or operator, and that publisher is visible in the namespace. Thus, the application makes a "subscription request" by providing the name. If the name includes wild-card characters, multiple subscriptions may result from the subscription request. If the name is context-sensitive, the subscription(s) may change over time.

In many cases, an application desires an event stream that is not already published. The application must specify the operator tree that merges, filters, transforms, and aggregates existing event streams into the desired stream of events. The events published by the tree's root flow into the application. The events flowing into the leaves of the tree are the result of subscription requests to publishers in the name space. The internal nodes of the tree are operators identified by the application.

SOLAR has a small language for applications to describe their subscription tree. The language constrains the structure of the subscription requests to a tree. By ensuring that the structure is a tree and its output is to a non-publishing node (the application), the overall operator graph remains acyclic. The language allows, but does not require, the programmer to specify a new name for each of the operators in the tree. Thus, some operators may be unnamed.

A key goal of our approach is to allow re-use of event streams in the operator graph. Re-use keeps the operator graph small, avoids extraneous computation and network traffic, and is critical for overall scalability. First, we allow applications to "export" published streams for public use by registering the operator in the name space. Second, we automatically match new subscription trees against the existing subscription graph, so that the new subscription tree can re-use any existing common subtree.

With deterministic, stateless operators, it is easy for SOLAR to detect opportunities for this implicit re-use: if a newcomer wants to subscribe to the event stream produced by an operator from a set of event streams, and another subscriber has already instantiated the same operator on the same input streams, then the newcomer need only subscribe to that existing operator's published output. If the operators have state, however, their output depends on their state as well as their inputs. For now, SOLAR does not implicitly re-use stateful operators. We plan to investigate conditions that may allow re-use of stateful aggregators.

4 Current prototype

In this section, we present our current design, and part of the SOLAR system architecture, for the operator graph abstraction and context-sensitive namespace. The SOLAR system is still evolving and the discussion here is based on a partial prototype implementation.

4.1 System architecture

The overall architecture is shown in Figure 4. At the center of any SOLAR system is a *Star*, which keeps a reference to the root of the naming tree, maintains the operator graph, and services requests for new subscriptions. When the Star receives a new subscription-tree description, it parses the description, checks the name space for imported names, updates the name space with exported names, and matches the subscription tree 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, on any platform. They use a small SOLAR library that allows them to send requests to the Star, and to manage their subscriptions, over standard network protocols.

4.1.1 The Star

The Star of the SOLAR system has two major components: an OPerator Space (OPS), and a Subscription Engine (SE), as shown in Figure 4.

The OPerator Space (OPS) is a data structure that loosely represents the current operator graph. The OPS tracks operators and their subscription requests, rather than the actual subscriptions, for two reasons. First, context-sensitive subscription requests

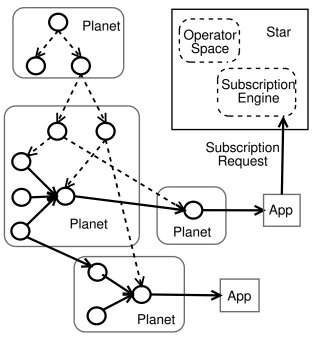

Figure 4: The architecture of the SOLAR system. The dashed arrows are tree edges in the name space while the solid arrows represent subscription links in operator graph.

can cause rapid changes to the underlying subscriptions, and it would be too expensive to track all the changes. Second, the purpose of the OPS is to match new subscription requests against existing subscription requests.

The subscription engine (SE) parses the subscription-tree descriptions received from applications, and builds the tree of subscription requests. It locates named publishers using the name resolution algorithm, and matches the tree against the OPS to identify opportunities for re-use. If necessary, it deploys new operators, and informs the Planets of the subscription requests. It records the new operators and subscription requests in OPS. Finally, it updates the name space with any names exported by this subscription description.

It is challenging to choose appropriate Planets to deploy operators. Our current prototype chooses a Planet at random. In the near future we plan to develop a deployment algorithm that attempts to map the operator graph to the Planets to balance and minimize the network traffic. Ideally, the SE would also consider the topology of the network and the computational load imposed by the new operator.

4.1.2 The Planet

A Planet is an execution environment for operators. (In our implementation, operators are Java objects and Planets are based on Java virtual machines.)

A Planet is identified by its host IP address and port number. Within a Planet operators are distinguished by a unique identification number. When the SE deploys a new operator, the Planet acknowledges the deployment and includes the identification number. Thus the SE can record in OPS the new operator's address as a triple: IP address, port number, and intra-Planetary id number.

The Planets play a key role in the subscriptions of resident operators. (For applications, the SOLAR library serves in this role.) When the SE asks a Planet to subscribe one of its resident operators to another operator, it specifies the subscriber's and publisher's address. The Planet contacts the publisher's Planet and asks it to add a new subscriber's address to the data structure for that publisher. When the publisher produces a new event, the event is sent to all subscribers in the list. More precisely, a copy of the event is sent to each *Planet* on that subscription list. Thus, if several operators on Planet X subscribe to the same publisher at Planet Y, only one copy of the event is delivered from Planet Y to Planet X. Along with the event is a list of the operator's id numbers, so that Planet X can dispatch the event to each operator.

So, every event flows from the publisher to the destination Planet, and is then dispatched to the subscriber(s) at that Planet. The Planet maintains a queue of input events for each resident operator, and a queue of output events for each remote Planet. With care, a Planet may avoid duplicating arriving events, placing a read-only reference to the event into each input queue.

A key advantage to this structure is that there need to be only one network (TCP/IP) connection between any two Planets, regardless of the number of operators resident on each Planet or the number of subscriptions between the two Planets. When there are no open subscriptions between two Planets, the Planets may close the connection at their discretion, and re-open it when needed.

Another critical advantage of this structure is that the Planet supports subscription requests that involve context-sensitive names (CSNs). These subscription requests are mapped to subscriptions, which need to be changed when the CSN binding changes. Consider an operator that records the name of every person Bob meets. The operator requests subscription to the CSN [/people/profs/Bob/location/people]. The operator's Planet subscribes to the name [/people/profs/Bob/location]. In the state push, the Planet receives the current binding and subscribes the operator to the current publisher, e.g., [/places/2F/215/people]. When the binding changes, the Planet contacts the old publisher's Planet to remove the operator's address from the old publisher's data structure, and contacts the new publisher's Planet to add the operator's address to the new publisher's data structure.

If a subscription request contains the $\star$ wild card, the list of matches may change. The subscriber's Planet initially arranges a subscription to each matching publisher, and then monitors the directory node above each $\star$ for changes. On any change, the Planet updates the set of subscriptions.

If a subscription request involves both wildcards and alias nodes, a change deep in the name space can affect many subscriptions. Planets subscribe to each node that may notice the change: the immediate parent of wildcards, and each alias node. Whenever such a node changes its bindings, it sends an event to its direct subscribers. It also invokes a method of its parent in the naming tree, recursively climbing the tree from the point of change up to the root; each node emits an event indicating the change in their subtree. This approach ensures that the Planets hear about all relevant changes that may affect the mapping of subscription requests.

Finally, Planets are responsible for garbage collection of unnamed operators. They monitor resident operators for lack of subscriptions. Planets may eliminate operators with no subscriptions, except for naming nodes, and notify the OPS.

4.2 Status

So far, we have implemented a simplified version of the Star, which deploys operators to specified Planets and instruments the subscriptions based on an input XML file. The subscription language and namespace are still under development. The Planets are functional in delivering events across the network and dispatching received events to resident operators. The Planet is not optimized, however; for instance, it sends multiple copies of same event to the subscribers residing on one Planet. We are using this partial prototype in a location-sensitive appointment-reminder service, developed by Arun Mathias [26].

5 Planned research

Solar is a work in progress, and we plan more research. Here we outline other research issues about information collection and dissemination for ubiquitous applications in Section 5.1, and on context utilization in Section 5.2.

5.1 Context collection and dissemination

We plan to continue our research on the operator graph abstraction, including the design of a language to express operators, some directions to refine our design, and exploration of code mobility in Solar.

5.1.1 Operator state and language

Currently Solar does not provide any specific support for state push operations. The reason is that we allow arbitrary Java code for the operator and Solar lacks the insight of the operator's state transitions. It is possible for Solar to archive the events an operator has published and replay them when a state push is needed. This brute-force solution is, however, extremely inefficient and a more favorable approach, like generating a most economical sequence of events by finding a shortest path in the state-transition graph [5], is desired. One approach is to provide a simple *operator language* for the programmers. With access to the source code of operators, Solar is able to build the state-transition graph and know how to generate a more economical sequence of state-push events.

One-time subscriptions require a state push. What is the correct reaction when a stateless operator receives a one-time subscription? Solar can leave it to the programmer, replay the most recent event the operator has published, or recursively send one-time subscriptions to upstream operators to get the latest state. Another issue is related to implicit re-use of the stateful operators. Is a state-push operation necessary when there is no explicit indication from the subscriber? We believe these issues need to be further investigated and a formalization of the operator state will help us to understand the correct semantics.

5.1.2 Code mobility

Code mobility [15] is an innovative approach to developing distributed applications [17] in a large scale and mobile environment [16, 23]. We recognize three variations of this technique: mobile code, mobile objects, and mobile agents.

The *mobile code* approach is to simply move a code fragment, which has not been instantiated, to a destination host and initialize and execute it there. The code hops once and no state information needs to be transferred. The *mobile object* approach may move an already instantiated object from one host to another while the object state is preserved during the transition. The object is not autonomous and does not have its own execution thread. The decision to move is made externally. A *mobile agent*, instead, is self-contained and can choose when and where to move. Either the execution stack state (strong mobility) or only the agent's data state (weak mobility) is captured and restored as the agent travels.

We already use mobile code in the current design of Solar. The operators deployed by the Star are dynamically loaded to destination Planets from code servers. This technique gives flexibility of code distribution without the need to install all operator code at every Planet prior to the deployment. Applications not in a Planet execution environment can simply download a SOLAR *proxy* from the Star, which hides all the details of communication protocols to send/receive events, flow control mechanisms, and handling of system events. This approach is analogous to Jini's service proxy and promotes great flexibility to the system evolution.

Mobile objects may have potential in SOLAR, but there are several research issues: 1) what benefits can we get by moving objects and under what conditions? 2) what factors help decide when and where should which objects move? 3) what is the cost of moving an object? and 4) how to store events that arrive during the transition?

5.1.3 Other research issues

The namespace grows as the system extends to wider area and one name tree can not handle mass-scale name queries. If we federate individual namespaces into several hierarchical levels, we need a protocol to traverse namespaces for global name resolution. The Star is a central point of failure and a possible performance bottleneck in systems where new subscriptions occur frequently. A distributed set of Stars is needed. If a Planet or host crashes, a network fails, or a host disconnects from the network, SOLAR must adapt gracefully. We need a mechanism for flow control between publishers and subscribers to prevent traffic jams in operator graph. Operators that are no longer in use need to be garbage collected. SOLAR's current context-sensitive namespace only encodes location context, through which applications can discover more location-dependent context. Could we include other types of context in the namespace? Finally, SOLAR must protect sensitive user context from tampering and unauthorized access. For instance, Planets need to execute untrusted operator code securely, Planets must limit the resource usage, and SOLAR must limit subscription to event streams according to an access policy.

5.2 Context utilization

In the preceding sections we outline the SOLAR approach for collecting and disseminating contextual information. In this section we discuss the *use* of context and the development of adaptive ubiquitous applications.

Part of SOLAR's goal is to provide a flexible and scalable approach to collect context and disseminate it to applications, which then adjust their behaviors accordingly. Adaptive applications, however, can not assume that the derived context is always complete and correct. An incorrect assumption may lead to undesired behavior and users will be reluctant to use the application again. There are several reasons for imperfect context:

- the sensor itself is not designed to be reliable and may produce random errors,
- a noisy environment may degrade the accuracy of sensor readings a lot,
- information may be lost in converting from sensor data to an application-desired format, such as numeric value to symbolic value or vice versa, and
- fast sensor update rates and physical mobility may cause a slow context meditation algorithm to return an outdated value.

Many existing context-sensitive applications [10] either assume the context is 100 percent correct or deal with the imperfection in an ad-hoc way. Several approaches for adaptive applications coping with unreliable context are:

- Each sensor value is assigned with a confidence level and a corroboration function can be used to calculate the combined value from several sensor sources for the same type of context (such as location) [33].
- The system profiles sensor inputs for a set of context situations during a training phase, and it generates a probability distribution across these situations during execution time according to current sensor values [9, 35]. Like the previous case, applications are responsible for interpreting the result.
- Another approach is to ask for user's explicit help to distinguish ambiguous situations [13], and the feedback is used to adjust an internal model and help future recognition.

In general it is hard to find a universal approach for all adaptive applications that use heterogenous contextual information. We identify several key research issues for a general approach to context adaptation.

- **Context representation.** Context has been represented internally using key-value pairs, tagged strings, and as an object [10]. Which one helps an application's understanding and adaptation most? Is a uniform representation possible and necessary for context adaptation?

- **Application structure.** It is crucial to determine how applications must be structured to take benefit of context. Shall the applications be redesigned with adaptiveness in mind or just to isolate adaptiveness in a few modules [40]?

- **Multi-level adaptation.** We believe that adaptive applications should be defensive/conservative, instead of aggressive, when making adaptation decisions because users will be reluctant to use the applications again when she gets the impression that they make mistakes. Applications need to have multi-level adaptations when they make choice given current context. Is this abstraction generalizable to all applications, and what is the appropriate programming model for the developers?

6 Related work

In this section we discuss related work in the following areas: smart spaces and environments, event-based middleware and dynamic composition, context-aware computing, naming systems, application partitioning, and context fusion. Here we list the work most closely related to SOLAR.

6.1 Smart spaces and environments

Of the research on smart spaces and intelligent environments, two research projects have made a focus of the information-dissemination problem.

The proposed DataSpace system [21] uses a query-based approach. The system sends queries to information sources that live in spatially partitioned "datacubes" by mapping the queries to multicast addresses. The system returns a summary of the results to the application, which can then "zoom in" to extract detailed results from the system.

The Cougar system at Cornell [7] is designed to support queries over millions of sensor devices. Cougar models each information source as a database. One research focus is to decompose SQL queries and execute them on the distributed sensor devices whenever possible.

Both systems use queries to obtain raw information from sources, but do not allow the application to construct any application-specific post-processing modules. As such, there is no mechanism to develop shared information derivatives as in SOLAR.

6.2 Event-based middleware and dynamic composition

SIENA [8] is a large-scale content-based event distribution service. Events are filtered and delivered, but not transformed or aggregated. Unlike SOLAR, SIENA applications do not subscribe to a specific publisher. Applications describe the desired type of events as a pattern, and publishers describe the type of events they will publish as a pattern. SIENA uses the patterns to establish routing paths for events to follow through the broker network, with a goal of minimizing network traffic.

In the Gryphon system [5] events flow through an Information Flow Graph (IFG). The IFG is somewhat like a dual of a SOLAR operator graph, except that their nodes are "information spaces" and the edges represent operations to derive one space from another. Operations include filters, transforms, a "collapse" operation to create a "state space" from a stream of events, and an "expand" operation to create a stream of events from a state space. Gryphon has a routing algorithm to distribute events from sources to applications through a network of Gryphon brokers. Gryphon appears to be intended for static flow graphs, however, because they do not indicate how to handle changes to the graphs.

In the Cambridge Event Architecture (CEA) [3] clients can subscribe to event publishers. Or, clients can ask an "event mediator" to help them subscribe to publishers that produce events matching a pattern supplied by the client. Or, clients can ask the "composite event service" to aggregate events produced by event sources and inform them when a certain pattern is matched. Unlike in SOLAR, the intermediate results involved in recognizing a composite event are not exposed and thus cannot be shared by others.

Kiciman and Fox [22] also propose a solution using dynamic composition to address a different problem, which is the protocol and data-format mismatches between legacy hardware and software or incompatible COTS. They form a communication path by dynamically inserting appropriate mediators between any two components that need to communicate. The mediators along the path are stateless and function as data transformers. A more general composition framework called Path has operators and connectors. They aim to select and deploy the necessary mediators automatically.

6.3 Context-aware systems

As in smart spaces, context-aware applications adapt their behaviors to their context, as obtained from pervasive information sources.

In many systems, a monolithic module transforms the raw sensor information into high-level context, which is then available to applications [34, 36]. They provide no support for applications to derive new refined information, in ways that were not expected by the information providers.

The Context Toolkit [13] is a distributed system that encapsulates each sensor as a *widget* with a service component. Applications may subscribe to sensor data produced by widgets. Applications may deploy "aggregators" to combine and transform data from widgets. It is not clear whether aggregators can be stacked.

The Context Information Service (CIS) is a proposed architecture [30] that represents the world as a set of objects (people and devices). The state of each object is a combination of several values derived from sensors (such as location, availability, temperature). The service updates object states by monitoring sensors or synthesizers. "Synthesizer" modules aggregate data from sensors to produce high-level information. Applications obtain the states of a set of objects from existing monitors or by deploying new monitors from a catalog.

Taking the service composition [22] idea further, Context Fabric [20] also proposes to automate the path creation from low-level sensor data to high-level context data. The high-level automation relieves the programmers from dealing with specific sensors. The numerous types of context data, however, make it hard for the system designer to build all possible operators for the applications that may potentially use unexpected contextual information. Also, it is unclear how the event-oriented service can be composed through these operators without subscription interfaces.

6.4 Context fusion

To handle unreliable contextual information, some fusion and aggregation should be applied on raw sensor data first. There are several approaches to this difficult problem.

Rizzo et al. [33] assign every sensor reading a confidence level and a corroboration function is used to calculate the combined value from several sensor sources for the same context. It is unclear, however, which confidence level should be assigned for the sensor data at the first place, and the corroboration function seems to be dependent on the type of context to be retrieved.

Nibble [9] and TEA [35] profile sensor inputs for a set of predefined locations or context situations during the training phase, and generate a probability distribution across these situations during execution time according to current sensor values. Like previous approaches, applications are responsible for interpreting the result.

Another approach is to ask for the user's explicit help to distinguish the ambiguous situations [13], and the feedback is used to adjust the internal model and aid future recognition. It is not always possible, however, that the user is available for help and users may feel the applications are intrusive and become reluctant to use them.

6.5 Naming systems

Solar allows publishers and services to be identified by context-sensitive names (CSN). Most other distributed naming systems, like Grapevine [6] and the Global Naming Service [24], assume name bindings are mostly static. Unix "symbolic links" allow for aliasing as in Solar, but they are largely static, and there is no automatic way to learn about binding changes.

Plan 9 [31] uses a file-system interface to represent a wide range of system resources in addition to data files. Like our dynamic directory nodes, the Plan 9 name space has special "directories" that do not contain normal files. Their contents are synthesized on demand when read or rewritten.

The Intentional Naming System (INS) [2] names each resource and service with a set of attribute-value pairs that describes the named object. An object periodically sends its current name to INS, causing any change in the description to be reflected in the name space. A late-binding mechanism that integrates name resolution and message routing makes it possible for a client to continuously communicate with a service object that is moving to a new network location. They do not specifically address the potential for their naming system to encode context.

7 Summary

Large-scale systems that support ubiquitous applications need flexible and scalable information-dissemination mechanisms to convey and transform contextual information to applications, and such systems also need to provide a general approach for context adaptation to be used by diverse adaptive applications.

In our Solar system, we treat information sources as event producers and applications as event consumers. Thus, sources are publishers, and applications are subscribers. We introduce operators, which subscribe to one or more event streams and produce a new event stream, either by merging, filtering, transforming, or aggregating their input stream(s). We encourage the dynamic composition of many such operators into a graph that connects many sources to many applications. This graph serves to transform the raw events produced by the sources into higher-level contextual information usable by applications.

Solar names many publishers in a hierarchical name space. The real value of the name space is in the use of dynamic directory nodes, which are operators that produce a list of child nodes based on the contextual information in their own subscriptions, and in the use of alias nodes, which are operators that allow context-sensitive aliases for objects. The combination provides a constant name for a context-sensitive concept, such as the list of people in the room with Bob: [/people/prof/bob/location/people], which changes when Bob moves or when people join or leave him. The result is an easy way for many applications to express their sensitivity to context simply by requesting a subscription to an information source through a context-sensitive name.

Solar has a small language to allow application programmers to express the information it wants in the form of a tree of operators to transform events from named publishers into events for the application. Solar parses the subscription description to dynamically deploy operators as needed, re-using existing operators where possible.

The heart of the Solar system is the Star, which processes new subscription descriptions. The Star deploys all sources, operators, and applications on a distributed network of execution environments, Planets. All events flow between Planets, directly from the publisher's Planet to the subscribers' Planets. If the operators are deployed well across Planets, this infrastructure avoids any unnecessary duplication of events on the network, and allows Planets to adjust context-sensitive subscription requests when the name bindings change.

We discuss several other important research issues surrounding the design of Solar. Ultimately, we expect that Solar's end-to-end support for information collection, dissemination, and utilization will make it easy to build adaptive applications for a ubiquitous mobile environment with many users and devices.

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] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 186–201, Kiawah Island Resort, South Carolina, December 1999. ACM Press.

[3] Jean Bacon, Ken Moody, John Bates, Richard Hayton, Chaoying Ma, Andrew McNeil, Oliver Seidel, and Mark Spiteri. Generic support for distributed applications. *IEEE Computer*, 33(3), March 2000.

[4] Paramvir Bahl and Venkata N. Padmanabhan. Radar: An in-building RF-based user location and tracking system. In *Proceedings of IEEE INFOCOM 2000*, Tel-Aviv, Israel, March 2000. IEEE Computer Society Press.

[5] Guruduth Banavar, Marc Kaplan, Kelly Shaw, Robert E. Strom, Daniel C. Sturman, and Wei Tao. Information flow based event distribution middleware. In *Proceedings of the Middleware Workshop at the 19th IEEE International Conference on Distributed Computing Systems*, Austin, Texas, May 1999. IEEE Computer Society Press.

[6] Andrew D. Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder. Grapevine: An exercise in distributed computing. *Communication of ACM*, 25(4):260–274, April 1982.

[7] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Querying the physical world. *IEEE Personal Communications*, 7(5):10–15, October 2000.

[8] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving scalability and expressiveness in an Internet-scale event notification service. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 219–227, Portland OR, USA, July 2000.

[9] Paul Castro, Patrick Chiu, and Richard Muntz. A probabilistic room location system for wireless network environment. Submitted for publication, April 2001.

[10] Guanling Chen and David Kotz. A survey of context-aware mobile computing research. Technical Report TR2000-381, Dept. of Computer Science, Dartmouth College, November 2000.

[11] Keith Cheverst, Nigel Davies, Keith Mitchell, and Adrian Friday. Experiences of developing and deploying a context-aware tourist guide: The GUIDE project. In *Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking*, pages 20–31, Boston, MA, August 2000. ACM Press.

[12] Norman H. Cohen, Apratim Purakayastha, John Turek, Luke Wong, and Danny Yeh. Challenges in flexible aggregation of pervasive data. Technical Report RC21942, IBM Research Division, Thomas J. Watson Research Center, P.O.Box 704, Yorktown Heights, NY 10598, January 2001.

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

[14] Mike Esler, Jeffrey Hightower, Tom Anderson, and Gaetano Borriello. Next century challenges: Data-centric networking for invisible computing. In *Proceedings of the Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 256–262, Seattle, WA, August 1999. ACM Press.

[15] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[16] Robert S. Gray, George Cybenko, David Kotz, and Daniela Rus. Mobile agents: Motivations and state of the art. In Jeffrey Bradshaw, editor, *Handbook of Agent Technology*. AAAI/MIT Press, 2001.

[17] David Halls. *Applying Mobile Code to Distributed Systems*. PhD thesis, Computer Laboratory, University of Cambridge, June 1997.

[18] Andy Harter, Andy Hopper, Pete Steggles, Andy Ward, and Paul Webster. The anatomy of a context-aware application. In *Proceedings of the Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 59–68, Seattle, WA, August 1999. ACM Press.

[19] Jeffrey Hightower, Roy Want, and Gaetano Borriello. SpotON: An indoor 3D location sensing technology based on RF signal strength. UW CSE 2000-02-02, University of Washington, Seattle, WA, February 2000.

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

[21] Tomasz Imielinski and Samir Goel. DataSpace: Querying and monitoring deeply networked collections in physical space. *IEEE Personal Communications*, 7(5):4–9, October 2000.

[22] Emre Kiciman and Armando Fox. Using dynamic mediation to integrate COTS entities in a ubiquitous computing environment. In *Proceedings of Second International Symposium on Handheld and Ubiquitous Computing*, pages 211–226, Bristol, UK, September 2000. Springer Verlag.

[23] David Kotz, Robert Gray, Saurab Nog, Daniela Rus, Sumit Chawla, and George Cybenko. Agent Tcl: Targeting the needs of mobile computers. *IEEE Internet Computing*, 1(4):58–67, July/August 1997.

[24] Butler W. Lampson. Designing a global name service. In *Proceedings of the 4th ACM Symposium on Principles of Distributed Computing*, pages 1–10, Minaki, Ontario, 1986. ACM Press.

[25] Ulf Leonhardt. *Supporting Location-Awareness in Open Distributed Systems*. PhD thesis, Imperial College of Science, Technology and Medicine, University of London, May 1998.

[26] Arun Mathias. SmartReminder: A case study on context-sensitive applications. Technical Report TR2001-392, Dartmouth College, Hanover, NH 03755, June 2001.

[27] Joseph F. McCarthy and Eric S. Meidel. ACTIVE MAP: A visualization tool for location awareness to support informal interactions. In *Proceedings of First International Symposium on Handheld and Ubiquitous Computing*, pages 158–170, Karlsruhe, Germany, September 1999. Springer Verlag.

[28] Giles John Nelson. *Context-Aware and Location Systems*. PhD thesis, Clare College, University of Cambridge, January 1998.

[29] Donald A. Norman. *The Invisible Computer*. The MIT Press, 1999.

[30] Jason Pascoe. Adding generic contextual capabilities to wearable computers. In *Proceedings of the Second International Symposium on Wearable Computers*, Pittsburgh, Pennsylvania, October 1998. IEEE Computer Society Press.

[31] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The use of name spaces in Plan 9. *Operating System Review*, 27(2):72–76, April 1993.

[32] Nissanka B. Priyantha, Anit Chakraborty, and Hari Balakrishnan. The Cricket location-support system. In *Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking*, pages 32–43, Boston, MA, August 2000. ACM Press.

[33] Mike Rizzo, Peter F. Linington, and Ian A. Utting. Integration of location services in the open distributed office. Technical Report 14-94, Computing Laboratory, University of Kent, Canterbury, UK, August 1994.

[34] William Noah Schilit. *A System Architecture for Context-Aware Mobile Computing*. PhD thesis, Columbia University, May 1995.

[35] Albrecht Schmidt, Kofi Asante Aidoo, Antti Takaluoma, Urpo Tuomela, Kristof Van Laerhoven, and Walter Van de Velde. Advanced interaction in context. In *Proceedings of First International Symposium on Handheld and Ubiquitous Computing*, pages 89–101, Karlsruhe, Germany, September 1999. Springer Verlag.

[36] Mike Spreitzer and Marvin Theimer. Providing location information in a ubiquitous computing environment. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 270–283, Asheville, NC, December 1993. ACM Press.

[37] Roy Want, Andy Hopper, Veronica Falcão, and Jonathan Gibbons. The Active Badge location system. *ACM Transactions on Information Systems*, 10(1):91–102, January 1992.

[38] Andrew Martin Robert Ward. *Sensor-driven Computing*. PhD thesis, Corpus Christi College, University of Cambridge, May 1999.

[39] Mark Weiser. The computer for the 21st century. *Scientific American*, pages 94–104, September 1991.

[40] Girish Welling and B.R. Badrinath. An architecture for exporting environment awareness to mobile computing applications. *IEEE Transactions on Software Engineering*, 24(5), May 1998.

[41] Jay Werb and Colin Lanzl. A positioning system for finding things indoors. *IEEE Spectrum*, 35(9):71–78, September 1998.