Solar: Building A Context Fusion Network for Pervasive Computing

Guanling Chen
Dartmouth College

SOLAR: BUILDING A CONTEXT FUSION NETWORK FOR PERVASIVE COMPUTING

Dartmouth Technical Report TR2004-514

A Dissertation

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Doctor of Philosophy

in

Computer Science

by

Guanling Chen

DARTMOUTH COLLEGE

Hanover, New Hampshire

August 2004

Abstract of the Dissertation

SOLAR: BUILDING A CONTEXT FUSION NETWORK FOR PERVASIVE COMPUTING

Dartmouth Technical Report TR2004-514
by
Guanling Chen
Doctor of Philosophy in Computer Science
Dartmouth College, Hanover, NH
August 2004
Professor David F. Kotz, Chair

The complexity of developing context-aware pervasive-computing applications calls for distributed software infrastructures that assist applications to collect, aggregate, and disseminate contextual data. In this dissertation, we present a Context Fusion Network (CFN), called Solar, which is built with a scalable and self-organized service overlay. Solar is flexible and allows applications to select distributed data sources and compose them with customized data-fusion operators into a directed acyclic information flow graph. Such a graph represents how an application computes high-level understandings of its execution context from low-level sensory data. To manage application-specified operators on a set of overlay nodes called Planets, Solar provides several unique services such as application-level multicast with policy-driven data reduction to handle buffer overflow, context-sensitive resource discovery to handle environment dynamics, and proactive monitoring and recovery to handle common failures. Experimental results show that these services perform well on a typical DHT-based peer-to-peer routing substrate. In this dissertation, we also discuss experience, insights, and lessons learned from our quantitative analysis of the input sensors, a detailed case study of a Solar application, and development of other applications in different domains.

Acknowledgments

Thanks to my parents for their love and encouragement that have always been a valuable asset.

My most heartfelt thanks are due to my advisor, David Kotz, for countless invaluable discussions, ideas, and practical support that helped me see this research to conclusion. His experience and insight were of great assistance throughout my time at Dartmouth.

Thanks to Apratim Purakayastha for supervising my internship at IBM Research and for strongly supporting my job search, both at a critical time of my professional life. He has been a great friend and colleague.

I also want to thank Bob Gray and Ron Peterson, for their incredible patience answering my questions and their willingness to work with me.

I thank my fellow students that made my time at Dartmouth an enjoyable experience. Special thanks to my officemates who made my working environment lively: David Wagner, Lea Wittie, Kazuhiro Minami, Srdjan Petrovic, and Udayan Deshpande.

Most of all, thanks to my wife You for her loving support in the past three years.

Funding

I 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. This project was also supported under Award No. 2000-DT-CX-K001 from the Office for Domestic Preparedness, U.S. Department of Homeland Security. Points of view in this document are those of the author(s) and do not necessarily represent the official position of the U.S. Department of Homeland Security or any other sponsor.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The essence of *pervasive computing*, also called *ubiquitous computing*, is to enhance the environment by embedding many computers that are gracefully integrated with human users [137]. At the time Mark Weiser proposed this vision in the early nineties, however, the hardware needed to achieve pervasive computing simply did not exist. After more than a decade of technology progress, computers (either specialized or general-purpose) in different form factors and heterogeneous wireless networks are now available as commercial products. As a result, we are in a much better position to pursue the pervasive-computing vision and indeed we have seen pervasive-computing projects growing in both academic universities and industries.

A pervasive-computing environment challenges the existing computational models with a large scale of devices and users, portable but resource-constrained hardware platforms, heterogeneous and volatile wireless networks, non-conventional user interfaces, and the need for dependable and adaptive system software [138, 117]. In particular, an information-rich pervasive-computing environment could be overwhelming to the users surrounded by embedded devices. To gracefully integrate a computation and communication saturated environment with human users, pervasive-computing applications need to be *context-aware*. Namely, the applications must be aware of and adaptive to the situation in which they are running to avoid exposing unnecessary information and complexity to end users.

There are many definitions of *context* in the research community [33], since context may involve many aspects such as the state of the physical space, of the human users, and of the computational resources. Lieberman and Selker loosely define context to be any input other than the explicit input and output [83], as shown in Figure 1.1, where explicit input is user's intentional action such as key strokes and mouse clicks.

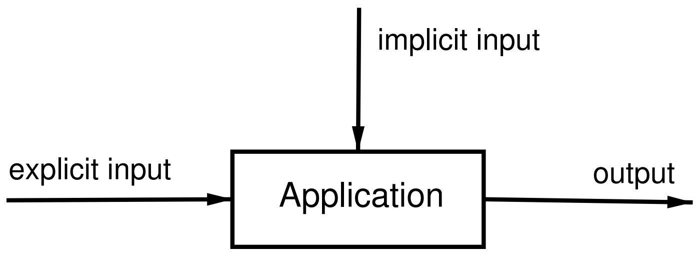


Figure 1.1: Context is defined as implicit input to applications.

Applications typically derive their desired context information (implicit input) from physical sensors and online information sources. A critical challenge for these applications is that the sensor data may not be accurate, since each sensor only has a limited view of the reality and the hardware itself may be error-prone. So the conversion of raw data into high-level context information, such as the user's current activity, requires applications to pre-process the data by filtering, transforming, and even aggregating the same or different types of distributed sensors to improve the quality of the derived context. Only with a reasonably-accurate context can applications be confident to make adaptation decisions.

The context computation could involve simple filtering based on a value match, or could involve sophisticated data correlation and machine-learning techniques. This process, deriving higher-level understanding from lower-level sensory data, we call *context fusion*. We believe context fusion plays a critical role in improving the accuracy of derived context, although the exact fusion technique to use is application and domain specific.

Consider a message delivery application, where a message destined to a particular user could be delivered at a nearby phone or at some other device carried by the user. Here the user location is the key context (implicit input) and can be obtained from one of many commercial or experimental location-provision systems, each of which has different accuracy, precision, and coverage in a particular deployment [65]. It is necessary to combine the output of all available locators to yield the best location estimation. For instance, Hightower and others used a trained Bayesian Network for this purpose [66]. To deliver the message based on the user's interruptibility, such as to queue the message temporarily if the user is having a meeting with her boss, the application may have to leverage more sensors to derive more activity context. We discuss one particular approach we implemented using pressure and motion sensors in Chapter 7.

Given the overwhelming complexity of a heterogeneous and volatile pervasive computing environment, it is not acceptable for an individual application to maintain connections to sensors and to process the raw data from scratch, because it increases the programmer's burden and the application may work poorly on a resource-constrained mobile device. On the other hand, it is not feasible to deploy a common context service that could meet every application's needs, since context is application-specific and diverse applications have different information needs. Instead, we envision an infrastructure that allows applications to share the same data-fusion computations, and to inject additional fusion functions for context customization if necessary.

The goal of our research is thus to provide such an infrastructure-based system that is both flexible enough to meet diverse application needs and scalable enough to service hundreds and thousands of sensors, applications, and users. This kind of system should not just be a Content Delivery Network (CDN) that connects multiple sources and sinks. Instead, the system needs also to accommodate the application-specific data-fusion functions and deliver only higher-level context to applications, which typically do not want raw sensory data. We call the system a Context Fusion Network (CFN).

In the rest of this chapter, we first present more background information on context-aware pervasive computing, then we discuss some motivating context-aware applications, and finally we summarize our contributions and give an outline of this dissertation.

1.1 Background

Many authors do not distinguish between "pervasive" and "ubiquitous" when it comes to computing visions. Some others argue that the difference exists: pervasive computing aims to *make* information available everywhere while ubiquitous computing *requires* information to be available everywhere [57]. Although the pervasive-computing essentials have emerged, the ubiquitous-computing age is still yet to come. On the other hand, we have seen major pervasive/ubiquitous computing projects booming in academia, such as project Aura at CMU,[1] Endeavour at UC Berkeley,[2] Oxygen at MIT,[3] and Portolano at the University of Washington.[4] Industry examples include work at AT&T Research, HP Labs, IBM Research, Intel Research, and many others. Each of these efforts addresses a different mix of issues and a different blend of near-term and far-term goals. Together, they represent a broad communal effort to make pervasive and ubiquitous computing a reality. For the purpose of this dissertation, we treat the two terms synonymously.

The pervasive-computing philosophy is essentially user centered, and the goal is for the technology to be "calm" or "invisible" to the users. Examples of older technologies that are calm include paper and motors [99], which have been so ubiquitously employed and natural to use that the users typically do not think of how to use them when using them. The pervasive-computing paradigm aims to build user-centric and information-rich applications. The path of pervasive computing does not lead to building powerful mainframe computers. Instead, pervasive computers should have many different configurations specialized and optimized for individual tasks. Norman gives an analogy comparing individual specialized tools, such as scissors, knife, and screwdriver, to the "all-in-one" Swiss Army knife that trades utility and usability with portability [99]. He calls the computer that is designed to perform a specific task an *information appliance*. A distinguishing feature of information appliances is the ability to share information among themselves.

Researchers at Xerox PARC pioneered the approach of building various computers in different form factors, such as palm-sized *tabs*, paper-like *pads*, and wall-mounted *boards* [137]. The researchers had to build many things from scratch, assembling hardware, designing wireless networks, and writing new operating systems, middleware infrastructure, and applications. Among the three devices, ParcTab is considered by many to be the most significant effort [135]. The constraints on the hardware capability, however, prevented pervasive computing from thriving at that time.

Context-awareness has been a key approach to meet the goal of pervasive computing, graceful integration with human users, since the early research days. Schilit and other researchers at Xerox PARC categorize context-aware applications that they built on top of their ParcTab platform [118]. They identify four context usage patterns: proximate selection, automatic contextual reconfiguration, contextual information and commands, and context-triggered actions. Later Satyanarayanan summarized the challenges for context-aware pervasive computing, such as context representation, middleware support, and adaptation models [117].

Like many other terms, however, "context" is an overloaded term. Many researchers have different perspectives on the definition of context, which typically involves the state of physical space, human users, and computational resources [33]. In particular, context is a piece of application-specific and time-sensitive information. Given desired context, applications may automatically adapt to the

[1]http://www.cs.cmu.edu/~aura
[2]http://endeavour.cs.berkeley.edu
[3]http://oxygen.lcs.mit.edu
[4]http://portolano.cs.washington.edu

context by changing their behaviors, which we call *active* context-awareness. The application may also present the new or updated context to an interested user or make the context persistent for the user to retrieve later, which we call *passive* context-awareness.

While application context is a rich concept, only location information has attracted much attention. The location of the objects and users is immediately useful to a large set of applications. Based on location information we might infer high-level context such as meetings, often with the assistance of other inputs. The recognition of the importance of location context has resulted in numerous active projects in location-provision systems [133, 140, 61, 6, 106, 29, 110, 63, 79] and location models [80, 97, 136, 67, 18, 66, 73].

The need to apply data-fusion to multiple incoming sensor streams to improve the quality of computed context has recently been recognized by researchers [66, 44]. While it may be possible to integrate all sensors on a single platform for a particular application, such as an augmented mobile phone [120], we are concerned about larger scenarios, with distributed sensors and multiple applications on different devices that may benefit from the aggregation of multiple data sources. These applications require customized context tailored to their needs, but they also may have some standard data-fusion steps similar or identical to those used by other applications.

Several research efforts address the need for distributed data-fusion for higher-level contextual understanding. Dey and others propose the Context Toolkit, wrapping sensors with a *widget* abstraction while computing context using *aggregators* [50]. Hong and Landy propose the Context Fabric that answers context queries with an automatically constructed data-flow path by selecting appropriate operators from a repository [68]. Both systems, however, focus on the interface between systems and applications with the goal of reducing programming complexity. These systems have largely ignored system issues such as mobility, scalability, and reliability.

We list several research directions in our survey of context-aware computing, such as context-sensing techniques, context modeling and representation, supporting system infrastructure, and security and privacy [33]. In this dissertation, we focus on a flexible and scalable context-fusion infrastructure that collects, aggregates, and disseminate contextual information.

1.2 Context-aware applications

One important application area is *Smart Spaces*, either office buildings or residential homes such as MIT's House_n[5] and Georgia Tech's Aware Home.[6] These smart spaces typically contain instrumented sensors and actuators, augmented daily objects [56, 123], and unconventional devices [134, 103, 122, 81]. A CFN can provide a unified approach to connect the sensors and their applications, each of which may use a CFN to customize individual information needs. The applications may use aggregated context, such as location and user activities, to teleport an X Window session [114, 61], to automatically route a phone call [133], to guide a user through a space [2], to proactively remind users of their tasks [49, 88], to help users interact with nearby objects [53, 12], to coordinate group interaction [75], to help occupants of a space control the environment [70], to assist with child education [32], to assist with laboratory experiments [4], and to automate elder care tasks [124]. Extensions of similar applications to a wider area, such as a campus or city, include location-based content delivery (tour guide) and annotation [2, 42, 58, 19]. Schilit and others give a categorization

[5]http://architecture.mit.edu/house_n/

[6]http://www.gatech.edu/innovations/awarehome/

of typical context-aware applications in such environments [118].

Another application area is *Emergency Response*, dealing with a disaster or crisis, either natural or man-made, in a timely and effective manner [91]. In such situations, diverse sensors, which are either deployed in the environment, carried by victims and responders, or installed on various equipment and vehicles, produce a huge amount of data. Human users also may act as data sources by entering observations using voice or text as they move around. A CFN can provide a scalable information infrastructure to handle hundreds and thousands of information flows. A CFN also can facilitate hierarchical data fusion to provide situational awareness and resource availability to decision makers from various participating organizations, each playing a different role in the Incident Command Hierarchy. The contextual information must be customized and prioritized to suit their roles and command levels. As the response proceeds and the crisis evolves, decision makers may use the CFN to quickly deploy new data-fusion functions and dynamically adapt the information spaces to the changes. A battlefield is another large-scale environment with similar information needs for intelligence gathering and situation awareness [16, 15].

1.3 Research challenges

One goal of a CFN is *flexibility*. A CFN is not the same as a common context service, since there is no way for any service provider to foresee all the context needs of diverse applications. Instead, a CFN must allow both deployment of well-known context services [66], and application-specific customization and user-specific personalization. In addition, a CFN should not limit the expressiveness of the data-fusion algorithms. Instead, it should accommodate arbitrary context computations chosen by individual applications. Finally, a flexible CFN also should encourage the sharing of context computation across multiple applications to facilitate both development and deployment.

A particular challenge for both the smart space and emergency response environments is system *scalability*. A CFN must be able to handle a large number of sensors, devices, applications, and users. It should be easy to increase the CFN capacity to handle increased load as necessary. In particular, a CFN must provide scalability across many layers, such as a naming and discovery layer, a data dissemination substrate, and fault recovery modules. Sharing of context computation across applications increases scalability by reducing redundant computation and network traffic.

Host and physical *mobility* are inherent in a pervasive-computing environment and must be addressed explicitly. The desired portability of a mobile device leads to asymmetric capabilities between a mobile and an infrastructure host. A moving device connecting to a CFN may traverse both the geographic and network boundaries. On the other hand, logical mobility also may play an important role in a CFN, which may move context fusion components in the infrastructure to balance the computation load or efficiently use the available bandwidth. As with scalability, mobility should be designed across multiple service layers.

The overall complexity of a pervasive-computing environment may quickly overwhelm a human user's ability to manage the supporting systems. Thus a CFN must be *self-managed* with minimum user intervention, adapt to the movement of applications and fusion components due to physical and logical mobility, and proactively monitor host failures to automatically recover lost components on other hosts. A CFN also must have a garbage collection mechanism to reclaim resources when application-specific fusion components are no longer in use.

In summary, a CFN must provide a flexible and self-managed pervasive computing platform, with scalability and mobility as inherent design goals. Our prototype system, named *Solar*, is a context fusion network that meets these research challenges.

1.4 Summary of contributions

In this section, we summarize the contributions of this dissertation and give an outline of the following chapters.

- Solar provides a flexible and scalable context fusion network for pervasive computing environments. Its programming model is based on a graph-composition model. This model allows applications to deploy custom data-fusion functionality inside the network. Chapter 2 discuss the composition model and Solar's service-oriented architecture.

- Solar's application-level multicast service employs a policy-driven data-reduction technique that allows loss-tolerant context-aware applications to trade completeness for fast delivery in case of buffer overflows, caused by rapid data streams and slow receiver. We present its design, implementation, and evaluation in Chapter 3.

- Solar's naming service supports persistent queries and context-sensitive resource discovery, which handles environment dynamics and allows applications to off-load most traffic and computation into the infrastructure. We present its design, implementation, and evaluation in Chapter 4.

- Solar's dependency management service employs a set of component monitoring and dependency tracking protocols that automatically recover operators that are lost due to host failures, and automatically adjust the operator-graph structure according to changes in resource availability. We present its design, implementation, and evaluation in Chapter 5.

- Our detailed analysis of several sensor traces, collected from location tracking systems that are in daily usage, provides quantitative characteristics of typical pervasive-computing environments. We present this analysis in Chapter 6.

- Our experiences, insights, and lessons learned from building many context-aware pervasive-computing applications, together with an open-source software package, provide valuable benefits to other researchers in the community. We present application case studies in Chapter 7.

We summarize our conclusions, and ideas for future work, in Chapter 8.

Chapter 2

Solar Overview

Solar is a middleware infrastructure with two kinds of clients: *sensors* as data sources and *applications* as data sinks. A sensor may publish a data stream, by pushing data items called *events* into Solar. Some sensors also may have a pull-based interface, allowing users to query its current state. Applications ask Solar to find specified sensors and to execute application-supplied data-fusion *operators* to compute context. An operator is an independent data processing module that takes one or more data sources as input and acts as another data source.

We have implemented two Solar prototypes, both in Java. Both prototypes adopted an operator composition programming model and similar design choices [35]. We implemented the first prototype, with a centralized architecture for simplicity, for two "pervasive-computing" seminar courses in which Solar was used by students to develop applications [34, 41]. Our experience with the first prototype, including an analysis of a sensor environment [38], performance and interoperability [141, 142], the security and access control design [93, 87], and several application studies [88, 132], contributed to the design and implementation of a second version of Solar [40]. The second prototype used a fully distributed and self-organized architecture, and the software package consisted of more than 13,000 lines of code. The later version also makes several research contributions on its generalizable operator-management services [36, 37, 39].

In this chapter, we first discuss Solar's operator composition model and then present the system architecture and services that manage the application-supplied operators. We discuss some related work in Section 2.3.

2.1 Operator composition

One of Solar's goals is to facilitate the development and deployment of context-aware applications. Given an environment where sensors are shared by many applications, we note that these applications typically go through some similar data-processing steps, such as filtering, transformation, and aggregation. It is critical, then, for Solar to provide a modular framework that promotes software re-usability.

We consider two kinds of reuse: one is *code-based reuse* when applications import existing modules from documented libraries, such as the one included in Java's development kit;[1] and the

[1]http://java.sun.com/j2se/

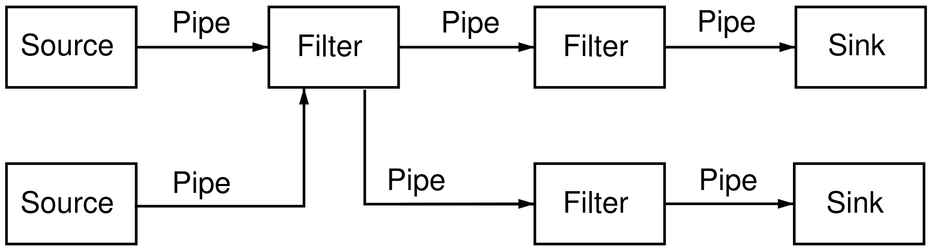


Figure 2.1: The filter-and-pipe software architecture style promotes reuse and composition.

other is *instance-based reuse* when applications discover and use already deployed data-fusion components. From the application's viewpoint, Solar encourages a modular structure and reduces programming time through code-based reuse. From the system's viewpoint, Solar minimizes redundant computation and network traffic and increases scalability through instance-based reuse.

2.1.1 Filter-and-pipe pattern

One popular software architectural pattern for data-stream oriented processing is *filter-and-pipe* [55], which supports reuse and composition naturally. In a filter-and-pipe style, as shown in Figure 2.1, each component (filter) has a set of inputs and a set of outputs. A component reads streams of data on its inputs and produces streams of data on its outputs. A connector (pipe) serves as conduits for the streams, transmitting outputs of one filter to inputs of another. The data flow starts from a source, through a sequence of pipes and filters, and reaches a sink.

There are several advantages of the filter-and-pipe pattern [121, 11]: 1) filters are independent and can be treated as black boxes, and this isolation of functionality helps to ensure quality attributes such as information hiding, high cohesion, modifiability, and reuse; 2) filters typically do not know the identities of their upstream and downstream components, and this simplicity helps to ensure low coupling; 3) pipes and filters can be hierarchically composed, and higher order filters can be created from any combination of lower order pipes and filters; 4) the construction of the pipe and filter sequence often can be delayed until runtime (late binding), and this permits a controller component to tailor a process based on the current state of the application; and 5) since the process performed by the filter is isolated from other components in the system, it is relatively easy to run a pipe-and-filter system on parallel processors or in multiple threads on a single processor. Example implementations of filter-and-pipe style in practice include Unix pipes [5], some Web servers [139], modern language designs [128], and a software router [78].

While Solar uses the filter-and-pipe pattern as its basic structure, we have several additional design considerations. First, we need a fan-in and fan-out structure, since a context computation may involve multiple sources and a component may be shared by multiple sinks (see Figure 2.1). Second, we need to implement the architecture in a distributed fashion for improved scalability. A centralized component is a potential bottleneck when dealing with many data sources and applications. Finally, the fact of distribution raises the inevitable issues of host failures and network congestion, which need to be addressed in the system design.

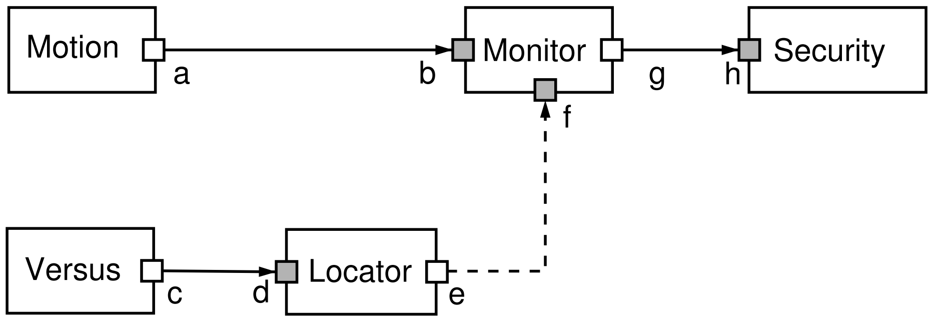


Figure 2.2: An example operator graph for a security control application.

2.1.2 Operator graph

In this section we present Solar's extensions to the filter-and-pipe style in detail. First we define some terms for use during the following discussion. In our terminology, we call a filter an "operator" and a pipe a "channel". A channel is directional and has two ends: one is a *source* and the other is a *sink*. A sensor is a source and an application is a sink; an operator is both a source and a sink.

An operator is a self-contained data-processing component, which takes one or more data sources as input and acts as another data source. Each operator has a set of *input ports* and a set of *output ports*. For simplicity, we call an input port an *inport* and an output port an *outport*. A port has a unique identifier to distinguish itself from other ports of the same operator. A channel connects an outport of an upstream operator to an inport of a downstream operator, and the direction of a channel indicates the direction of data flow.

A sensor may have a set of outports, but not any inports. On the other hand, an application may have a set of inports but not any outports. For an operator to function in a graph, it must have at least one inport and one outport connected with channels, and an outport can have multiple channels connected.

A port can be either *push-based* or *pull-based*. A channel connecting a push-based outport and a push-based inport is a *push channel*. A channel connecting a pull-based outport and a pull-based inport is a *pull channel*. With a push channel, the source spontaneously passes data to the sink. With a pull channel, the sink sends an explicit request to the source and the data then is passed through the channel. It is illegal for a channel to connect two ports with mismatched push/pull types.

The simple port-based interface allows us to easily connect the sensors, operators, and applications with channels to form an acyclic graph. We call this kind of data-processing graph an *operator graph*, and show an example in Figure 2.2. Here b, d, f, and h are input ports, while a, c, e, g are output ports. The lines $a \to b$, $c \to d$, and $g \to h$ are push channels, while $e \to f$ is a pull channel (denoted with a dashed line). In this scenario, a simple security control application enables/disables some functionalities in a particular room based on the identity of the user. The "monitor" operator receives events from a motion sensor, then pulls a list of people from a "locator" operator that receives events from a Versus location tracking system (Section 6.1). The output of the monitor, containing motion status and a list of users, is used by the security application to decide whether to disconnect the network, encrypt certain data files, or turn off some displays if the users do not have security clearance. The application also may trigger an alarm if there is motion in the room, but nobody is detected by Versus.

2.1.3 Functional separation

In object-oriented programming, the contract between an object and its user is a set of method signatures, including the method name, return type, parameters and their types. Similarly, the contract between operators and the composer, which builds an operator graph, is the *port specification* of the operator's inports and outports, such as the port identifier and push/pull type. The composition must obey the rule that two ends of a channel have the same push/pull type. While currently not implemented in Solar, we could further specify the structure and type of the data passing through the inports or outports, and even what requests a pull-based inport and outport could issue and handle. This additional information adds more constraints on properties of a legal channel.

The specification of an operator is documented by its developer and would best be stored in a *code repository* with APIs allowing programmable inspection. A composer who builds an operator graph, either a human or a program, uses the code repository as a library to import the operator modules into the composed graph. The code repository thus enables code-based operator reuse. On the other hand, any instantiated operators also may register a *name advertisement* in the *name space* and act as a virtual sensor (Chapter 4). In a composed operator graph, the inputs are specified as *name queries* that are resolved by the name space to select from existing real or virtual sensors (operators). The name space thus enables instance-based operator reuse.

From another point of view, the developer is responsible for the operators and the composer is responsible for the channels. They interact through the port specifications. Both operators and channels are first-class objects; the developer provides functionalities of operators and the composer customizes the channel behaviors. For instance, the composer may specify a policy to drop or summarize data in a channel whenever it overflows (Chapter 3). For a pull-based channel, the composer also may specify that the channel should pull (query) its source periodically and cache the results, which then are available for the sink when it actually sends a request. Some suitable caching policies are given in iQL [44].

2.1.4 Composition language

Solar provides an XML-based language for operator composition. We illustrate the language using an example here. Suppose an application wants to detect a human sitting on a chair, so that it may automatically turn on the desk lamp for reading. We attach a pressure sensor and a motion sensor to the chair, as shown in Figure 2.3. While a single sensor could detect human presence with some accuracy, combining two sensor outputs will give us fewer false positives, such as when the chair is bumped by a walking user or when a heavy object is placed on the chair.

The operator graph used to detect human presence is straightforward (Figure 2.4). The operator "presence fusion" takes two inputs, both through push-based channels, and outputs whether there is someone sitting in the chair or not using a particular fusion algorithm, which could be as simple as reporting human presence if both sensors claim there is a user sitting in chair, or as complex as using a supervised machine-learning approach to predict human presence based on historical observations. In Section 7.1, we present and evaluate a specific solution to detect ongoing meetings based on human presence.

We show how this operator graph is encoded with Solar's XML language in Figure 2.5. We first define the fusion operator with a variable name `$presence`, which uses the specified Java class and is initialized with a particular parameter. The channel element has source and sink attributes. The source could be either a previously defined operator or a name query as shown in our example.

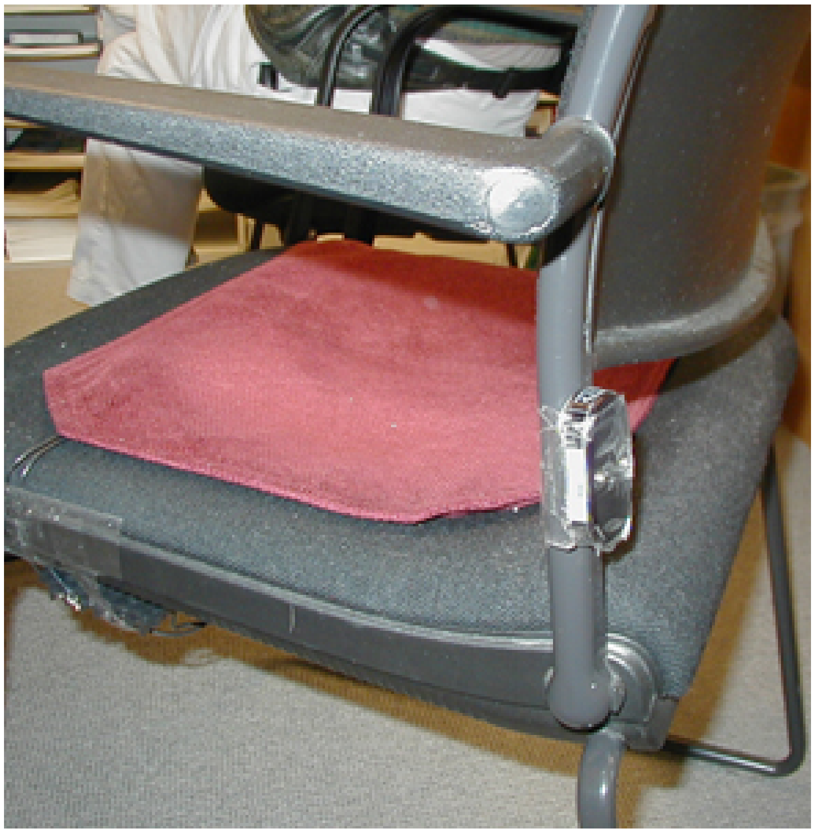


Figure 2.3: An instrumented chair with a wireless pressure (placed in the seat mat) and a motion sensor (taped at the back of the chair).

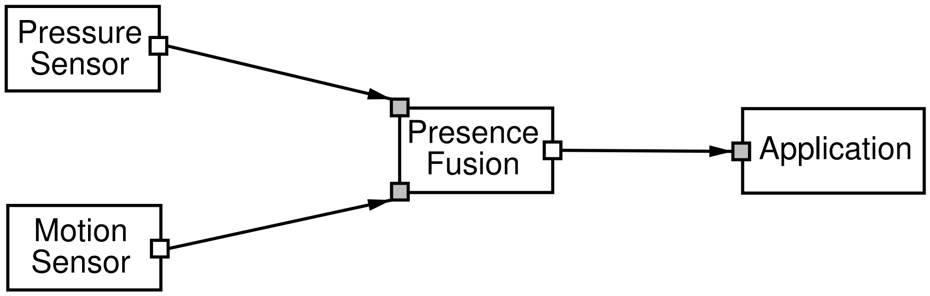


Figure 2.4: A simplified operator graph for detecting human presence in an instrumented chair.

```
<fusion>
  <operator name="$presence"
    classname="solar.app.PresenceFusion" >
    <param name="delay" value="2" />
  </operator>

  <channel source="[sensor=motion, chair=116]"
    sink="$presence" inport="mport" />

  <channel source="[sensor=pressure, chair=116]"
    sink="$presence" inport="pport" />
</fusion>
```

Figure 2.5: An XML example that encodes the operator graph shown in Figure 2.4.

Here `[sensor=motion, chair=116]` is an attribute-based name query, which is resolved to the motion sensor attached to a particular chair. We discuss the details of naming and discovery in Chapter 4.

Besides a source and a sink, a channel also may define the outport for the source and the inport for the sink. Note that here the outport is omitted, which means that the sensors have only one outport that will be used. The default types of these channels, if not specified, is push based. The application that deploys this operator graph sets up a channel to an outport of the presence operator, which represents the last link in the operator graph.

2.2 Planetary overlay

The logical operator graph needs to be mapped onto physical hosts. For scalability reasons, we want to avoid a centralized architecture where all operators are executed on a single server, which would be a potential performance bottleneck and the single point of failure. Solar takes a fully-distributed approach and consists of a set of functionally equivalent hosts named *Planets*. The Planets connect with each other to form a service overlay using an application-level distributed hashtable (DHT) based peer-to-peer (P2P) routing protocol [115], as shown in Figure 2.6. The Planets are denoted as P, and they connect sensors S and applications A, and cooperatively execute data-fusion operators (filled circles).

A sensor may connect to any Planet, called a *proxy*, to register a name advertisement and to publish its data stream. An application also may connect to any Planet to request the composition of specified data sources and operators into an operator graph deployed across the Planets, through which the application receives the derived context. A client, either a sensor or an application, may disconnect from a proxy Planet P_1 and reconnect to P_1 or a new Planet P_2 some time later. There are several reasons that a client may switch to a new proxy Planet: the current proxy is overloaded, the client finds a "better" Planet that is either closer or more powerful, or the client's current proxy has failed.

The Planets cooperatively provide several services: operator hosting and execution, sensor/operator registration and discovery, data dissemination through operator graphs, operator monitoring and recovery, and operator garbage collection [40]. Later in this chapter, we describe a clean system architecture, in which inter-service invocations are local and intra-service communications are hid-

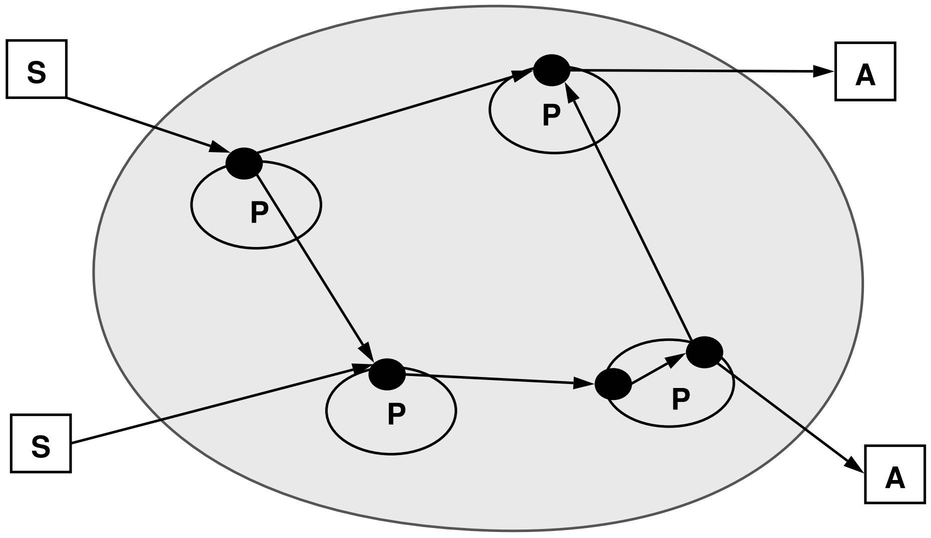


Figure 2.6: Solar consists a set of functionally equivalent Planets that form a service overlay.

den.

2.2.1 Distributed hashtable

Unlike a traditional client-server architecture, peer-to-peer (P2P) protocols do not distinguish the role of participating nodes. Each peer could be both a client and a server. The first generation of P2P protocols, mainly motivated by file swapping over the Internet, suffer from scalability problems. The representatives of such protocols are Napster,[2] Gnutella,[3] and Kazaa.[4] Napster peers are mediated through a centralized server while the other two protocols are fully distributed.

Several P2P protocols coming out of the research community are designed specifically to handle a large number of participating nodes [7], such as Pastry [115], Chord [125], Tapestry [143], CAN [111], and SkipNet [62]. They all provide a self-organized routing substrate in which each peer has a unique numeric key. The interface these protocols expose is sending a message to a numeric key. The message will be delivered to a peer with the numerically closest key. For simplicity, we chose Pastry because it is implemented in Java and could be easily integrated with Solar.

In Pastry, numeric keys represent application objects and are chosen from a large numeric space. Each Planet in Solar is a participating node (peer), which is assigned a key chosen randomly with uniform probability from the same key space. Pastry assigns each object key to the live node whose key is numerically closest to the object key. It provides a primitive to send a message to the node that is responsible for a given key.

The overlay network is self-organizing and self-repairing, and each node maintains a small routing table with $O(log(n))$ entries, where n is the number of nodes in the overlay. Messages can be routed to the node responsible for a given key in $O(log(n))$ hops. Simulations on realistic network topologies show that: 1) the delay stretch, i.e., the total delay experienced by a Pastry

[2]http://www.napster.com
[3]http://www.gnutella.com
[4]http://www.kazaa.com

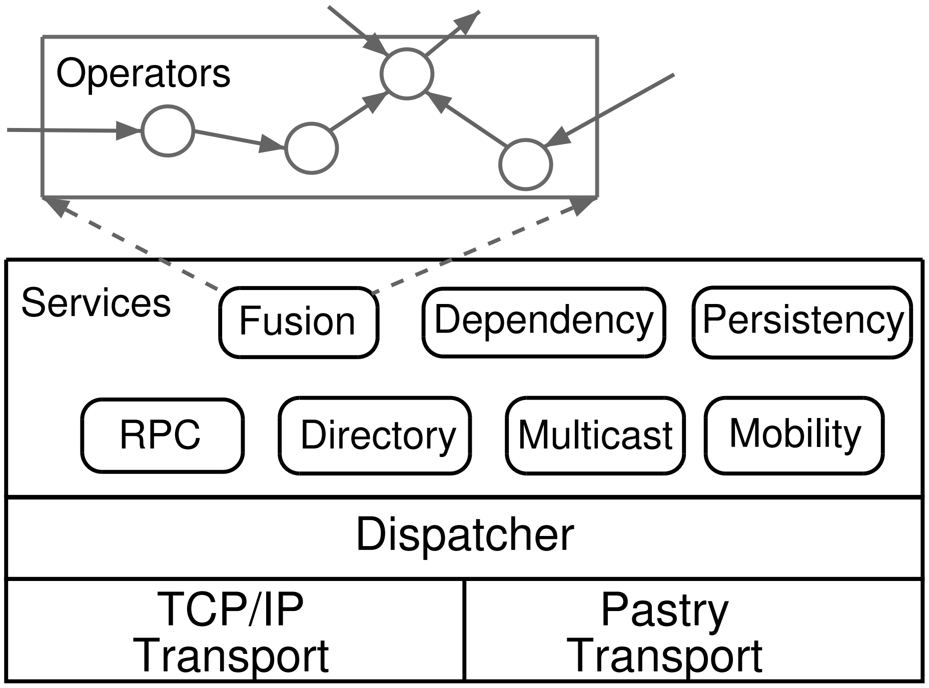


Figure 2.7: The architectural diagram of a Planet.

message relative to the delay between source and destination in the underlying network, is usually below two; and 2) the paths for messages sent to the same key from nearby nodes in the underlying network converge quickly after a small number of hops [28].

2.2.2 Planet architecture

Planets are execution environments for operators and they cooperatively provide several operator-management functionalities, such as naming and discovery, routing of sensor data through operators to applications, operator monitoring and recovery in face of host failure, and garbage collection of operators that are no longer in use. These requirements make Solar a complex infrastructure, and Solar provides a service-oriented architecture to meet the software engineering challenges.

We consider each functionality mentioned above as a *service*, which runs on every Planet. The core of the Planet is a service manager, which contains a set of services that interact with each other to manage operators and route context data. We show the architectural diagram of a Planet in Figure 2.7.

A Planet has two kinds of message transports: normal TCP/IP based and DHT (Pastry) based services. Thus a service running on the Planet may send a message with destination specified either as a socket address or as a numeric Pastry key. A dispatcher routes incoming messages from two transports to all other Solar services based on the multiplex header. From a service's point of view, it always sends messages to its peer service on another Planet. A service also may get a handle of another service on the same Planet and directly invoke its local interface methods.

An application, which is a Solar client, chooses a Planet and sends a request to the *fusion* service on it. The fusion service may ask the local directory service to discover the sensors desired by the application. The directory services on all the Planets determine among themselves how to partition the name space, or which Planet stores which name advertisements, and the directory users do not need to know the internals.

We believe that our architecture based on service-level modules is a simple but powerful abstraction to build distributed systems, particularly overlay-based systems. The idea builds on object-oriented modules and allows easy upgrading and swapping of service implementations as long as the service interface does not change. For instance, we could easily add a caching capability to the directory service to improve query performance. The hidden intra-service communication, either through TCP/IP or DHT transport, is important to ensure low service coupling. The local access to a remote service through a downloaded proxy in Jini shares a similar idea [130].

Our architectural approach also allows us to extract some common functionalities from several services and consider them primitive services. For instance, both the RPC service, which provides blocking remote calls, and the Heartbeat service, which sends certain messages to a remote peer periodically (not shown in Figure 2.7), are used by several other services. Thus we can improve the Planet efficiency by reducing redundant resource usage.

The set of services on a Planet is configurable and it is easy to add new functionalities to a Planetary network by simply adding another service to each Planet. For instance, we are building a Web proxy service on the Planet. Besides serving HTTP pages for clients, the proxies are all federated with the Planetary network so they could work together on content caching, prefetching, and re-directing the requests. The new service may simply reuse the existing services, such as directory, multicast, and fusion.

In reality, service integration may be more complicated. For instance, one service may only work with attribute-based directory, or it may desire a directory that may return query results within some latency constraints. This kind of advanced service interaction is still supported by Solar framework, but requires the service interface to explicitly expose its properties within some mutual ontology.

2.2.3 Service interaction

When a Planet starts up, it reads a configuration file that contains all the services to be initialized on that Planet. We show part of the configuration in Figure 2.8. The configuration simply contains a set of key/value pairs, while the second field (delimited by a dot) of the key is the service name, such as “directory”. A service may retrieve a local handle of another service from the Planet’s service manager given a service name.

Note that in Figure 2.8, we have two transport services: one based on IP and the other based on DHT. The two RPC services, which simulate remote blocking calls, actually use the same Java class but with different underlying transport. We discuss the multicast service in Chapter 3 and the directory service in Chapter 4. The dispatch service registers a callback with both transports to receive messages, which contain a multiplex header indicating the destination service. The dispatch service exposes the following interface to send a message:

```
dispatch(message, destAddress, serviceName, transportName)
```

The `destAddress` could be specified with either an IP address or a DHT key; the name of receiving service at the destination Planet is specified as `serviceName`; and `transportName` is the name of the transport service used to deliver the message.

```
service.dispatch.classname=
        solar.service.dispatch.SolarDispatchService
service.dispatch.transport=dht_transport ip_transport

service.directory.classname=
        solar.service.directory.DistDirectoryService
service.directory.transport=dht_transport
service.directory.rpc=dht_rpc

service.multicast.classname=
        solar.service.multicast.ScribePackService
service.multicast.dht_transport=dht_transport

service.ip_transport.classname=
        solar.service.transport.TcpTransportService
service.ip_transport.port=5465
service.ip_rpc.classname=
        solar.service.rpc.SolarRpcService
service.ip_rpc.transport=ip_transport

service.dht_transport.classname=
        solar.service.transport.PastryTransportService
service.dht_rpc.classname=
        solar.service.rpc.SolarRpcService
service.dht_rpc.transport=dht_transport
```

Figure 2.8: A portion of a Planet's service configuration.

We show partial service interaction in Figure 2.9. The `fusion` service executes local operators, and the dependency between operators are managed by the `dependency` service (Chapter 5). The arrow means that the source service relies on the sink service. Most services rely on the RPC service, though those arrows are not shown for simplicity. The `load balance` service measures the load on the local Planet and may decide to relocate some operators at runtime if the Planet is overloaded. In the next chapter, we discuss the `multicast` service that disseminates events from a publisher to receivers.

2.3 Related work

Xerox PARC's ActiveMap is one of the first systems to support context-aware applications. It uses a centralized architecture to manage location information about dozens of active badges in a building, deployed to support daily usage of the ParcTab system [119]. ActiveMap has limited scalability, but is sufficient to meet its specific goals. User agents and device agents selectively publish their representative's current location to the map server according to previously specified control policies. Applications may use the map server to discover location information, or directly send requests to the agents. ActiveMap uses IP multicast as its basic facility for disseminating location information. Solar aims to support a larger scale of application scenarios, and addresses system scalability in terms of number of sensors, number of applications, and the volume of data.

The Context Toolkit seeks to simplify the development of context-aware applications by wrap-

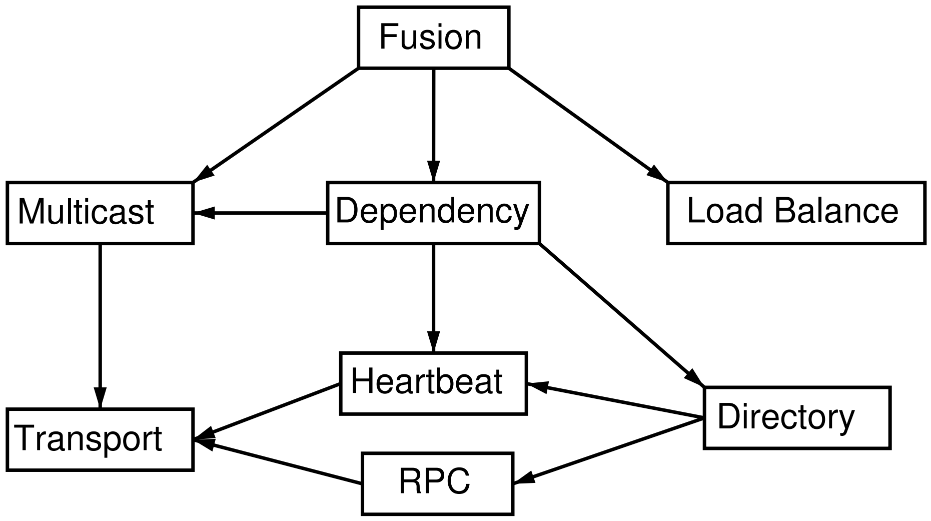


Figure 2.9: Partial service interaction on a Planet.

ping sensors with a widget abstraction and by providing pre-defined aggregators for commonly used context [50]. Its data-flow representation of aggregators bears some similarity to Solar's operator graph. The system structure, however, is established by administrators and becomes static at runtime. Solar differs by emphasizing application-specific context customization given the diverse needs of different applications.

The Context Fabric also focuses on the interface between system and applications [68]. It allows an application to specify a high-level context query, and based on the type of requested context, it automatically constructs a data-flow path by selecting operators from a repository. Similar path automation appears in Paths [77] and CANS [54]. While it is convenient, this approach gives the application little control and has restricted applicability due to the limited expressiveness of type matching. Unlike Solar, Context Toolkit and Context Fabric do not address key systems issues such as scalability, failure recovery, and flow control.

In industry, ContextSphere (formally called iQueue) shares many similar goals with Solar [45]. A Solar programmer needs to explicitly specify an information flow graph, while ContextSphere provides an implicit programming model by developing an expressive composition language named iQL [44]. Given an iQL program, however, it is possible to parse it into an operator graph that is deployable over Solar. On the other hand, we believe that the act of manually deriving an operator graph, and the temptation to use existing operator classes where available, will encourage programmers to derive similar graphs in similar situations, increasing the opportunities for re-use of data streams. Similarly, programmers will be likely to name operators for use by other users or in other applications [35]. The currently available literature about ContextSphere says little about how it manages its composers or other systems issues.

The EventHeap used by the iRoom project employs a tuplespace model, which aims to decouple the data producer and consumer [74]. This loosely-coupled coordination model reduces component interdependency and allows easy recovery from crashes. The simple interface of a tuplespace, tuple retrieval based on pattern matching, limits the expressiveness of data processing. In particular, there is no direct support for data fusion.

PQS is an information-fusion engine that tries to identify a process by observing its output data stream [13]. It currently employs a JMS messaging middleware, which could be replaced with Solar for large-scale deployments. On the other hand, PQS contains several ready-to-use fusion models that can be used by Solar operators.

The problem of achieving a new composite service by composing existing autonomous Web services has generated considerable interest in recent years. Researchers have been working on composition languages [82], specification toolkits [105], support systems [25, 24, 108], and load balancing and stability algorithms [107]. If we consider sensors in Solar as output-only services, we also could enhance Solar's composition model with previous results, such as a more powerful composition language, a rule-based composition engine, and algorithms for achieving certain quality of composed service.

Other emergency response and battlefield projects, such as Rescue [91], EventWeb [31], and JBI [16, 15], all have their own middleware systems for distributed data management. In particular, EventWeb and JBI have an event-processor composition model similar to the one used by Solar. Their supporting architectures are not clear from the literature.

Data aggregation is also a useful technique inside sensor networks to reduce unnecessary data transmission [17, 64, 86]. Unlike Solar, these systems work at a lower level and are designed for a resource-constrained environment, where the focus is on power consumption and communication costs. They are often designed for a single-application environment and the expressiveness of the data processing is fairly limited. On the other hand, Solar's operators may take advantage of infrastructure nodes to perform intensive data fusion. These sensor-network systems, however, are complementary to Solar since the aggregated results coming out of a sensor network could supply one event stream to Solar.

Recently we have seen many research efforts on continuous query over data streams from a database [126, 1, 96]. A large part of these efforts focus on the algorithmic efficiency of implementing SQL-like queries over data streams with little attention to more general data fusion or to support systems. It may be possible to execute the continuous queries with Solar, which provides a flexible and scalable platform to distribute and manage the SQL operators.

Chapter 3

Data Dissemination

In an operator graph, an outport of a sensor or operator may have multiple push channels connected, forming a fan-out structure. This means that every time the sensor or operator publishes an event through that outport, the event has to be delivered through all the connected outbound push channels. The simple approach to pass one copy of the event to each channel may not be the most efficient, for instance, if the sinks of multiple channels are on the same host. A common solution to improve the efficiency of such data dissemination problem is multicast [47]. The idea is to aggregate the channels and build a minimum spanning tree out of the network topology. Thus an event is only duplicated at a parent node for all its children, instead of being duplicated at the source (the root) for all receivers.

Solar disseminates events with an application-level multicast (ALM) facility built on top of its peer-to-peer routing substrate. ALM improves the scalability of data dissemination and does not rely on IP multicast, which is often turned off in practice. While ALM is not a new idea [72, 69, 112, 116, 26, 27], buffer overflow management remains a challenge in ALM. In this chapter, we first present the basics of ALM over Solar in Section 3.1. We motivate the data reduction requirements to handle buffer overflow on the dissemination path in Section 3.2, and we present our policy-driven approach in the rest of the chapter.

3.1 Application-level multicast

The clients, each of which host one or more *data endpoints* (either senders or receivers), are not part of the Planetary overlay. Instead, a client has to explicitly attach to a Planet to request services for its endpoints. The Planet to which the client attaches acts as the *proxy* for all the endpoints on the client. Each endpoint and Planet has a unique numeric key randomly chosen from the same key space. The subscriptions of a sender S are managed by S's *root* Planet, whose key is numerically closest to S's key among all live Planets. Note that a sender's root is not necessarily the same node as S's proxy. All the Planets are functionally equivalent and may play several roles simultaneously.

As shown in Figure 3.1, a data dissemination path is constructed as follows: the client hosting a sender S forwards all its published events to the sender's root SR via the proxy SP; then the events are multicasted to the proxy Planets of all subscribing receivers RP, hopping through a set of intermediate forwarding Planets MF; finally the events are forwarded to the clients hosting each receiver R.

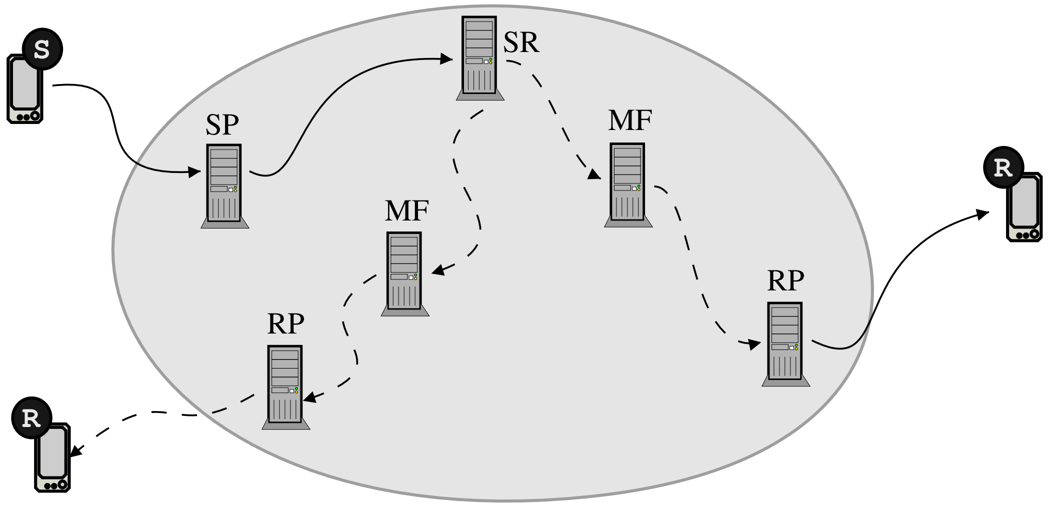


Figure 3.1: Multiple data dissemination paths converge to a multicast tree.

To establish a dissemination path, R sends a subscription request to K_S, which is the DHT key of S and could be discovered using Solar's naming service. The subscription request is routed to the Planet responsible for K_S, which is SR. The subscription is recorded on each of the intermediate Planets MF along the path. As multiple receivers make subscriptions, their data paths converge to a multicast tree rooted at SR. The sender S always delivers its events to its own key K_S using the DHT interface. Once the event reaches SR, it is forwarded through the multicast tree to all the receivers. Note again that the event is only duplicated at every parent Planet for each child in the multicast tree. Castro et al. present and compare some protocols that can be used to build an ALM on DHT-based peer-to-peer overlays [27].

Mobile clients may experience temporary disconnection caused by weak links or mobility hand-offs. During disconnection, a client may roam and change its network address (network mobility), and it may or may not choose the original proxy when it reconnects (host mobility). A client may voluntarily decide to change proxies if it finds a "better" overlay node, such as one that is closer or has a lighter load. The proxy also may make its own decision to disconnect a client, if the proxy is about to shutdown or is too crowded, and force the client to select a different proxy.

Thus the client and its proxy engage in a protocol maintaining state about each other. A client (and the endpoints it hosts) may appear in three states to the proxy, *attached*, *detached*, or *departed*. State transitions from *attached* to *detached* are triggered either by an explicit request or by missing several heartbeat signals. If the client has been detached longer than a configurable threshold, the proxy assumes the client has departed (and will not re-attach to this host). The proxy appears to the client in two states: either *attached* or *detached*, and transitions are managed in a way similar to the client state.

The sender client starts to buffer events for all receivers if it is detached from the proxy. A receiver proxy starts to buffer events for a receiver client when the receiver client is detached. If the receiver client departs, its proxy removes its subscription and all accumulated event queues. When a receiver R re-attaches to a proxy RP' different than its previous RP, it first asks RP' to join the multicast tree and cancel its subscription at RP. Then R asks RP for all the buffered events before requesting data from RP'. The sequence number in the events is used to prevent duplicated delivery. Fiege et al. [52] present another approach that we could use to handle host mobility.

3.2 Buffer overflow

Consider a receiver R that takes actions on received events. If consuming an event does not block R from receiving new events, new events are typically buffered, waiting to be processed in order. If the event consumption rate is consistently lower than event arrival rate, the buffer faces the danger of running out of space. We say that a buffer overflows if it is full but new events continue to arrive. As mentioned above, the events for a mobile receiver R are buffered at the proxy Planet during R's disconnection. Here the event consumption rate is zero, so this buffer is vulnerable to overflow. Finally, the buffer at the intermediate Planets also are subject to overflow due to network congestion.

There are two typical approaches to manage buffer overflow. First, the new events may be simply dropped if there is no more space in the buffer, which leads to arbitrary data loss. Second, the receiver may notify the sender about its buffer condition, either explicitly or implicitly, so the sender may slow down to prevent overwhelming the receivers. An IP-family protocol similar to the first approach is UDP, while the second approach is typically seen in a reliable data transmission protocol, such as TCP.

While it is convenient for the applications to have reliable delivery guarantees, it may require infinite storage (either in memory or on disk) at the sender, particularly when a sensor is continuously producing data. An infinite buffer is of course not feasible, and not desirable as well because it introduces long delay for the events at the tail of the buffer. In the case of reliable multicast, slowing down the sender due to some slow receiver hurts all others in the multicast group and thus may not be acceptable either. On the other hand, however, arbitrarily dropping data is also not acceptable for context-aware applications that are monitoring events.

We observe that many applications are loss-tolerant, which means that they can adapt to occasional data loss and often do not require exact data delivery. There are many examples of loss-tolerant multimedia applications, but we are mainly interested in non-multimedia applications. For instance, an application that maintains a room's temperature will likely be able to function correctly even if it misses several sensor readings. Similarly, an ActiveMap application can adapt to loss of location-change updates by fading the object at its current location as a function of time since the last update [90]. One reason these applications are able to tolerate data delivery loss is that they are designed to cope with unreliable sensors, which also may lead to data loss and inaccuracy.

In this chapter, we present a buffer-management module, named PACK, for the multicast service. PACK allows applications to specify data-reduction policies, which contain customized strategies for discarding or summarizing portions of a data stream in case of buffer overflow. The summaries of dropped data serve as a hint to the receiver about the current buffering condition; the receiver may adapt by, for example, choosing a different data source or using a faster algorithm to keep up with the arriving data.

In addition to the policies at the end hosts, it is necessary to install data-reduction policies on the buffers of the intermediate forwarding Planets, so they can be triggered closer to congested links or disconnected clients. It is not practical and may not be efficient to inject PACK functionalities into a widely deployed protocol stack (such as IP). Instead, we implement PACK policies at the application layer using the buffers above the networking stack. We assume that Planets are strategically placed in the infrastructure to form a multicast overlay service capable of executing data-reduction policies.

Our PACK buffer management provides three contributions. First, it enables customized data-reduction policies so loss-tolerant applications can trade data completeness for fresh data, low latency, and semantically meaningful data. Second, it employs an overlay infrastructure to support mobile data end-points for temporary disconnection and hand-off. Finally, it provides an adaptation mechanism so receivers may react to current buffering conditions.

We discuss the data-reduction policy in the next section. Note that unlike congestion control in the network layer, which makes decisions based on opaque packets since it does not recognize the boundaries of application-level data objects, the PACK policies work at the granularity of Application Data Units (ADU) [43], which we called events. Since PACK is able to separate the events that follow a common structure, PACK can get the *values* inside the event object, enabling a much more flexible and expressive policy space for receivers.

3.3 Data-reduction policy

Each sender produces a data stream, a sequence of events carrying application data such as sensor readings. We model an event as a list of *attributes*: each contains a *tag* string and a *value* object. Currently we assume that all events from the same sender have the same structure, namely, the same set of attribute tags.

To receive a data stream, the receiver *subscribes* to some sender. The sender client, intermediate forwarding Planets, and the receiver client form a dissemination path for that subscription. We allow many receivers to subscribe to a single sender, or a single receiver to subscribe to multiple senders. Conceptually there is a FIFO *queue* on each host of the path for a particular subscription, temporarily holding the events in transition. A *buffer* consists of multiple queues, each of which contains events from the same sender to which one or more receivers have subscribed. We discuss the detail of buffer management in Section 3.4.

Receivers may attach a data-reduction policy (or simply *policy*) to their queues (on any node of the path), to specify how to shorten the queue when it becomes full, by discarding and summarizing certain events according to applications needs. Figure 3.2 shows the overall structure of the multicast service, with two receivers subscribed to the same sender. Each receiver subscribes to the sender with a customized policy ($p1$ or $p2$). Policies are installed on all the hosts along the path from sender to receiver. Nodes on multiple paths contain multiple policies (node A contains both $p1$ and $p2$).

PACK, running on all clients and Planets, puts all events that arrive either from a local sender or from the network into its internal queue, where they wait to be consumed by a local receiver or transmitted to the next host on the path. If a queue becomes full, PACK triggers its associated policy to examine the events in the queue and determine which should be dropped. The policy also may specify how to summarize the dropped events into *digests*, which are placed in the resulting queue as well. On the receiver's client, PACK pulls events or digests from the queue and invokes a different interface of the receiver for each one. We now describe how to specify a policy and how PACK executes a policy.

3.3.1 Policy specification

A policy defines an ordered list of *filtering levels*, and each level contains a single *filter* or a chain of filters. The list of levels reflects a receiver's willingness to drop events under increasingly desperate

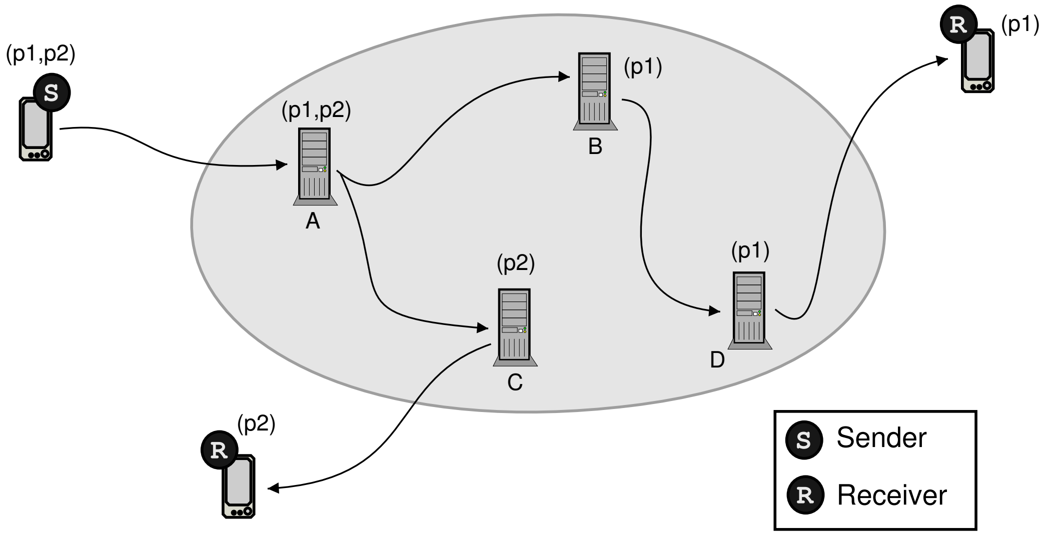


Figure 3.2: The multicast service consists of a set of Planets that run PACK policies.

overflow conditions: more important events are dropped by filters at higher levels than filters at lower levels. The policy may contain any number of levels. Given an event queue to be reduced, PACK determines which level to use and then passes the queue through all the filters defined up to and including that level, starting from the lowest level.

A filter is instantiated with application-defined parameters and determines what events to keep and what to drop given an event queue as input. The filters are independent, do not communicate with each other, and do not retain or share state. Since an event may contain several attributes, the filter typically requires a parameter indicating which attribute to consider when filtering.

Filters drop some events. Optionally a policy also may specify how to summarize dropped events using a single or chain of *digesters*. The result of summarization is a *digest* event injected into the event stream. Thus an event queue may contain a mixed set of events and digests. The digests give some rough feedback to the receiver about which events were dropped, and also serve as a buffer overflow indication; the receiving application may take action such as switching to different sources or using a faster algorithm to consume events.

We show an example policy in Figure 3.3 using XML syntax (although it is not the only possible specification language). First the policy specifies that all the filters apply to the attribute with tag "PulseRate". It is also possible to specify a different attribute for each filter. All dropped events are summarized to inform receivers about the number and average PulseRate value of the dropped events. The example gives a single filter for each buffering level. The first-level filter drops events whose pulse rate has not changed much since the previous event; the second-level drops all events that have a pulse rate inside of a "normal" range (since they are less important); and the last filter simply keeps the latest 10 events and drops everything else. In urgent buffering situations, all three filters are applied in sequence to each event in the queue.

Currently we support basic comparison filters, such as GT ($>$), GE ($\geq$), EQ ($=$), NE ($\neq$), LT ($<$), LE ($\leq$), MATCH ($=\sim$), and WITHIN ($[k1,\ k2]$). We also provide some set-based operators such as INSET ($\in$), CONTAIN ($\ni$), SUBSET ($\subset$), SUPSET ($\supset$), and some sequence-based operators such as FIRST (retains only the first value in a set) and LAST (retains only the last value

```
<policy attribute="PulseRate">
 <summary>
  <digester name="MEAN">
  <digester name="COUNT">
 </summary>
 <level>
  <filter name="DELTA">
   <para name="change" value="5"/>
  </filter>
 </level>
 <level>
  <filter name="WITHIN">
   <para name="low" value="50"/>
   <para name="high" value="100"/>
  </filter>
 </level>
 <level>
  <filter name="LATEST">
   <para name="window" value="10"/>
  </filter>
 </level>
</policy>
```

Figure 3.3: An example of data-reduction policy with three filters.

in a set). More advanced filters include UNIQ (remove adjacent duplicates), GUNIQ (remove all duplicates), DELTA (remove values not changed much), LATEST (keep only the last N events), EVERY (keep only every N events), and RANDOM (randomly throw away a certain fraction of events). The digesters for summarization are MAX, MIN, COUNT, SUM, and MEAN, which have typical semantics as their names suggest.

As indicated in Figure 3.3, our approach is to allow applications to compose predefined filters into a customized policy. We could have used a general-purpose language to express more general policies or even more general filters. The trade-off is that as the language gets more powerful and more complex filters are supported, it is more likely that PACK will involve more overhead for filter execution and eventually reduce system scalability [22]. Based on our experience so far, many loss-tolerant applications desire simple and straight-forward policies. Thus our strategy is to keep the filters simple and efficient, and to expand the filter repository as necessary.

3.3.2 Policy execution

Due to previous packing operations performed locally or at upper stream hosts, a queue may consist of a sequence of mingled digests (d) and events (e) as follows (the sequence number reflects the order in the queue instead of the original counter at the sender):

$$e_1, e_2, d_3, e_4, \ldots, e_5, d_6, e_7, e_8 \ .$$

Suppose a policy is executed on this queue and e_2 is to be dropped, d_3 should be updated using e_2. On the other hand, if all the events between e_1 and e_8 are to be dropped, a new digest should be computed based on dropped events. In particular, d_3 and d_6 should be combinable.

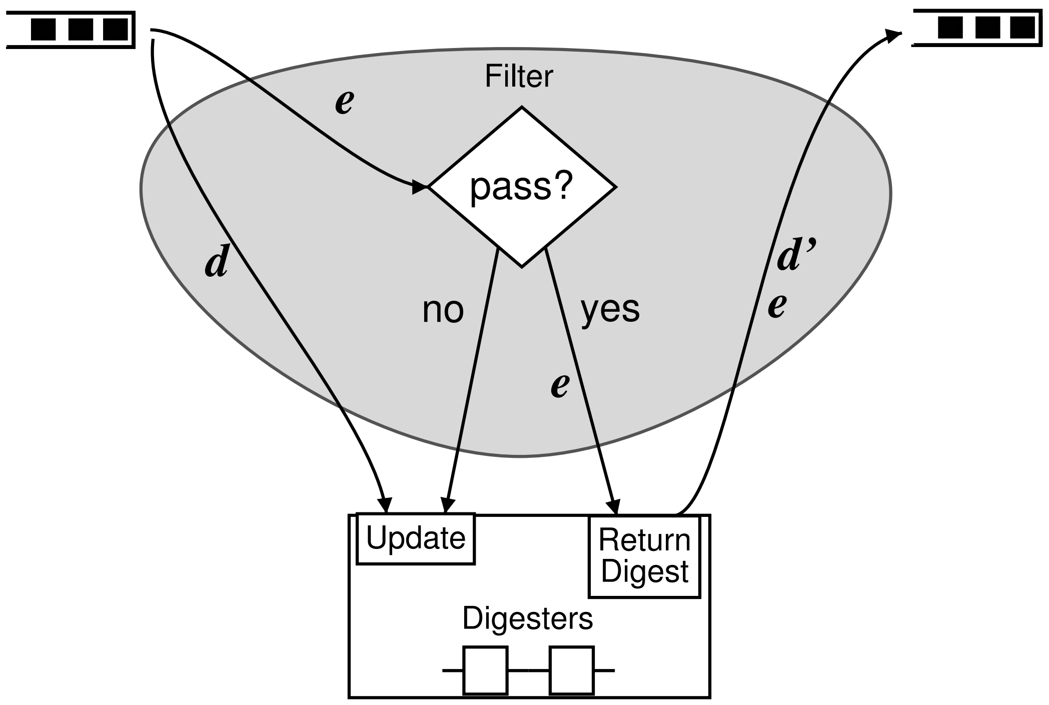


Figure 3.4: A filter takes an event e or a digest d from input queue in order.

Thus the digesters should be "associative" so they can be recursively applied on previous results. Note that, since the same policy exists on every host in a path, this associativity applies across hosts as well as within a host (when a buffer must be packed again). All the digesters we mentioned above (such as MAX, MIN) satisfy this requirement. If we were to provide a digester that computes the number of unique values in dropped events, the digests have to carry all the unique values so they can be merged or updated accurately. The number of unique values, however, may be unbounded and defeat the purpose of summarization. Although it is possible to use these digests as packing boundaries (so they do not have to be updated or merged), a queue may end up with many digests with little actual data and reduce the effect of filters applied later.

When a policy is triggered, PACK takes the input queue and forms a chain of filters up to the filtering level on which it has decided. PACK feeds the queue to the first filter, passes the resulting queue to the next filter, and so on until the last filter. Figure 3.4 visualizes how a single filter executes the policy. For each event e in the input queue, if it fails to pass the filter, it is used by digesters to update the current summary state, such as previously computed digest d in the input queue. If e passes the filter, a new digest d' is computed and placed in the output queue together with the satisfying event e.

A design alternative is to take one event from the input queue and check it against all filters until it is either fails in the middle or passes all. Only after the previous event has already run through all filters is the next event in the input queue admitted and put through the same procedure. Our approach, however, takes the input queue as a whole and feeds it through all filters. We believe the first approach limits what a filter can do since the event passes the filter only once and the filter does not know how many more events are coming. Our PACK filters, however, are able to perform tasks on the whole queue, such as LATEST and GUNIQ. The overhead of the two approaches, however,

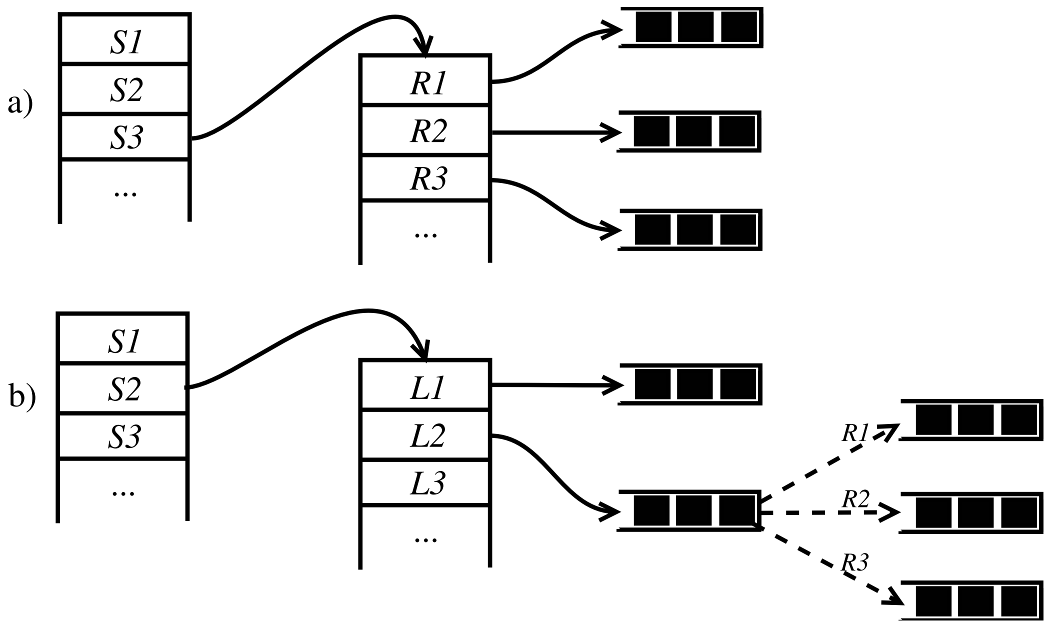


Figure 3.5: Two-level indexing structure of buffers, both having the sender's key as first index.

should be comparable since each event has to be checked against all filters in sequence.

It is possible that PACK may not be able to reduce a queue at all even after applying the highest filtering level. It may be that the policy does not apply well to current data values so all filters are not effective, or the link to the next host is congested or disconnected and the queue already has been filtered at the highest level. In such cases, PACK drops all the events in the queue and applies the policy's digesters, or COUNT as a default if the policy does not have one.

3.4 Buffer management

A buffer is a data structure containing multiple subscriptions or queues for receivers. We distinguish two kinds of buffers: one is the *local buffer* for receivers on the clients, and the other is the *remote buffer* containing events to be transmitted to clients or some Planet. Events in a local buffer are consumed locally by the receivers' event handlers, while the events in a remote buffer are transmitted across a network link. While there might be multiple endpoints on a client, there is only one local buffer for all resident receivers and one remote buffer for all senders. On a Planet, there are several buffers, one local and several remote, to serve the different roles the Planet may play.

Both local and remote buffers adopt a two-level indexing structure (shown in Figure 3.5), where the first index is the sender's key. The local buffer on a client uses the receiver's key as the second index, while a remote buffer uses link address as the second index. An entry for a given link address means there is at least one receiver subscribing to the corresponding sender across that link. The two indexes in a local buffer point to a queue for a single receiver. On the other hand, the two indexes in a remote buffer point to a shared queue for all receivers across the same link under normal conditions. As the shared queue reaches its limit due to, for instance congestion or disconnection, a private queue is created for each receiver and packed using individual policy.

On each client or Planet, a dispatcher thread pulls events from the network, adding a reference (pointer to event object) for each event into one or more queues, based on the headers of the received events. The header contains the sender's key sid, the receiver's key rid, and the destination *toward*. For instance, if *toward*="SP", the event was just admitted into the Solar overlay from some sender client. If rid is empty, the event is a multicast event destined to all subscribers of this sender. Otherwise, it is a unicast event destined to one specific receiver.

An alternative buffer design is to maintain a single queue for all the received events. Then, when packing is necessary, we need to scan the whole queue to find events for a particular subscription to apply that subscription's policy. It may use less memory, since our approach may put multiple references to the same event into several queues if there is more than one subscriber. We believe memory usage is not likely to be a significant concern, however, since the events are not replicated. The two-index structure for separate queues gives us greater flexibility to choose queueing and packing policies and reduces a large amount of implementation complexity.

3.4.1 Event routing

There are several types of buffers in the PACK system and we adopt the following naming convention for ease of discussion. A buffer has a name of capitalized letters ending with "B", and the prefix denotes the destination of the events in the buffer. For instance, a buffer named SRB contains all the events being forwarded to sender's root SR, where the SR depends on the sender's key. Buffer RCB contains events destined to receiver client RC.

A client has a local buffer RB (Receiver Buffer) for all receivers, and a remote buffer SPB (Sender Proxy Buffer) for all senders. The first index of RB contains a list of sid and the second index contains a list of subscribing local rid. Each second index of SPB contains only one entry, namely, the proxy's address. When a receiver makes a subscription, an entry is added to RB (extending the first- and second-level indexes as necessary). When a client receives an event relayed from its proxy, it adds it to the appropriate queue within the RB. When a client receives a new subscription from its proxy, it adds an entry (extending the indexes as necessary) to SPB. When a sender publishes an event, it adds it to the appropriate queue within the SPB.

Since a Planet may play several roles simultaneously, it maintains several remote buffers. One is the Sender Root Buffer (SRB) containing events being relayed to their senders' root. Another is the Receiver Proxy Buffer (RPB), which contains events being forwarded to a receiver's proxy RP. And finally, the Receiver Client Buffer (RCB) contains events that should be sent to directly connected clients. There is only one of each these buffers on a single Planet.

If a receiver R is currently detached, the RCB at its proxy is notified to "suspend" the queue for R, which means the dispatcher may continue to put events in the queue, but the scheduler is not allowed to pull events from the queue and send them to client RC. The queue is "resumed" when R is re-attached. Similarly, the SPB on the client also may be suspended and resumed when that sender client is detached or re-attached to its proxy.

A typical event flow is thus to traverse the named buffers as follows (the first and last buffer are on the clients while the middle four are on overlay nodes):

$$\boxed{\text{SPB} \rightarrow \text{SRB} \rightarrow \text{RPB} \rightarrow \text{RCB} \rightarrow \text{RB}\ .}$$

As PACK propagates a receiver's subscription request through Planets in the reverse direction, it adds a subscription entry in the two-level index structure, together with a PACK policy, to appropriate remote buffers along the path. The algorithm is described in Algorithm 1. Note that all the

Algorithm 1 Propagating subscription requests through the Planetary overlay. The *request* originates from R whose *toward* is initialized to be RP.

```
1:  $(sid, rid, toward, lasthop) \leftarrow request$
2:  if toward is RP then
3:      add subscription to RCB
4:      request.toward ← SR
5:      send request to sid's root
6:  else if toward is SR then
7:      add subscription to RCB
8:      if local-node is sid's root then
9:          request.toward ← SP
10:         send request to sid's proxy
11:     else
12:         send request to sid's multicast parent
13: else if toward is SP then
14:     add subscription to SRB
15:     request.toward ← SC
16:     send request to sid's client
17: else
18:     error
```

buffers mentioned here are remote buffers and the field *lasthop* is used to set up the second index of the buffer. The field *toward* indicates where to forward the subscription request. The proxy periodically probes the root to maintain a (proxy–root) address mapping so that requests can be forwarded correctly.

As mentioned above, the dispatcher receives events from the network and puts them into appropriate buffers based on the event header. An event header contains a sender key sid, a receiver key rid, and a field *toward* indicating where to forward the event. The forwarding procedure is simple and similar to Algorithm 1 in the reverse direction. When a Planet receives an event, it checks the *toward* field. If the buffer leading to *toward* contains the sid as the first index, *toward* is updated to be the next stop and the event is enqueued.

3.4.2 Queue reduction

Each queue in a buffer has a limited size and may overflow if its consumption rate is slower than the event arrival rate. Whenever a new event arrives to a full queue, PACK will trigger its PACK policy to reduce the number of events in the queue. For a local buffer, this operation is straightforward, since the second index of the buffer points to a single queue for an individual receiver. The second index of a remote buffer, however, is the link address that points to a queue shared by several receivers over that link. When PACK decides to pack a shared queue, it runs all the events in the queue through each receiver's policy, placing each policy's output in a private queue for that receiver. Note that all the event duplication is based on references, not object instances. Figure 3.5 shows private queues in the lower right.

All newly arrived events are added to the shared queue, which is now empty. The buffer's consumer thread always pulls events from the private queues first and uses the shared queue when all private queues are empty. It is possible that another pack operation is necessary if the shared queue fills up and adds more events to private queues before they are completely drained.

Note that, as PACK splits a single stream of events into multiple unicast streams during network congestion, the forwarding node may end up with more events to deliver. Since the queues continue to be full, PACK will aggressively trigger the receiver's filters to reduce the number of events. A well-designed policy should cope with this situation by having a digester for event summarization. Otherwise, PACK triggers a built-in worst-case policy that drops all events in the queue and summarizes them using COUNT.

3.4.3 Ladder algorithm

When packing an event queue is necessary, PACK must determine which level of filters to apply. Packing at a high level may drop many important events. On the other hand, packing at a low level may not drop enough events, and the time spent packing may exceed the time saved processing or transmitting events. Unfortunately there is no straightforward algorithm for this choice, because there are many dynamic factors to consider, such as the event arrival rate, current network congestion, the filter drop ratio (which depends on values in events), and the receiver consumption rate.

PACK employs a heuristic adaptive approach in which each queue is assigned a specific filtering level, initially one. The heuristic changes the filtering level up or down one step at a time (like climbing up and down a ladder), based on the observed history and current value of a single metric. We define that metric, the *turnaround time* t, to be the amount of time between the current packing request and the most recent pack operation (at a particular level l). The rationale is that changes in t_l captures most of the above dynamic factors. An increase in t_l is due to a slowdown in the event arrival rate, an increase in the departure rate, or an increase in the drop rate of filters up to level l, all suggesting that it may be safe to move down one level and reduce the number of dropped events. A decrease of t_l indicates changes in the opposite direction and suggests moving up one level to throw out more events.

PACK keeps a history of the turnaround time for all levels, t_l, smoothed using a low-pass filter with parameter $\alpha = 0.1$ (empirically derived) from an observation $\hat{t}_l$:

$$t_l = (1 - \alpha)\hat{t}_l + \alpha t_l \ .$$

We define the change ratio of the turnaround time at a particular level l as:

$$\delta_l = (\hat{t}_l - t_l)/t_l \ .$$

To respond to a current event-reduction request, PACK chooses to move down one filtering level to $l-1$ if δ_l exceeds a positive threshold (0.1), or to move up one level to $l+1$ if δ_l exceeds a negative threshold (-0.1). Otherwise, PACK uses the previous level.

3.5 Evaluation

Our implementation is based on Java SDK 1.4.1. We chose Pastry [115] as the overlay routing protocol, but PACK uses its own TCP transport service to disseminate events rather than Pastry's transport library, which has a mixed UDP/TCP mode and its own internal message queues. We used Scribe [27] to maintain application-level multicast trees for PACK to populate the subscription policies.

Since PACK works on the queues accumulated above TCP (namely, after a sender's TCP buffer is filled), the events in the TCP sending buffer are not accessible to PACK and they may be blocked until they get through to the other end. Rather than developing a customized protocol to replace TCP, we limit TCP's send buffer size (to 1024 bytes) to diminish TCP's overhead for now. Ultimately it may be best to replace TCP with UDP and extend our transport service to handle event (packet) retransmission and congestion detection, which also may relieve the problem of events lost in the TCP buffer if the client disconnects and moves without an explicit request.

Next we present some experimental results from the PACK service, using the Emulab testbed at Utah.[1] In all the tests we turned off the just-in-time compiler and garbage collector in the Java VM. We focused on measuring the performance of the PACK buffer.

3.5.1 Queueing tradeoff

To measure the queueing behavior when a policy is triggered, we used Emulab to set up two hosts connected by a 50Kbps network link. We placed a single receiver on one host, and a single sender and an overlay node on the other. The sender published an event every 30ms, and the events accumulated at the overlay node due to the slow link to the receiver. We compared two approaches to drop events when the queue fills: one is to drop the new event, simulating "drop-tail" behavior, the other is to use a three-level PACK policy. Each level of the policy contains a single filter, randomly throwing out events (10%, 25%, and 50% respectively). We show the results in Figure 3.6.

Figure 3.6(a) shows the latency perceived by the receiver. After the buffer filled up, events in the DropTail queue have a (nearly constant) high latency because each event has to go through the full length of the queue before transmission. On the other hand, events in the queue managed by the PACK policy exhibit lower average latency because events may be pulled out of the middle of the queue, so other events have less distance to travel. From these results, it is clear that application designers should use filters that are more likely to drop events in the middle (such as EVERY, RANDOM, GUNIQ) rather than at the tail.

Figure 3.6(b) plots a running sequence of the event loss rate for each 1 second window at the receiver. We see that the DropTail queue's loss rate was about 30% because the arrival rate was one third more than the bottleneck link could handle, and after the queue filled it was always saturated. The loss rate of PACK was high during intervals when the queue was packed, and zero in intervals when the queue was not packed. The loss rate depended on which level of pack operation was performed. Figure 3.6(c) shows a trace from the overlay node denoting when the queue was packed and what fraction of events were dropped. It shows that most pack operations were performed at the second level, dropping events at rate of $0.1 + 0.9 * 0.25 = 0.325$, which fit well with this event flow because the arrival rate was one third higher than the consumption rate (link bandwidth). The filtering level varied, despite the steady publication rate, because the RANDOM filter dropped varying amounts of events and our heuristic adapted to longer or shorter inter-packing intervals by adjusting the filtering level.

[1]http://www.emulab.net/

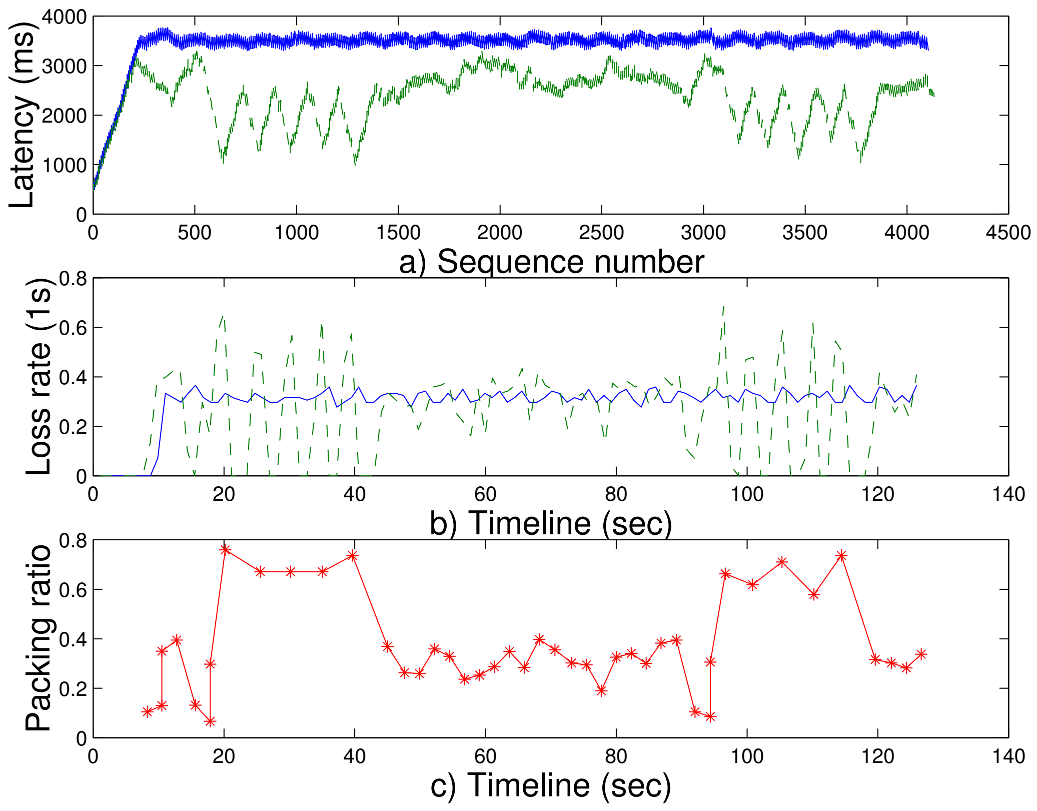


Figure 3.6: Comparison of queueing behavior of event reduction using DropTail (solid line) and a three-level PACK policy (dashed line).

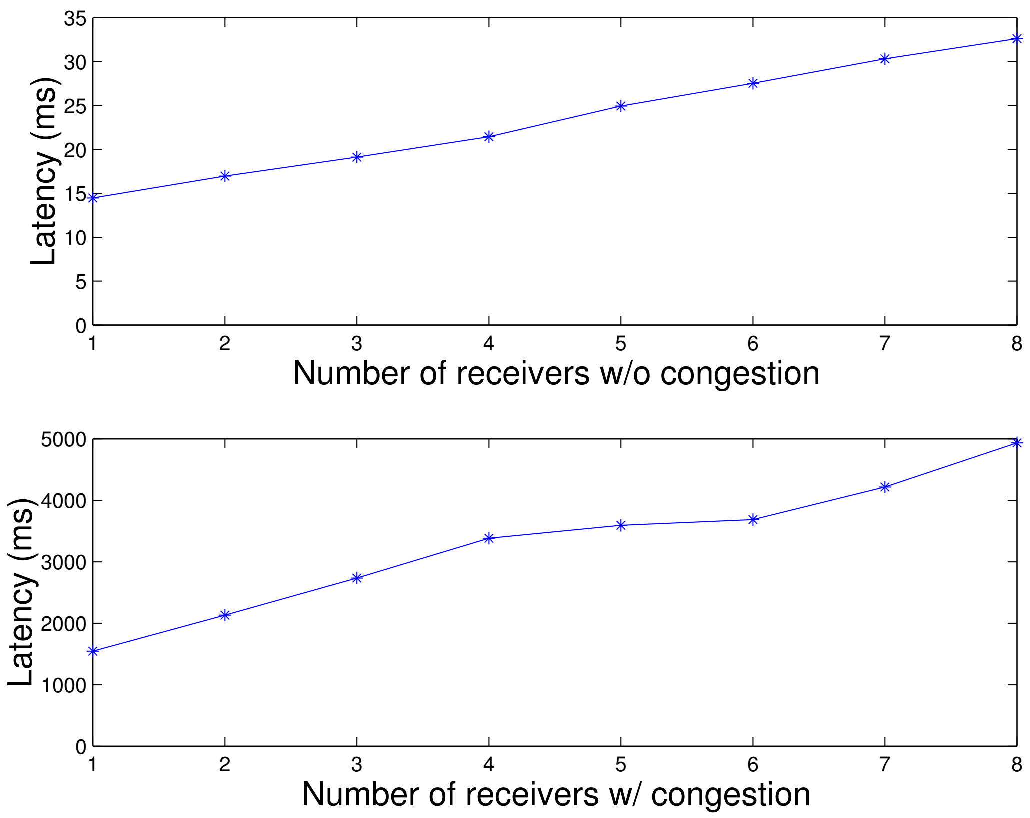


Figure 3.7: Buffering overhead measurements over non-congested link (top plot) and congested link (bottom plot). Note the different scales.

3.5.2 Buffering overhead

We also measured the overhead posed by the two-level indexing buffer structure (Figure 3.5). Again we used Emulab to set up a sender client and an overlay node on the same host, with multiple receiver clients on other hosts. We first connected the overlay node and the receivers with a 100Mbps LAN with the sender publishing events at a 200ms interval. In another test we connected the overlay node to receivers using a 50Kbps link while the sender publishing events every 30ms. In the first setup, the overlay node's buffer has multiple entries on the second index but the queue never filled up and no policy was triggered. In the second setup, the shared queues overflowed and the buffer created private queues for each individual receiver and triggered their policies due to the restricted link. All receivers used a 3-level policy that dropped a certain fraction of events from the tail of the queue (10%, 25%, and 50% respectively). In both setups, we measured the event delivery latencies at all the receivers and show the averages in Figure 3.7.

The first plot in Figure 3.7 shows that, as the number of entries in the second index increases, the average latency perceived by all the receivers also increases linearly. This result indicates that to build a large-scale PACK multicast tree, the output degree of each node has to be relatively small although collectively the overall tree may have many leaves (receivers). The second plot shows a worst case, in which a network congestion forced a multicast stream to split into several substreams with individual policies. Although still a linear increase, the added latency perceived by the receivers

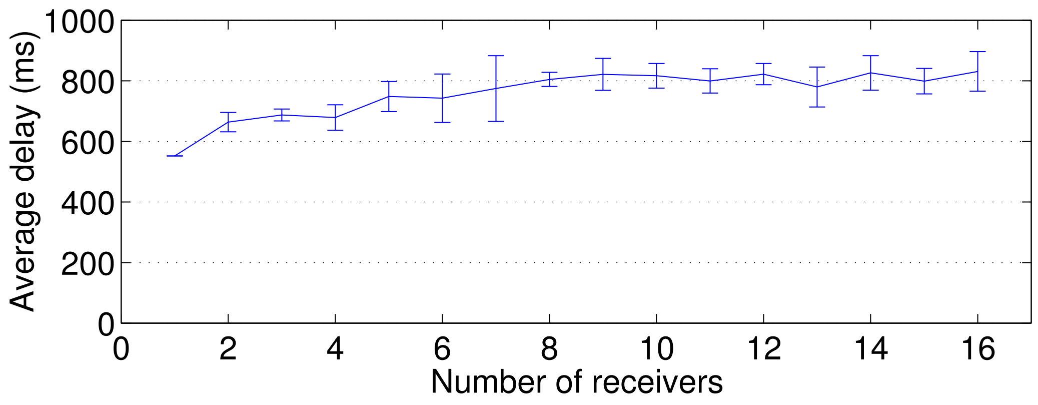


Figure 3.8: Measurements of client detach/attach delay.

is a non-trivial overhead. In addition, the space required by the private queues (event references) also relates to the number of receivers across the congested link.

If the congestion occurs at the higher part of the dissemination tree, which we expect to happen less frequently than at the network edges, the buffer may have to manage many policies. The copying of events into multiple private queues consequently causes many events to be dropped; those events that do arrive may experience long delays. In other words, the PACK service itself does not try to prevent or relieve the congestion. Instead, it reduces data for each individual receiver, who may use the summaries as a congestion indication and decide to cancel the subscription if the congestion persists. This gives us advantages over other approaches, such as TCP, that blindly push back on the sender, whose queue may eventually overflow and crash while the receiver has no method to determine the network conditions for adaptation.

A fundamental issue, however, is that PACK pushes arbitrary application-specified policies into the network; this flexibility restricts scalability during congestion since each overlay node has only limited resources for all the policies. Carzaniga and others discuss the tradeoff between expressiveness and scalability in a similar context [22]. One approach to relieve the situation is to limit the flexibility of PACK policies. For instance, RLM essentially uses a set of hierarchical filtering layers that apply naturally to multimedia data streams [89].

3.5.3 Client attach/detach

As a mobile client detaches from and re-attaches to its proxy, PACK suspends and resumes its event queue in the RPB buffer (located at RR). To measure how this operation scales with many moving clients, we again set up one sender and one overlay node on one LAN host in the Emulab topology, and varied the number of receiver clients (distributed evenly across the other four LAN hosts). Each client had one endpoint. Each client explicitly repeated the operations of attaching to and detaching from the overlay node 20 times, while waiting 5 seconds before each state transition. We measured the delay from the client-issued "attach" request until the first buffered event arrived, and Figure 3.8 shows that the average delay was less than one second. This latency is important because it directly affects the user experience in many applications.

While the average delay clearly grew as the number of receiver clients increased, there was a large variance of the delay across receivers. We saw a similar wide variance across the 20 requests within a single receiver. We believe that this variance was due to thread scheduling and synchronization effects in the Java VM. The detach/attach requests, implemented by the RPC service, were handled by the proxy using a small thread pool. Also, the RPB buffer had one queue for each client; each had a consumer thread that transmitted events across the TCP connection to a client. The threads compete for the network since every queue in the RPB buffer had events buffered during client disconnection. This competition may be the more significant effect since Figure 3.7 shows little latency variation under a light load.

3.5.4 Application tests

As an example application, we use PACK to monitor a campus-wide wireless network. Our campus is covered by more than 550 802.11b access points (AP), each configured to send its syslog messages to a computer in our lab. We run a data source on that host to parse the raw messages into a more structured representation and to publish a continuous event stream. By subscribing to this syslog source, applications can be notified when a client associates with an AP, roams within the network, leaves the network, and so on.

One of our goals is to provide an IP-based location service: given a wireless IP address, the service can identify the AP with which the device is currently associated. This enables us to deploy location-based applications, often without modifying legacy software. Figure 3.9 shows a Web proxy, modified from an open-source Java proxy,[2] which is able to push location-oriented content to any requesting Web browser on wireless devices based on the IP address in the HTTP header. Currently we insert information about the building as a text bar on top of the client requested page. Similarly, a location-prediction service could instruct a Guide application [42] on a mobile device to prefetch content based on the next likely stop.

To provide this kind of service, a locator subscribes to the syslog source and monitors all devices' associations with the network. An association message contains the device's MAC address and associated AP name, but does not always include the IP address of that device. In such cases, the locator queries the AP for the IP address of its associated clients using a HTTP-based interface (SNMP is another choice, but appears to be slower). The query takes from hundreds of milliseconds to dozens of seconds, depending on the AP's current load and configuration. We also do not permit more than one query in 30 seconds to the same AP so our queries do not pose too much overhead over normal traffic. As a result, we frequently find that the locator falls behind the syslog event stream, considering the large wireless population we have.

We focus our discussion on the subscription made by the locator to the syslog source, where the events tend to overflow the receiver's queue RB. The locator uses a 6-level set of filters. Some of them could be chained on the same level, but we chose to separate them for easier tracing. These filters are listed as follows (the policy is not shown to save space):

1. EQ: retain only events whose *message type* is "Info";
2. INSET: discard certain events such as "Authenticated" or "roamed";
3. MATCH: discard the events whose *host name* represents an AP instead of mobile clients;
4. FIRST: retain only the first event whose *action* is any of the four messages indicating the clients' departure from the network;

[2]http://muffin.doit.org/

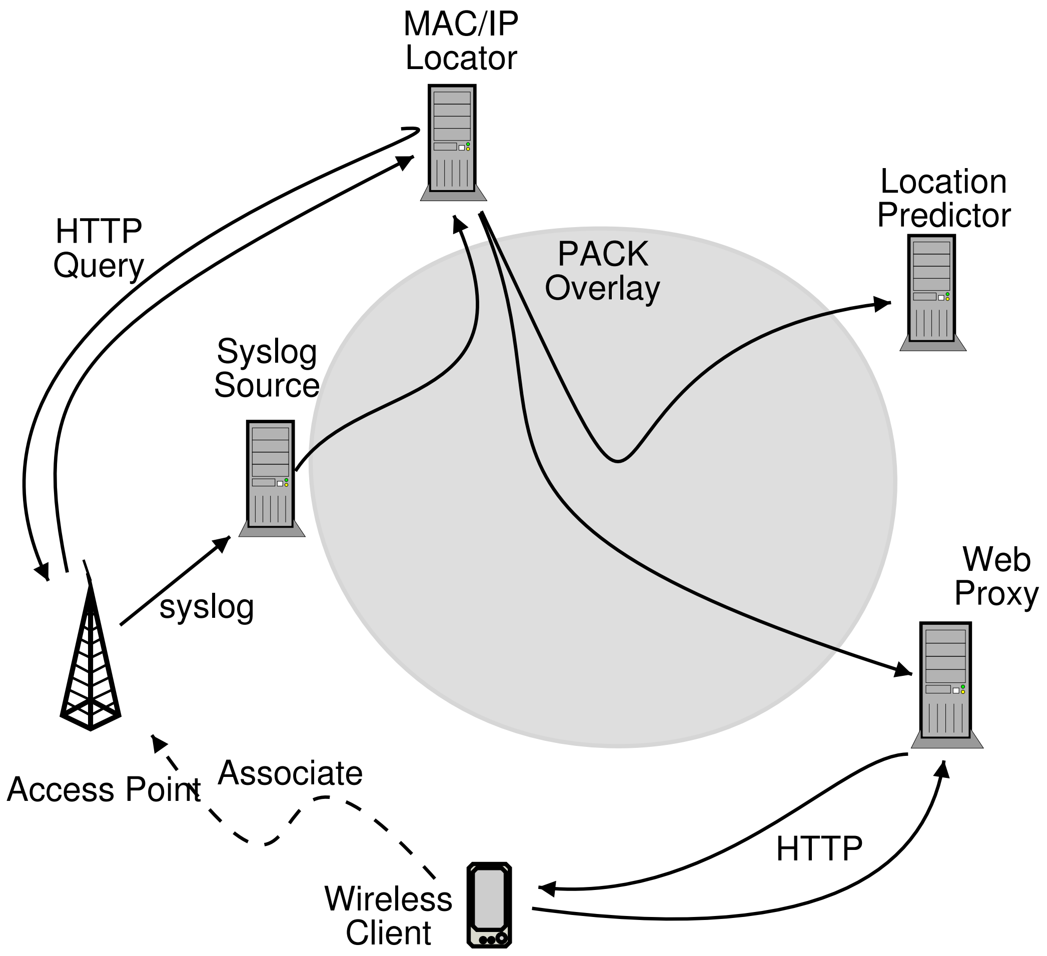


Figure 3.9: The setup for a campus-wide WiFi locator and its applications.

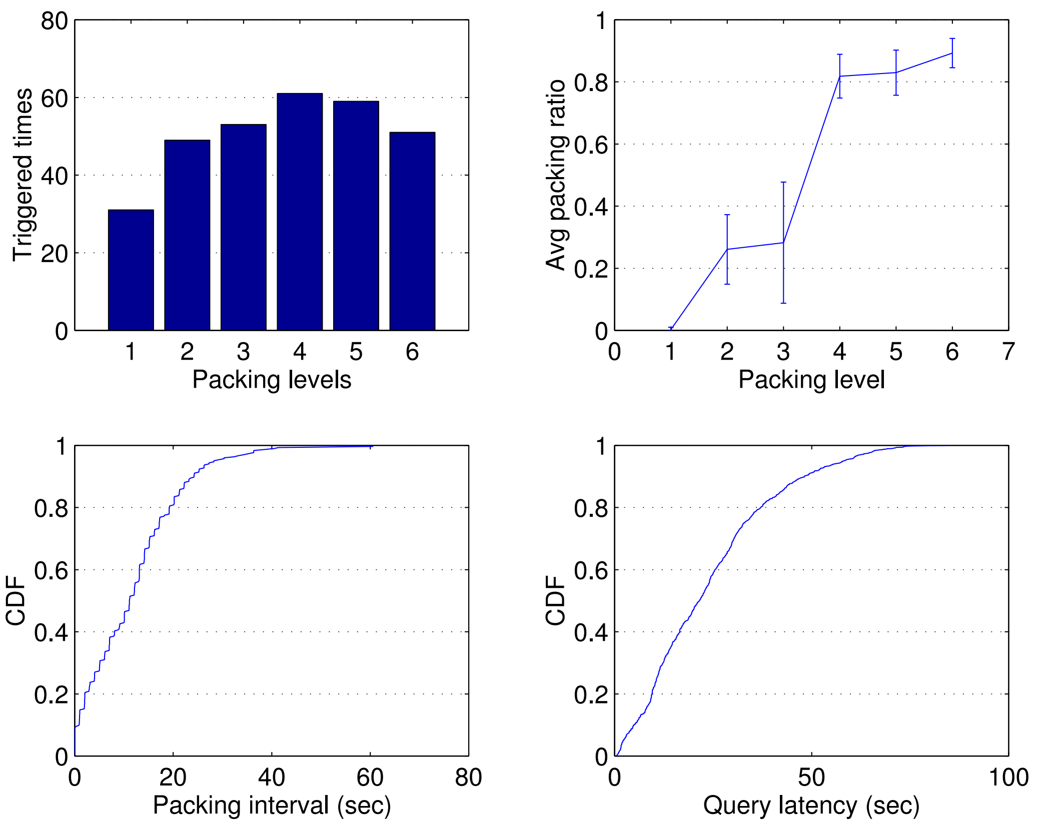


Figure 3.10: Statistics derived from the PACK trace collected on behalf the MAC/IP locator with a 6-level filtering policy.

5. GUNIQ: remove all events with duplicated *AP name* except the first one (see the optimization discussed below);

6. EVERY: drop one event out of every three.

To accelerate the query performance, we made two optimizations to the locator. First, we do not query the AP if the syslog event already contains an IP address for the client. Second, when querying the AP we retrieved the list of all its associated clients and cached the results to speed up lookups for other clients. We collected the PACK trace for a hour-long run and Figure 3.10 shows some basic statistics.

The upper-left plot presents the distribution of the filtering levels triggered by the PACK service. All filtering levels were triggered, varying from 31 times to 61 times, out of 304 pack operations. The upper-right plot shows that the filters had a wide variety of packing ratios over that one-hour load. It seemed that filters 2 and 4 discarded most of the events while filters 1, 3 and 5 did not help much. This suggests strongly that an application programmer should study the work load carefully to configure more efficient policies. The lower-left plot indicates that PACK triggered the policy rather frequently, with the median approximately 11 seconds. The lower-right plot shows the latency, derived by the time the query is resolved and the timestamp in the original syslog event. Although we set the connection timeout to be 30 seconds for each poll, the longest delay to return a query was 84 seconds suggesting some AP was under heavy load and slow to return results even after the connection was established.

The locator could adapt to situations where level 6 is frequently triggered by creating multiple threads for parallel polling, so fewer events (which might be association messages) might be dropped. We are currently reluctant to take this approach since the downstream application may want in-order event delivery. The location predictor, for example, is sensitive to the sequence of moves.

We note that filters 1, 2, and 3 throw out events having no value to the locator service. If the source supports filtered subscription, none of those events need to be transferred across the network. The source, however, might become the bottleneck as the number of filters to run increases. Rather than using PACK as a filtering system, we believe a more general infrastructure is necessary, such as a content-based event system with built-in (limited) filtering or a data composition network supporting a more powerful language [44]. PACK complements these systems to deal with buffer overflow issues.

3.6 Related work

The design choices made by PACK generally follow the principle of Application-Level Framing [43]. The data manipulation and transfer control are based on Application Data Units (ADU). In our case, pack operations are performed on the queued data units with a particular structure. On one hand, it is simplest to drop the recent ADU when a queue is about to overflow. On the other hand, this policy is inadequate or even incorrect for many applications with different requirements. Although this flexibility could be implemented at both sender and receiver, it is not easily deployable to intermediate IP routers. Thus an application-level overlay infrastructure is attractive since we can push the “packing” function closer to congestion and disconnection to improve scalability and responsiveness.

Traditional congestion and flow control protocols concern both unicast and multicast. They are typically transparent to applications and provide semantics such as reliable in-order data transport. When computational and network resources are limited, these protocols have to either regulate the sender’s rate or disconnect the slow receivers [71, 104]. The usual alternative, UDP/IP, has no guarantees about delivery or ordering, and forces applications to tolerate any and all loss, end to end. Our goal, on the other hand, is to trade reliability for quicker data delivery and service continuity for loss-tolerant applications. Our PACK service applies to data streams with a particular structure. This loss of generality, however, enables PACK to enforce receiver-specified policies. The PACK protocol does not prevent or bound the amount of congestion, which is also dependent on cross traffic. But with an appropriate customized policy, a receiver is able to get critical data or summary information during the time of congestion or the recovery period. For many applications this outcome is better than a strict reliable service (TCP) or a random-loss (UDP) service.

Performing application-specific computation, including filtering, inside networks is not a new idea. In particular, it is possible to implement our PACK service using a general Active Network (AN) framework [127]. We, however, chose an overlay network for its flexibility of placement and easier deployment. Based on AN, Bhattacharjee and others propose to manage congestion by dropping data units based on source-attached policies [14]. Receiver-driven layered multicast (RLM) [89] actively detects network congestion and finds the best multicast group (layer) for the multimedia application to join. Pasquale et al. put sink-supplied filters as close to the audio/video source as possible to save network bandwidth [100]. The Neem protocol removes the "obsolete" messages, as indicated by the source, or random messages when its queue becomes full [95]. Our work, however, aims at broader categories of applications and must support sink-customized policies since the source typically cannot predict how the sinks want to manipulate the sensor data. PACK policies thus need to be more expressive than the filtering operations on multimedia streams. Our protocols also explicitly support disconnect operations caused by end-host mobility.

Data aggregation also is a useful technique inside sensor networks to reduce unnecessary transmission, such as TAG [86]. While designed for different purposes, both TAG and PACK try to enforce application-specific policies. The goal of TAG is to apply the aggregation wherever and whenever possible in the sensor network, while PACK policies are only triggered when the buffer starts to overflow. These two systems are complementary since the aggregated results coming out of a sensor network could supply one data stream as a Solar source.

Recent work using an overlay of event brokers to provide a content-based pub/sub service has focused on routing and matching scalability and has largely ignored end-to-end flow control [10, 23]. Pietzuch and Bhola, however, study the congestion-control issues in the context of the Gryphon network [102]. Congestion in the whole system cannot be solved by simply interconnecting nodes with TCP because the overlay is constructed in application space above TCP. Their solution is to apply additional protocols for end-to-end reliability for guaranteed event delivery. The sender (or the broker serving the sender) then has the responsibility to store all the events during congestion for later recovery, such as using a database. From the application's point of view, their protocols are no different than traditional approaches, and there is no explicit support for mobile clients.

Receiver-driven layered multicast (RLM) [89] leverages the fact that multimedia streams can be encoded in different layers (rates), each of which requires different bandwidth. The receivers then join only the multicast group (corresponding to layer or encoding rate) that best matches available network capacity. In a way, this idea is similar to the PACK service, which enforces receiver-specified policies. RLM, however, focuses only on multimedia applications, works at the packet level, and requires IP multicast. PACK is built in application space and requires no special capability in an IP network; it uses the same "packing" mechanism for flow control (managing the queues at end hosts) and congestion control (managing queues in overlay nodes); and it provides explicit support for mobile clients (either data sources or sinks). On the other hand, PACK needs a deployed overlay infrastructure, and PACK requires more application programmer effort since layer selection is transparent in RLM and requires no explicit application policies.

Researchers in the database community provide a query-oriented view on continuous stream processing. One of their foci is to formally define an SQL-like stream-manipulation language, which has the potential to replace PACK's current "ad-hoc" XML-based interface. In particular, the Aurora system reduces the load by dynamically injecting data-drop operators in a query network [126]. Choosing where to put the dropper and how much to drop is based on the "QoS graph" specified by applications. Aurora assumes complete knowledge of the query network and uses a pre-generated table of drop locations as the search space. The QoS function provides quantitative feedback when dropping data, while PACK allows explicit summarization of dropped events.

Chapter 4

Naming and Discovery

To compute the desired context, Solar applications typically select some existing data sources and compose them with some operators into an operator graph (Chapter 2). Solar provides a naming service for sensors and optionally some deployed operators to register a *name advertisement*. Solar stores the advertisements in distributed directories to improve scalability, and applications use a *name query* to find particular data sources.

A name advertisement for a data source should be a descriptive handle, including all information necessary to distinguish it from other sources. An alternative approach is for sources to describe the type of data they provide, and for applications to choose sources based on the desired data type. Often, however, much of the descriptive information about a data source naturally resides outside of its data-type space, and it is more flexible and expressive to allow applications to select sources based on such meta information. For example, all of the temperature sensors in a building output the same data type, but should be distinguished by the *location* of the sensor.

In addition to typical `advertise` and `query` interfaces, Solar's naming service also supports persistent queries and context-sensitive advertisements and queries. We believe that our contributions about context-sensitive resource discovery are critical for dynamic and volatile pervasive-computing environments, so we have implemented the service as a wrapping layer to leverage any directory service.

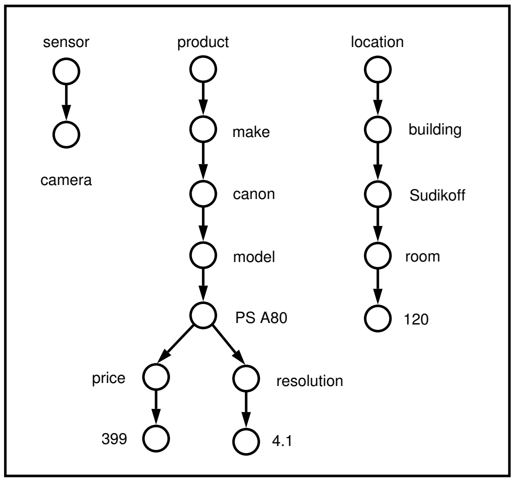
We first present the representation structure used for name advertisements and queries. We then discuss the details of our context-sensitive resource discovery framework and its evaluation, using Intentional Naming System (INS) as the core directory service [3]. Finally we briefly present, in Section 4.5, a newly designed distributed directory service that can replace INS for improved scalability.

4.1 Naming representation

The name space of the data sources could be organized as a tree, as in many file systems. For those sources given names, the name describes a path from the root to a leaf in the tree. For example, a temperature sensor in Sudikoff room 215 might be named

```
[/Sudikoff/2F/215/temp-sensor].
```

To enhance scalability, multiple name trees may be federated; perhaps the most common example is the two-level name (hostname:filename) used in URLs. There are alternative naming architectures



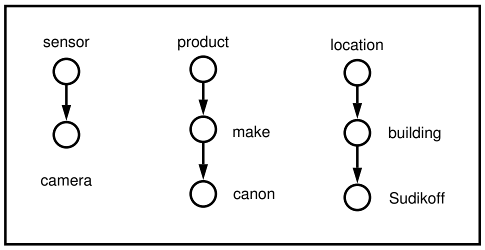


Figure 4.1: A camera advertisement on the left and a query on the right to find all the Canon cameras in building Sudikoff.

with less structure than a tree. Each named data source could be given a set of descriptive attribute-value pairs [64]. The above temperature sensor might be named

`[sensor=temperature, room=215, floor=2, building=Sudikoff].`

It is arguable whether one approach has clear advantages over the other [35]. In either case the name should be a descriptive handle. In one case the description is a tuple of attributes and values, and in the other case the same attributes may be implicit in the structure of the tree. Both depend heavily on conventions that define the names of the attributes (or structure of the tree) and the range of values (or names of tree links). While the hierarchical tree structure produces concise names and is easy to traverse and explore, the conventions used to structure the tree are likely stricter and harder to extend than those in a set of attributes, which may make the tree less attractive in a dynamic pervasive-computing environment.

Another important role for naming is to facilitate resource discovery. In tree-based names a wildcard allows an application to easily describe a large set of publishers, e.g.,

`[/Sudikoff/*/*/temp-sensor/].`

The same effect might be obtained in an attribute-based system that allows partial matches, e.g.,

`[sensor=temperature, building=Sudikoff].`

While some other operations (such as range selection) might be achievable using tree-based names, the syntax could be awkward.

Solar uses a hybrid approach. The data structure for Solar's naming scheme is a *record*, which is a set of *attributes*, each of which is a tag-value pair. The tag of an attribute is a string, and the value of an attribute is either a string or another record, which leads to an attribute hierarchy. Each name advertisement and each name query contains a record, which is a forest if we consider the tag to be the parent of the value in an attribute. A name advertisement and a name query are shown in Figure 4.1.

We call a path from a root to any non-root node in the name advertisement or query a *name strand*. For instance, in Figure 4.1 `sensor=camera` is a short strand and a longer strand is `location=building=Sudikoff=room=120`. We say that an advertisement matches a query if the query's strands are a subset of the advertisement's strands. For instance, the advertisement in Figure 4.1 matches the given query on the right.

Since the directory may return data sources for a query, the application may use a customized function to select appropriate sources. For instance, a location-aware application may want to use the location service with maximum granularity or fastest update rate. When data sources come and go, as is typical in a pervasive-computing environment, the results of the query change occasionally; the function is re-evaluated, permitting quick adaptation for the applications. We discuss the specification of names and queries in the next section.

4.2 Context-sensitive resource discovery

Pervasive-computing applications must discover and use resources based on the current context. Imagine a nursing home equipped with networked cameras and sensors that can track the location of residents. A "SafetyCam" application can track a person's location and automatically retrieve the video stream from a nearby camera. The cameras may be named according to their location, and the dynamic location information of the senior is used to identify appropriate cameras. This scenario requires a context-sensitive name query.

In another situation, the camera may be mobile. At the scene of a disaster, rescue workers might wear helmets with small attached cameras and a wireless network interface. If these cameras are named according to their location, a supervisor's monitoring application can request photographs of a particular area by selecting cameras whose location (in advertisement) matches the area of interest. The display automatically adjusts when a rescuer moves into or out of that place. This scenario requires context-sensitive advertisements, which may change over time, and persistent name queries, so that the application is notified about the changing set of matching advertisements.

These scenarios place several requirements on the naming service. It must be flexible, so advertisements can characterize the resource and so queries can express the desired characteristics; it must be scalable, to handle many advertisements and queries; it must be fast, to support frequent advertisement updates; and it must be responsive, to quickly notify applications about changes to the set of matches for their persistent queries.

These scenarios also place several requirements on the resources and applications. Resources must actively track their context so that they may update their advertisement. Applications must also track their context so that they may update their query. We off-load these duties from the resources and applications, for reasons of performance (since resources and applications may reside on a constrained platform attached to a low-bandwidth network) and of engineering (to simplify the construction of context-aware services and applications).

One approach is to build a *context service* in the infrastructure, which is responsible for finding the appropriate sources, collecting the necessary data, and deriving the desired context [48]. The resources and applications then can subscribe to (register a callback with) the context service. When notified of changes in the context, resources contact the name service to update their name, and applications contact the name service to issue a new query. It is possible to integrate the context service and the name service into a single context-sensitive directory service [84].

Solar's naming service, on the other hand, reuses our context-fusion infrastructure to help resources to make context-sensitive advertisements, and to allow applications to make persistent and context-sensitive queries. Later in this section, we show a simple specification language used to describe context-sensitive advertisements and queries. We also present how Solar off-loads the task of updating and monitoring the name space, and permits a decentralized and scalable implementation to support context-sensitive resource discovery.

Our naming service makes two contributions: a) an extension to a typical advertisement-based resource-discovery mechanism that supports context-sensitive advertisement and context-sensitive queries, and b) a distributed infrastructure that efficiently supports both one-time and persistent queries in such a name space. Our infrastructure is designed to leverage existing distributed directory services; for our prototype we built the distributed directory using INS [3], but any other service providing an attribute-based registration and look-up interface would suffice (Section 4.5).

Since one of our goals is to reduce load on thin-client devices and on the low-bandwidth networks that serve them, we embed the service in the infrastructure; thus our techniques are not readily applicable to an infrastructure-free (ad-hoc) environment. Security and privacy issues are beyond the scope of our research focus; interested readers may find more information in another paper [94].

4.2.1 Name specification

Solar provides a light-weight specification language that can be used to specify context-sensitive advertisements and queries. The idea is to define some attribute value, in an advertisement or query, as context that is dynamically derived from another operator graph. As the context changes, the advertisement or the query is updated automatically inside the Solar infrastructure. In the specification

```
[sensor=camera, room=$alice-locator:room, building=Sudikoff]
```

the value of the "room" attribute is defined by context information derived from an operator called `$alice-locator`. Every context-sensitive name specification must be accompanied by an operator graph that defines the desired context computation.

Figure 4.2 demonstrates these concepts using the nursing-home example. At top left, a location source advertises a static (not context-sensitive) name and publishes location events about all residents. Its subscriber, a filter operator, discards events not pertaining to user Alice. The event (in italics) indicates that Alice is in room 120. At lower right, the SafetyCam application uses the context-sensitive query

```
[sensor=camera, room=$alice-locator:room, building=Sudikoff]
```

to identify and subscribe to a camera source near Alice. If `$alice-locator` refers to the event stream produced by the filter, then `$alice-locator:room` is resolved by the filter's events. As shown, the `room` attribute of the query is resolved to be "120", which matches the advertisement of the camera source in the lower left.

To arrange the context-sensitive subscription depicted in Figure 4.2, SafetyCam uses the operator-graph specification shown in Figure 4.3. Note that here we encode the operator graph with a customized language. It should be easy, however, to convert it to an XML-based notation (Chapter 2). The graph specification contains two parts, `define` and `load`. The `load` section provides the URL and class name of the Java classes for any non-standard operators used in the definition section, here `@userFilter`. The `define` section contains a sequence of statements, each of which defines an operator. Each statement defines the subscriptions for its operator, and together the statements determine the graph structure.

The first statement defines an operator that subscribes to any one of the publishers whose name matches the specification in brackets. The statement assigns the variable name `$locator` to that

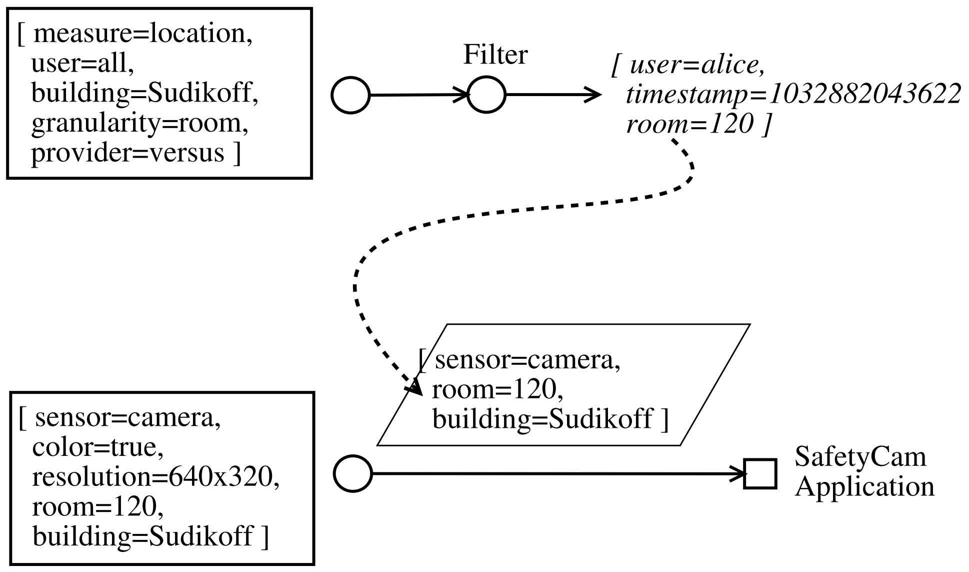


Figure 4.2: A nursing-home example shows the use of the output from an operator graph to select an advertisement for the SafetyCam's subscription.

```
define
{
  $locator := @relay <- ( @any  [ measure=location, user=all,
                                            building=Sudikoff ] );
  $alice-locator := @userFilter ("alice") <- ( $locator );
  $cameras := @merge <- ( @all  [ sensor=camera, building=Sudikoff ],
                                            room=$alice-locator:room ] );
}
load
{
  @userFilter at ("http://codebase/", "solar.operators.UserFilter");
}
```

Figure 4.3: Example graph specification to calculate Alice's current location and how it is used to define a context-sensitive subscription.

```
// application subscribes to  a specified operator graph
// or to a context-sensitive name defined by the graph
subscribe(String graph-spec, String name-spec);

// resource adverties a context-sensitive name (or static if graph-spec
// is null) to the directory.
advertise(String graph-spec, String name-spec);

// application makes one-time (or persistent if persist is true) context-sensiti
// name query (or static if graph-spec is null).
query(String graph-spec, String name-spec, boolean persist);
```

Figure 4.4: The set of Solar naming API methods.

operator's event stream. The syntax `@func [name spec]` identifies the desired subscriptions, using the function `@func` to select among the set of names matching the name specification. Selection functions `@any` and `@all` are built-in; the user also may define custom selection functions.

The second statement defines an operator that filters, transforms, or otherwise aggregates the events in its subscriptions. The statement identifies a composition function and its parameters; here, the user-defined function `@userFilter` takes one parameter, "alice", and the name of an input event stream, `$locator`. The filter discards any input event that does not contain attribute `user=''alice''`. The result is a stream of events containing Alice's current location; the event stream is called `$alice-locator`.

The third statement defines an operator `$cameras`, which simply merges all photo events received from camera sources co-located with Alice. This merge operator has a context-sensitive subscription, using the built-in selector `@all` and the context-sensitive name specification mentioned earlier. The events in the stream `$alice-locator` determine the subscriptions of this merge operator.

The SafetyCam application may receive the stream of photo events by simply providing this graph specification (see the *subscribe* interface in Figure 4.4); since `$cameras` is the root of the resulting operator graph, the SafetyCam receives its events. The Solar system deploys the operators, arranges the subscriptions, and actively monitors the context-sensitive name specifications to adjust subscriptions as necessary. Notice that Figure 4.2 is a conceptual diagram and does not contain the relay and merge operators defined in Figure 4.3; the purpose for these operators will become clear below where we discuss operator-graph deployment.

The above example demonstrates the use of a context-sensitive name specification to support a context-sensitive subscription request from the SafetyCam application. A similar graph specification (without the `$cameras` definition) also could be used by an application that simply wishes to query the name service for a list of camera sources near Alice, using the name specification

```
[sensor="camera", room=$alice-locator:room, building="Sudikoff"]
```

as a context-sensitive query. The application could ask once to receive the current list of matching names, or it could register a *persistent query*, and be notified any time the set of matching names changes (see the *query* interface in Figure 4.4).

In another situation, suppose Alice carries a camera that provides a photo-capture source. Her camera source provides the same graph specification and the above name specification as an advertisement; the result is that the camera has a context-sensitive advertisement (see the *advertise* interface in Figure 4.4).

4.3 Implementation with INS

A Solar data source may contact any Planet with a request to advertise a name, providing a graph specification if the name is context-sensitive (see the *advertise* interface in Figure 4.4). The Planet creates an internal object as a proxy for the source, and the proxy internally registers that name. Subscribers naming that source actually subscribe to the proxy, which forwards events from the source to all subscribers.

An application may contact any Planet with a name query, providing a graph specification if the query is context-sensitive (see the *query* interface in Figure 4.4). The Planet creates a proxy object to service the query. An application also may contact any Planet with a graph specification if it wishes to subscribe to an event stream (see *subscribe* interface in Figure 4.4). The Planet deploys the desired operator graph and creates a proxy object to serve the application, which subscribes to the operator graph and forwards any received events.

A proxy is responsible for managing the subscriptions on behalf of its client, for managing context-sensitive advertisements and queries, and for forwarding events. The use of proxies allows us to manage subscriptions and names entirely inside the Planet, which is reliable and well-connected, rather than on the host of the data source or application, which may be slow or poorly connected.

Internally, we build Solar's name service on top of a generic directory service, using Solar operators to obtain, process, and monitor the necessary context information. This layered approach allows us to leverage existing research on (and implementations of) scalable, distributed, and flexible directory services. In particular, we used the Intentional Naming System (INS) to implement the directory service [3]. To Solar we add INS resolvers, as shown in Figure 4.5. The shaded circle is an operator that may process events from the source to provide context for an application's query or a source's advertisement. The ovals marked with P are proxies representing connected Planet clients. Arrows represent (1) context-sensitive advertisement; (2) context-sensitive query; (3) contextual events; (4) actual query based on current context; and (5) actual advertisement based on current context.

4.3.1 Extending INS

INS is a resource-discovery and communication system [3]. In conventional networks, a name service resolves names to addresses, then routers route messages to destination addresses. In INS, a distributed collection of *resolvers* form an overlay network that routes messages to destination names. Thus, INS combines name resolution and message routing into a single abstraction.

To receive messages, an application "advertises" its name by announcing it to any INS resolver. The resolvers disseminate the name throughout the resolver network. Name records are discarded as they age, so an application must re-advertise the name periodically.

An application may send an intentional *anycast* message to a given name *pattern*; the resolvers route the message to any destination with a matching name. [Names and patterns are sets of hi-

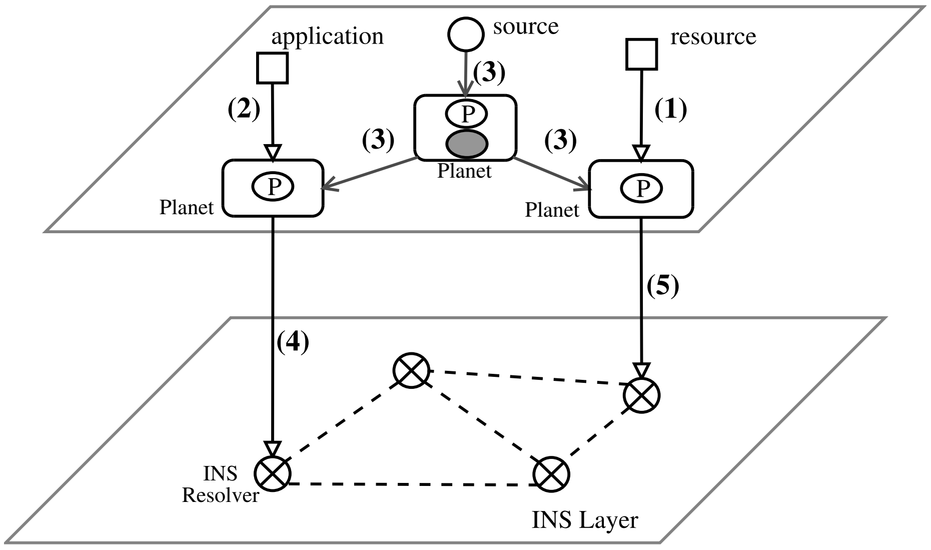


Figure 4.5: System architecture with both the Solar and INS layers.

erarchical attribute-value pairs; a pattern matches a particular name if all attributes of the pattern are present in the name, and the values of corresponding attributes are equal.] An application also may send an intentional *multicast* message to a given pattern; the resolvers route the message to *all* destinations with matching names.

INS clients can request a list of advertised names that currently match a pattern. There is no mechanism, however, for a client to register a callback so it can be notified when a new matching name arrives, or an old matching name disappears. To support name queries, the directory service must match a new pattern against existing names; INS has that capability. To support *persistent* name queries, new names must be matched against existing patterns; INS does not have that capability.

When routing a message, INS delivers a message to a destination if the message is tagged with a pattern that is a subset of the destination's name. We extended INS to support superset matching in addition to subset matching. Messages tagged with [_type=name] use the default subset matching, and messages tagged with [_type=pattern] use our new superset matching.

We currently use a simple superset matching algorithm. The goal is to obtain a set S of the patterns p that match the given name n. Starting with an empty set S, we add each pattern p that matches n in any attribute. Then we reduce S by discarding all patterns p that do not match n in all attributes of p.

Next, we show how Solar transparently adds these tags and uses the new feature to support persistent queries.

4.3.2 Using INS

For each use of a name specification, the Planet creates a *monitor* object to interact with INS. The monitor is attached directly to the object associated with the name: the operator (or proxy) advertising a name, or the operator (or proxy) requiring a persistent name query.

Consider "static" names, those that do not depend on context. Suppose an application wishes to subscribe to all sources whose name matches the pattern [sensor="camera"], and sends a simple graph specification describing such a request to a Planet. Upon parsing the graph specification, the Planet creates a proxy and an associated monitor. The monitor converts the Solar name into INS syntax [sensor=camera] and uses INS in two ways: a) it asks INS for a list of existing names that match pattern [sensor=camera][_type=name] and returns to the subscriber these names in Solar's format, and b) it creates an announcer thread to advertise the INS name [sensor=camera][_type=pattern]. Later, when a new publisher asks its proxy to advertise [sensor="camera", color="true"], the monitor associated with that proxy uses INS in two ways: a) it creates an announcer thread to advertise a name [sensor=camera][color=true][_type=name], and b) it sends an intentional multicast to [sensor=camera][color=true][_type=pattern] with a payload indicating "new name." The purpose of the "_type" attribute is to allow new patterns to find names, and new names to find patterns.

Now consider a context-sensitive name specification; over time, the name may change. Solar deploys an operator graph according to the accompanying graph specification, and subscribes the name's monitor to the resulting event stream. The monitor receives each event and re-computes the value of the name, substituting concrete values. For example, [sensor="camera", room=$locator:room] becomes [sensor="camera", room="215"] upon receipt of an event with attribute room="215" from the $locator event stream.

For a persistent context-sensitive name advertisement, when the monitor detects that the query specification has changed due to a new value for some context-sensitive attribute, it a) sends a special meta-event to the operator's subscribers, indicating the name change; b) sends an INS intentional multicast to the "_type=pattern" form of the old name, indicating the name change; c) tells INS to unadvertise the "_type=name" form of the old name; d) tells INS to advertise the "_type=name" form of the new name; and e) sends an intentional multicast to the "_type=pattern" form of the new name. The result is to inform current and future subscribers that the old name is gone and the new name is here.

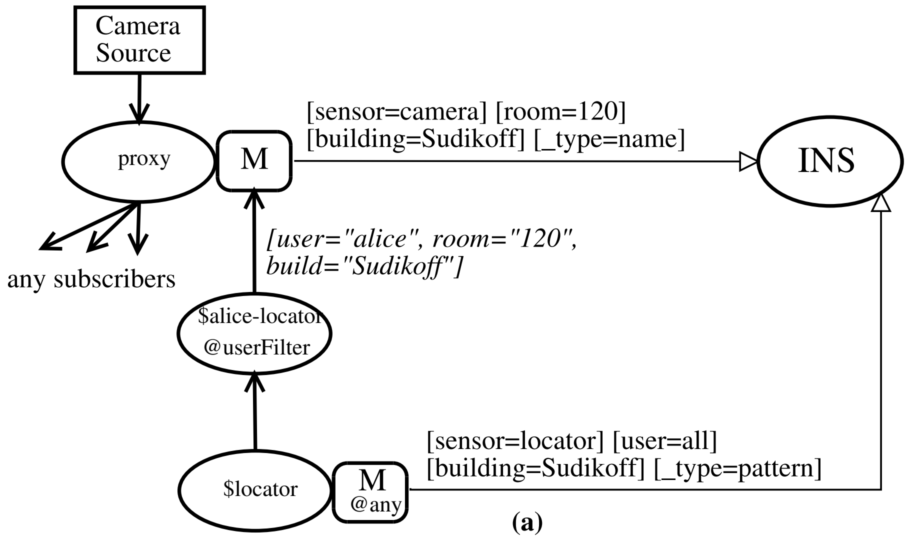
For a context-sensitive name query, when the monitor detects that the name has changed, it a) asks INS to provide a list of matching advertisers using the "_type=name" form of the new name; b) asks INS to unadvertise the "_type=pattern" form of the old name; and c) asks INS to advertise the "_type=pattern" form of the new name. The result is to be notified of any future names that might match this pattern.

A monitor attached to an operator is responsible for adjusting the operator's subscriptions whenever necessary. If it receives an INS message or a Solar meta-event indicating a change in the set of matching names, the monitor evaluates the new set using the selection function to determine the set of desired subscriptions.

4.3.3 Camera example

In Figure 4.6 we show the deployment of the two variants of the camera example from Figure 4.3. Case (a) depicts a camera source with a location-dependent name, and case (b) depicts applications with location-dependent subscriptions to camera sources.

In both cases, the $locator operator's monitor finds all the locators in Sudikoff and picks any one of them (using selection function @any), so that the $locator operator subscribes to that locator source. (The locator sources are not shown in the figure). If for some reason the locator source changes its name and no longer matches the pattern, the monitor of its proxy sends a meta-event to


Camera Source
proxy
M
[sensor=camera] [room=120]
[building=Sudikoff] [_type=name]
INS
any subscribers
[user="alice", room="120",
build="Sudikoff"]
$alice-locator
@userFilter
$locator
M
@any
[sensor=locator] [user=all]
[building=Sudikoff] [_type=pattern]
(a)

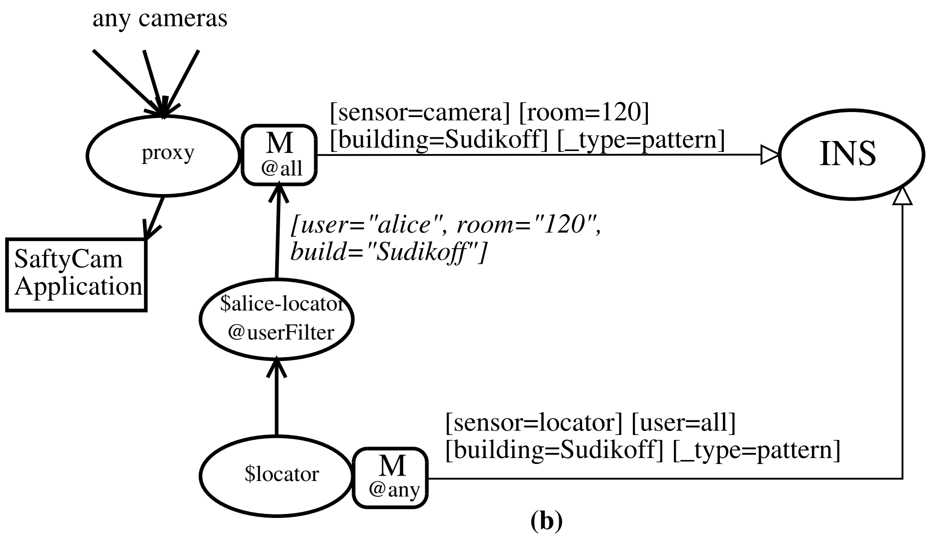

any cameras
proxy
M
@all
[sensor=camera] [room=120]
[building=Sudikoff] [_type=pattern]
INS
SaftyCam Application
[user="alice", room="120",
build="Sudikoff"]
$alice-locator
@userFilter
$locator
M
@any
[sensor=locator] [user=all]
[building=Sudikoff] [_type=pattern]
(b)

Figure 4.6: Deployment of two variants of the camera examples.

its subscribers including $locator. The $locator monitor receives the meta-event and selects another locator among matching names. The events produced by $locator are filtered to produce $alice-locator, then used by the proxy's monitor to adjust the name advertisement (case a) or subscription (case b).

Again, the Solar system and our naming approach could use any scalable, decentralized naming service that support attribute-based queries and persistent queries. In this section we demonstrated how to extend INS to fill that role, but other implementations are possible.

4.4 Evaluation

We set out to measure the name-update performance of Solar's naming service. To obtain realistic events for our experiments, for 12 days we recorded the sensor data from a location system installed in our building.[1] For our experiments we created a "replay source" that read this trace and published each sensor reading as an event, and used it to drive our test clients.

For our experiments we used a set of fixed hosts (FH)[2] with 100 Mbps duplex connections. The average network latency among them was 0.37 ms. We also used two mobile hosts (MH): a Linux laptop[3] and a pen-based Windows tablet.[4] The mobile hosts were connected via 802.11b through a dedicated access point with a clear 11 Mbps channel (distinct from channels used by any other nearby access point). The average network latency from a FH to the tablet was 2.9 ms and to the laptop was 2.7 ms. We used Sun Microsystem's Java runtime (v1.4.1-hotspot) on all platforms.

4.4.1 Latency measurements

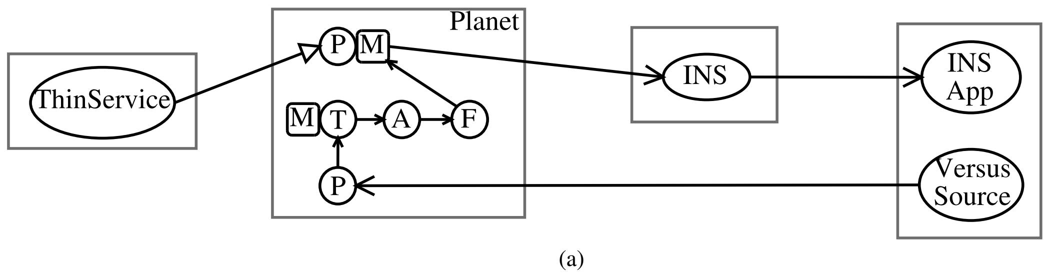
Consider the task of advertising a location-sensitive name, based on the Versus location data. We compare an approach with Solar and an approach without Solar (Figure 4.7). Note that the rectangle represents a FH or MH, P is a proxy, and M is a monitor.

In the Solar approach (Figure 4.7a), we set up an operator graph to provide the desired event stream, using our Versus replay source, a transformation operator (T) that converted badge and sensor numbers to symbolic names, an aggregation operator (A) that remembered the current location of each badge and produced an event whenever a badge changed location, and a filter operator (F) that removed all events except those about a particular person. We created a dummy "ThinService" that submitted this operator graph with a simple context-sensitive name specification. Both the ThinService and the Versus source have a proxy (P).

In the other approach (Figure 4.7b), a "SelfService" application received all the events directly from the Versus source and did all the transformation and filtering internally. Whenever it detected a location change, it announced a new name to INS and sent a name-update multicast message to all matching patterns. We used a standalone INS client application, which registered a simple pattern to receive these name-update messages.

For comparison, we also measured the INS-only configuration in Figure 4.7c. An INS client "sender" periodically changed its announced name by announcing it to an INS resolver. An INS

[1]An IR-based badge-tracking system provided by Versus Technologies, Inc., http://www.versustech.com/.
[2]Dell GX260, 2.0 GHz Pentium 4, 256 MB RAM, running RedHat Linux 2.4.18-openmosix.
[3]Gateway Solo 3400, 450 MHz Intel Pentium 2, 128 MB RAM, running RedHat Linux 2.4.18-14.
[4]Fujitsu C-500, 500 MHz Celeron, 128 MB RAM, running Windows 2000.



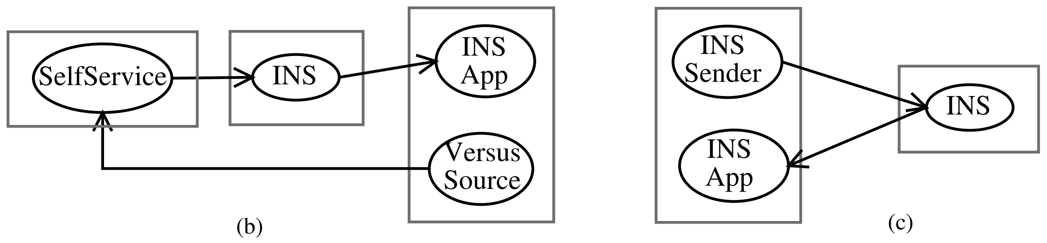


Figure 4.7: The setup for the latency tests.

client application received all the name-update messages and calculated the difference. This configuration allowed us to isolate the overhead of event handling of the other two cases.

The replay source published many events, but only a few ultimately caused any change in the advertised name. We call such events "triggering events." For our experiments, we used a reduced trace that contained only the triggering events, since our goal was to measure the latency between the moment our replay source produced a triggering event and the moment that a client application noticed the name change. Each message contained the timestamp of the original event produced by our replay source. By arranging for this INS application to be on the same host as the replay source, the application simply subtracted the event's timestamp from the current time, obtaining the latency without concern for clock skew.

Figure 4.8 shows the latencies for the three configurations of Figure 4.7, using only fixed hosts. The purpose of this test is to demonstrate the overhead of our approach, and to isolate the effects of the INS core. The x axis represents the progress of time, marked by the count of triggering events. Each data point is the moving average of the preceding 20 triggering events. ThinService and SelfService had similar performance because the bulk of their work, aggregating context information and updating INS, was essentially the same and executed in a similar environment (by the SelfService on an FH or by operators in a Planet on an FH). In general, ThinService had more overhead because it partitioned the work into three operators and connected them with queues, and there was some overhead for queue management. A substantial fraction of the latency, and its variation, was in INS. The bottom curve of Figure 4.8 shows the latency for the INS-only approach of Figure 4.7c. Clearly INS was a significant source of the variation seen in the other curves.

Note that latencies became smaller as the experiment progressed, due to the incremental compilation by the "Hotspot" JVM, but reached steady state after a few hundred events. There were, however, noticeable variations, caused by thread scheduling and temporary network congestion. The latency numbers were fairly small, so these factors had a visible effect on the overall latency.

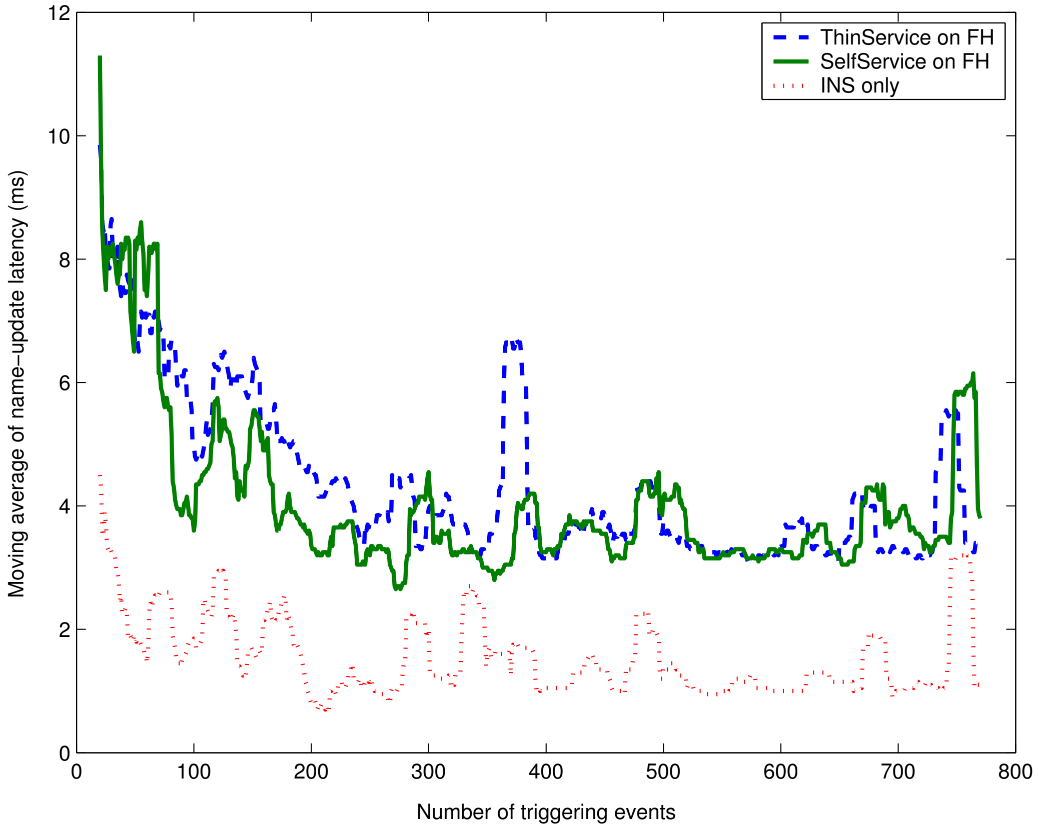


Figure 4.8: Latency results for fixed hosts.

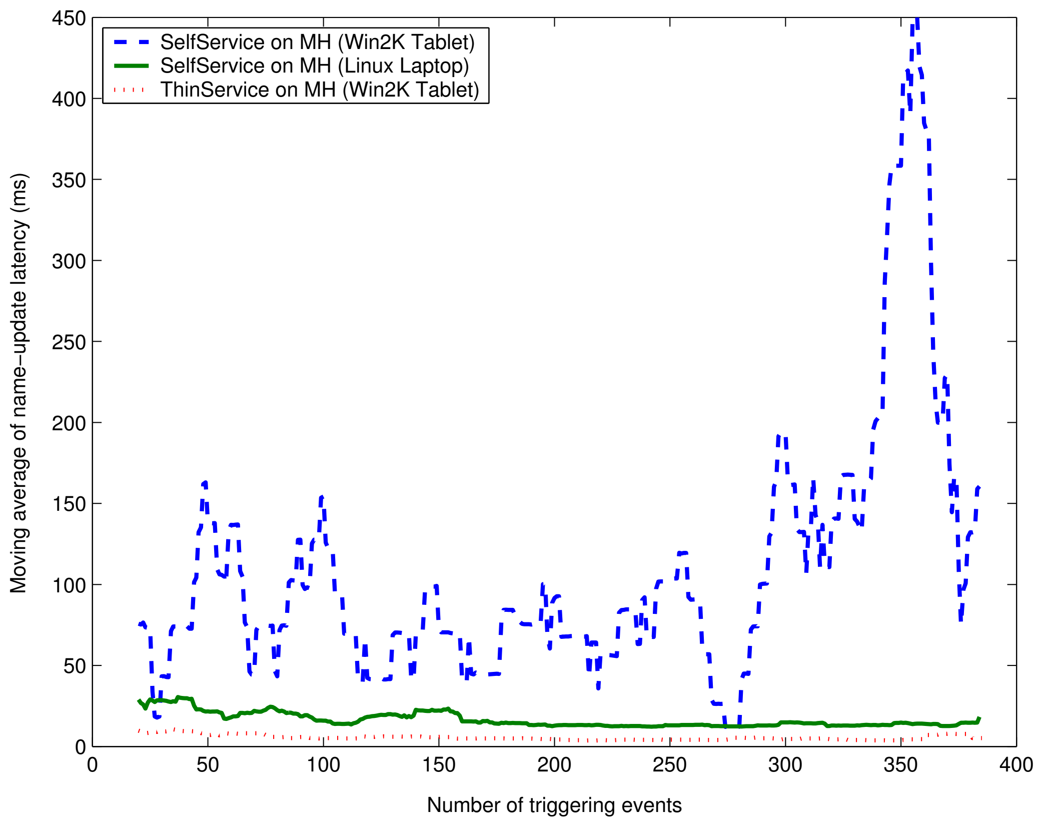


Figure 4.9: Latency results for mobile hosts.

Our second test demonstrates the advantage of our infrastructure approach. Figure 4.9 shows that the ThinService using Solar clearly outperformed SelfService when these services ran on a mobile host. The name updates issued by SelfService (on the Linux laptop) took about twice as long to reach applications as those issued by Solar on behalf of the ThinService. This situation perfectly demonstrates the value of Solar's ability to off-load context aggregation and name updates into the infrastructure, particularly for poorly connected client hosts.

It is interesting to see that running SelfService on the tablet adds further delay and variation. While we have no explanation for the variation, it is likely related to inefficiency of the Java run-time, driver, or OS on the tablet. In any case, Solar's approach minimizes the impact of the client environment by moving the context collection, aggregation, and monitoring into the infrastructure.

Based on these latency experiments, we conclude that our framework can greatly improve responsiveness for context-sensitive names, particularly for clients who reside on devices connected through a slow network. ThinService pushes the chore of monitoring context and updating its name away from the service itself, into the Solar system on the fixed hosts; SelfService was required to receive all events across the slow network, and arrange the name change across the slow network.

4.4.2 Scalability analysis

To evaluate a Planet's capability to support name updates under heavy load, we devised an experiment with a single Planet on one FH, a single INS resolver on another FH, and an array of FHs with clients that request context-sensitive name advertisements. Each client host had three processes: a ThinService, a Versus replay source, and an INS application. The ThinService made the advertisement request, with a graph specification that transformed, aggregated and filtered the Versus location events. The output of the operator graph defined the context-sensitive name for the ThinService. The Versus source advertised a static name, chosen carefully so that it could only be discovered by the ThinService on the same host. The INS application announced a pattern so it could receive name-update messages about the ThinService on the same host. The Versus source publishes (only) the triggering events at a fixed interval, and the sequence number of the event is carried by the name-update message. All the sources waited for a "go" command issued from a control console before they started publishing events. The independence of each host's sources, graphs, and applications is not typical of a real Solar system but allows us to easily scale the number of client applications and their associated load.

We first measured the maximum (triggering) event processing throughput of the Planets. A triggering event was considered processed by the Planet after it triggered a monitor to cancel its previous name announcement, announce a new name based on the values in the event, and send out a name-update message. While varying the number of client applications and publishing rates, we sampled the length of the queues in the Planet to find the maximum throughput that kept the queue length stable. The result is about 870 events per second for one Planet.

The above measurements did not consider, however, the scalability of the directory service, in this case INS. In the next experiment we distributed 40 client applications over 20 FHs and varied each Versus source's publishing rate. The INS application co-located with each source calculated the name-update throughput. We show the results in Figure 4.10. Instead of reaching 870 name updates per second, the throughput peaked near 500. With 40 clients each pushing over 10 triggering events per second, the INS resolver was overloaded and some packets were lost (INS routes name-update messages using UDP packets). The green curve (with * marks) shows the loss rate. The red curve (with + marks), which is the sum of name-update throughput and loss rate, flattens near 910 indicating the maximum event-processing throughput of the Planet. The number is similar to (though a bit higher than) the conservative throughput we measured above (about 870 events per second).

This experiment demonstrates that the Solar system is scalable to the limits of INS, but further experiments are necessary to determine the scalability of our approach on a more realistic workload.

In this experiment, all events were triggering events. In practice, the majority of incoming events are not triggering events, so the system would achieve higher gross event throughput. On the other hand, the three operators we used in our operator graph were quite simple. A computationally intensive operator can dramatically reduce the throughput. While it is possible for the Solar system to distribute operators across Planets to balance the load, or increase the number of Planets and INS resolvers to achieve overall system scalability, ultimately the system will be limited by the Planet CPU(s), by the network, or by the directory service (INS), if a context-sensitive request is defined by a fast event-generating source. Again, the policy-driven data reduction technique presented in Chapter 3 could be used as a flow control mechanism.

4.4.3 Graph loading

Upon receiving a context-sensitive name advertise or query request, the Planet is responsible to load the operator graph defined by the graph specification and attach monitors at appropriate places.

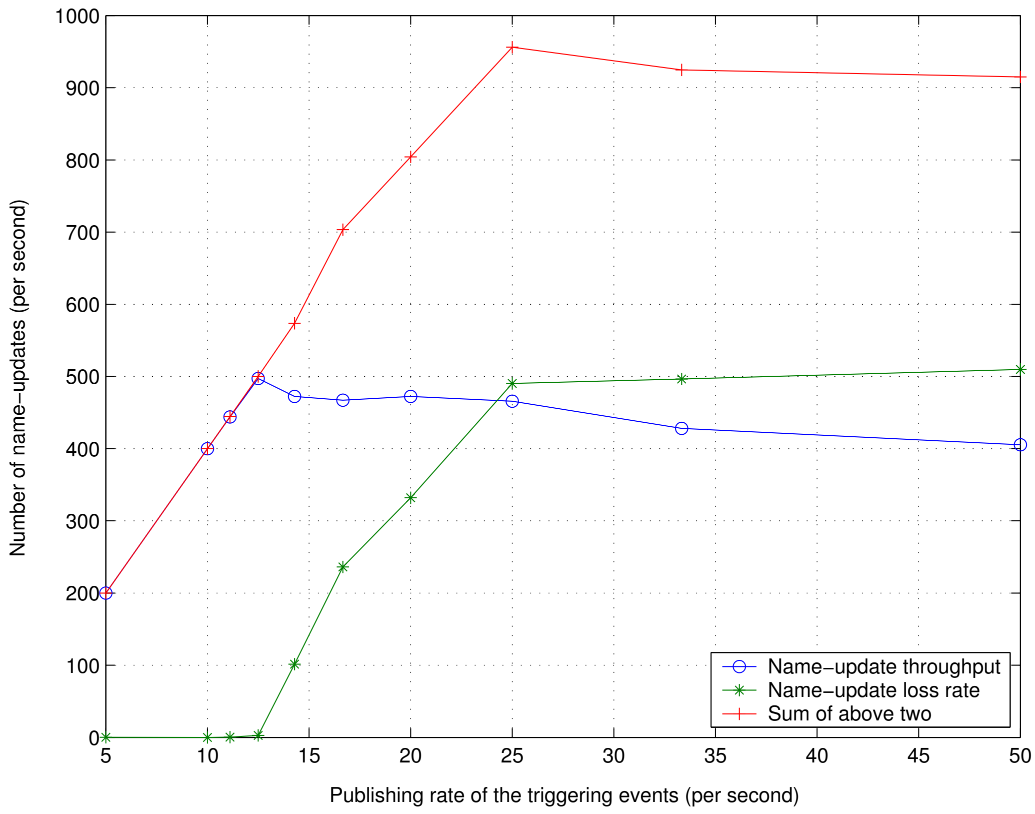
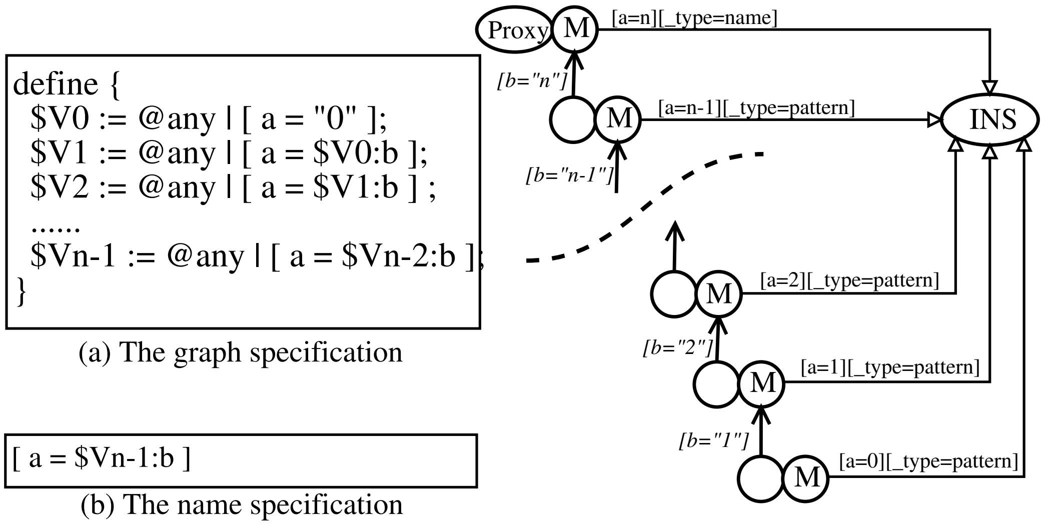


Figure 4.10: Results of scalability tests.



```
define {
  $V0 := @any | [ a = "0" ];
  $V1 := @any | [ a = $V0:b ];
  $V2 := @any | [ a = $V1:b ] ;
  ......
  $Vn-1 := @any | [ a = $Vn-2:b ];
}
```

(a) The graph specification

```
[ a = $Vn-1:b ]
```

(b) The name specification

(c) The operator graph

Figure 4.11: The operator graph/chain that we used to measure performance of graph loading.

In this section, we measure the time to load the operator graph given a graph specification. The performance of graph loading determines a Planet's capability to service context-sensitive requests.

We set up a test case for a context-sensitive name advertisement, which used a chain of operators and context-sensitive name selectors. Each operator was selected using the output of the previous operator in the chain (Figure 4.11). We made several measurements on how long it takes to load the operator chain for different chain lengths.

We ran the experiment on a set of identical Linux workstations (the FH described in Section 4.4.1) that are interconnected with a full-duplex 100 Mbps switched Ethernet. We ran one INS resolver on one host and one Planet on a different host. On a third host we ran a test client that periodically sent an advertisement request, including the chained graph specification. The client's machine sent an advertisement every 5 seconds. The length of the chain varied from 1 to 20 with a step size of 5, and we repeated each length 4 times; we report the average latency for each chain length.

In one measurement we consider all the time needed to recursively load the operators, attach their monitors, establish the subscription links among operators and monitors, start the operator and monitor threads, and initialize the monitors (contacting INS, creating an INS announcer thread to register a name or pattern with INS, and sending a name-update message to INS). In another measurement, the monitor initialization was excluded (so INS costs are not included). We show the results in Figure 4.12.

While both curves grow almost linearly, the time with INS is not a constant overhead compared with the loading time without INS because the longer the operator chain the more monitors need to be initialized with INS. It took about 1.3 seconds to load a monitor chain of length 15 and 0.48 seconds for the length of 5 with the monitors fully initialized. This test was admittedly quite stressful. Typical graphs, we expect, would have a small collection of operators with monitors only at the ends of the graph, and a client only issues requests at startup time. Certainly it appears that a single Planet could load 2–5 graphs per second, for the smaller graphs.

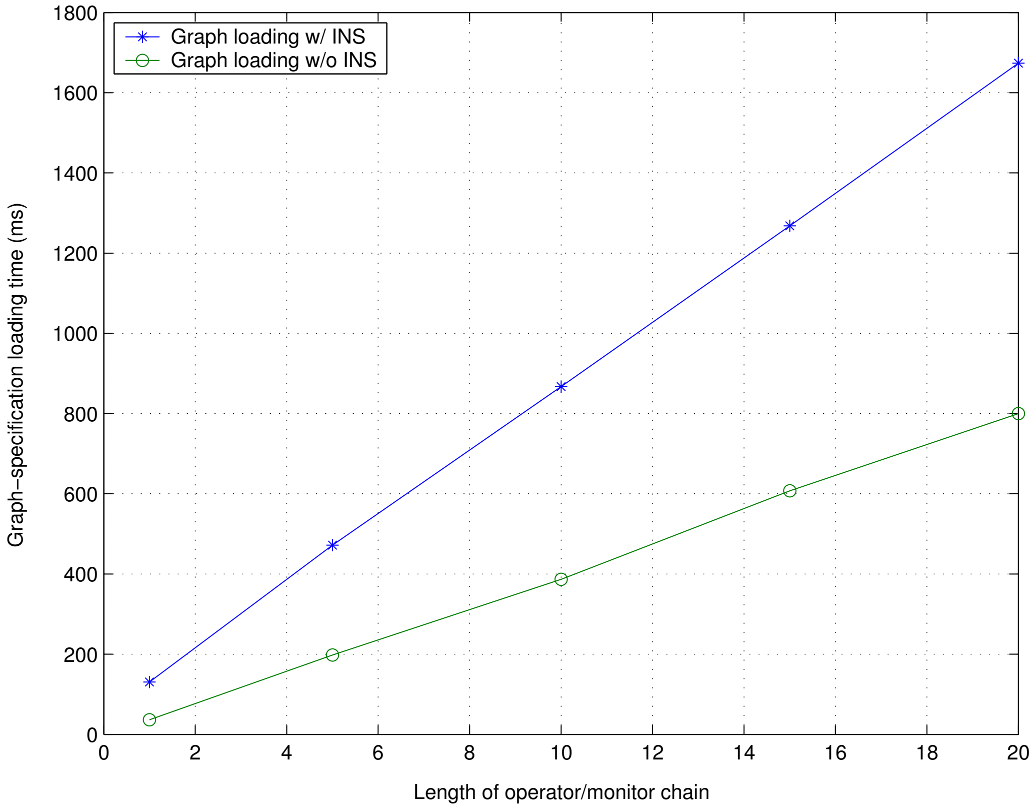


Figure 4.12: Measured time to load an operator chain.

While we believe serving 2 requests per second is sufficient in most situations for a single Planet, there are several ways a busy Planet might handle more frequent context-sensitive requests. The Planet could refuse the request, redirect the request to another Planet, or delay the monitor initialization (with INS) until the monitor receives enough contextual events so the name specification is fully qualified (all values filled).

4.5 Improving scalability

Recently Solar has replaced INS with a more scalable directory service. The performance of the directory's wrapping layer that supports context-sensitive resource discovery, however, should be similar to the results presented in Section 4.4. We are currently evaluating the performance of the new directory service of Solar in a Masters student's thesis [131].

Recall that every Solar service, such as the naming service, runs on all Planets in the network. The new Solar distributes the name space (the advertisements) across all the Planets using the DHT-based routing substrate (Pastry). Each Planet contains a *directory*. For scalability and reliability reasons we make two design choices: the directory on any Planet only holds part of the name space, and each advertisement should be replicated on multiple Planets.

When a Planet receives an advertisement registration, it splits the advertisement into a set of strands. The Planet then hashes the strands into numerical keys and sends the advertisement to those Planets responsible for the keys using DHT transport. The receiving Planets then store the full advertisement in their directory. Similarly, a query also is split into strands, and the Planet hashes the longest strand to a key and sends the query to that key for resolution. Since the Planet responsible for that query key stores all the advertisements containing the corresponding query strand, the query can be resolved appropriately.

This naming distribution and query mechanism is similar to INS/Twine [8]. Note that, however, we do not enforce the alternative tag-value semantics on the record structure. Solar also supports persistent queries to allow monitoring of the name space while INS/Twine does not.

4.6 Related work

INS [3] unifies resource discovery (naming) and communication (message routing). Applications desiring context-sensitive names, however, must monitor the context themselves and re-advertise their name as needed. Furthermore, INS does not support persistent name queries. Solar uses an extended version of INS as the directory service at the core of its own context-sensitive name service. By offloading context processing and monitoring to Planets in the Solar infrastructure, our framework improves responsiveness and scalability. INS/Twine achieves more scalability by partitioning the name space across resolvers by mapping names into numeric keys [8]. Solar could use INS/Twine as its core directory service, but would need a different mechanism to implement persistent queries. Instead, we chose to adapt Pastry and include support for persistent queries.

A Location Information Server (LIS) [85] integrates location information and a resource directory, based on the X.500 directory service and the Light Weight Access Protocol (LDAP). The LIS shields the application from the methods for obtaining the location information and provides a set of APIs for query and event notification. While LIS provides some Solar features, including limited context translation and support for location-sensitive names, the capability of LIS is limited by the predefined configurations and rules. Solar provides a programmable interface to allow arbitrary definition of context-sensitive names. Solar also uses peer Planets to cooperatively service clients' requests and to disseminate contextual events, for better responsiveness and scalability.

Active Names [129] is a flexible approach to locate a service and perform customized operations on the returned results by allowing applications to specify a chain of mobile programs through which the result should pass. While it is geared towards wide-area service composition, we could use a similar concept to support context aggregation by merging several chains, giving the effect of an operator graph. Solar's specification language, however, is more expressive than a sequence of names. The unstructured name representation also has limited expressiveness and does not allow context-sensitive source selection and name definition. On the other hand, Solar only allows context-sensitivity at the edge of an operator graph (source selection) while any Active Names program (operator) on the chain can alter (presumably based on some context) the next destination for each event at runtime. This extreme flexibility, we believe, is not necessary for the context-aggregation task and may actually make it difficult for developers to keep track of the event flow structure. Finally, Solar implements the recursive context monitoring and graph adjustments directly in the Planet, while the Active Names applications need to implement this feature individually.

A non-procedural language, iQL, also can specify the logic for composing pervasive data into context [44]. Their model uses attribute-based naming and supports both requested and triggered evaluation. For one composer, iQL allows the inputs to be continually rebound to appropriate data sources as the environment changes. The iQL language is powerful and expressive, with many language-level facilities for context aggregation. The language iQL complements Solar in two ways: iQL could be the programming language for individual operators, or iQL could be the high-level specification language the compiler could decompose into a graph specification used by Solar.

The Context Toolkit provides several abstractions to construct a context service [50]. It is a distributed architecture supporting context fusion and delivery. 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 to compose refined context that can be shared by other applications. The Context Toolkit provides a static attribute-based directory to allow its components to register a name so applications can find them. It does not support context-sensitive names or subscriptions, or persistent name queries.

There are numerous directory services and resource- and service-discovery systems. Jini [92] allows a client to locate a service and download its proxy interface, matching on the class of the proxy. The Service Location Protocol (SLP) focuses on the protocol for automatic discovery [59], Czerwinski et al. focus on expressiveness and security [46], and Castro et al. deal with inter-domain service discovery [30]. DeapSpace [98] and Heidemann et al. [64] propose approaches for discovery in ad-hoc sensor networks. None of these systems, however, has explicit support for context-sensitive names or subscriptions.

Chapter 5

Dependency and Load Balance

Applications rely on the operators injected into Solar for customized context information. If a Planet fails, all the operators hosted by that Planet are lost. Considering the relationship among the Solar components (applications, operators, and sensors), we can say that a directed operator graph is also a *dependency graph*. Given a channel with a source X and a sink Y, we say Y *depends* on X and we say Y is a *dependent* of X since Y needs the data produced by X. These dependencies present several challenges. First, the dependents of some component X cannot function if X is lost due to host failures. Second, the dependents of X must track the location of X to keep data flowing as X moves from one Planet to another. Finally, any application-specific operators should be garbage collected to reclaim computational resources when the application finishes.

We describe Solar's approach on dependency management in Section 5.1. A related issue, while not the focus of our research, is *load balancing*. Namely, it is desirable to distribute operators evenly across all Planets instead of putting all operators on a relatively few Planets. We briefly describe a possible solution based on k-way min-cut graph partition algorithm in Section 5.2.

5.1 Dependency management

Each component in Solar is assigned a globally unique numeric key that is invariant once the component is registered, even if it is restarted or moved to another Planet later. Thus, we can specify the dependency relationship between two components as if one key depends on another, for example, $K_X \to K_Y$ if X depends on Y and K_X and K_Y are their keys.

One approach to manage the dependencies is for the dependent of a component X to monitor X's liveness using a soft-state based protocol [109] and trigger a restart process whenever the dependent detects the failure of X. This approach is conceptually easy but difficult to achieve efficiently and scalably. First, the dependent may be co-located with X on the same host and thus they may fail together. Second, if X has multiple dependents, they have to run a distributed election algorithm to select one as the monitor, which is non-trivial given a dynamic group of dependents.

In this section, we present the system design details of Solar's dependency service, which is used to manage components of the context fusion network, including both external clients (data sources and applications) and operators hosted on Planets. The key contributions of this service are distributed protocols for proactive component monitoring and recovery, and for both instance and name based dependency tracking.

X, K_X	A component and its key
R_X, M_X	A component's root and monitor
P_X, P_{R_X}, P_{M_X}	Planet hosting X, R_X, and M_X
P, P_M	Planet and its monitoring peer
C, CP	A client component and its proxy

Table 5.1: Notation used in discussion of dependency management.

We list the notation used in this chapter in Table 5.1. In the following discussion, we indicate implicitly that a message is delivered through the Pastry DHT routing mechanism if the destination is a key. Otherwise, the message is delivered through a direct TCP/IP connection if the destination is an IP address.

5.1.1 Component registration

Before using the dependency service, a Solar component (whether a source, an operator, or an application) must first register with the service. When registering, a component X provides its key and configuration information as follows: 1) the action to take if X fails, such as to restart it or to email an administrator; 2) the command (or object class and initialization parameters) used to start X; 3) any restriction regarding the set of hosts where X be restarted; and 4) whether Solar can reclaim X when it has no dependents for a certain period of time. Solar records such configuration information and makes it available when X fails.

Some components may not be restarted on just any host. For instance, a data source may have to run on a particular type of host, to access a piece of sensing hardware. On the other hand, most operators are self-sufficient, processing events as they arrive, so Solar may restart them on any available Planet when they are lost.

Some components maintain state during operation, and require that state to be restored after a crash. We assume the component may checkpoint its state at a different host (using some persistence service, for instance) so the execution state is available during recovery. This issue is beyond our research scope. Solar also may migrate a running operator to another Planet. Solar implements a weak mobility scheme for operator migration, namely, it asks the operator to capture its own state and to restore it at the destination host [20].

Solar requires each component to explicitly identify the set of components it depends on; since dependencies may vary with the circumstances, components register (or remove) dependencies whenever necessary. A component may specify two types of dependencies: *key-based* or *name-based*. In other words, a component may specify the keys of the components on which it depends, or a name query that will be resolved to discover components, who supply their keys in name advertisements. Since a name query may be resolved to multiple names, the requesting component may use a customized function to select an appropriate component (Chapter 4).

For either type of dependency, the dependent supplies a policy determining how to handle the failure, restart, or migration of the other component, or (for name-based dependencies) a change in the results of the name query and selector function. For example, if X depends on Y and Y fails, the policy of X may be to wait until Y is restarted. If the $X \rightarrow Y$ dependency is name-based, another reasonable policy is to use the output of the function to select a different component. When a component has zero dependents, the component may be subject to garbage collection to release occupied resources.

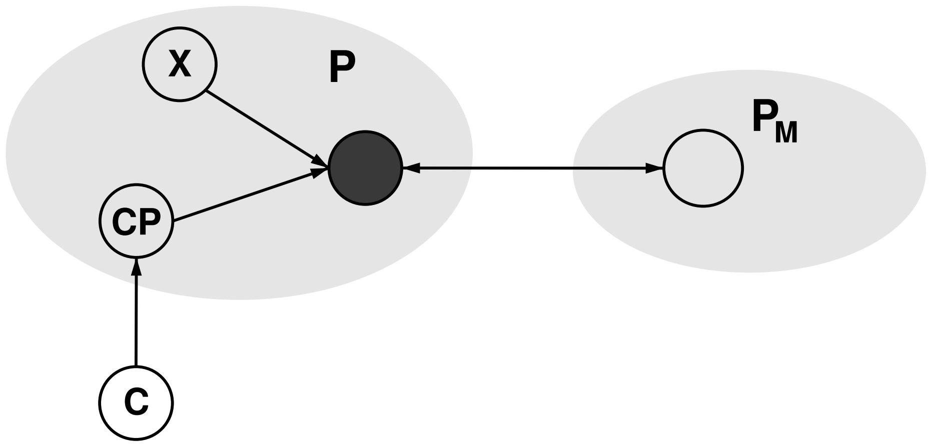


Figure 5.1: Dual monitoring of a Planet on behalf of its operators and attached clients.

In summary, a component registers with Solar's dependency service to provide its key, information about its restart configuration, and a list of its dependencies and associated policies. Solar monitors the component and restarts it using its configuration in case of failure; Solar also tracks the component's state on behalf of its dependents and takes appropriate actions according to the dependent's policies as the state changes.

5.1.2 Monitoring and recovery

If an operator X failed due to an internal exception, its hosting Planet can simply restart it. To recover X from a Planet failure, we install a dedicated monitor M_X as X's watchdog. Note that the monitors here in the dependency service are different than those in the naming service (Chapter 4). X periodically sends a *token* message to M_X through direct IP, and M_X assumes X is lost if it has not heard any token for a given period time. Then M_X selects a Planet to restart X, and the new X will register with the dependency service and send tokens to M_X. On the other hand, M_X also periodically sends a token to X for monitoring purposes. So X also could detect the failure of M_X and restart it at another Planet. This bi-directional monitoring ensures that both X and M_X can be restarted on new Planets unless they fail simultaneously.

We could have installed one monitor per component, but that would incur a large amount of monitoring traffic given many operators and relatively few Planets. It is a waste of both CPU power and network bandwidth. Thus we group the operators on the Planet and monitor them as a whole, so we only need one monitor for the Planet and the monitor restarts all the operators in the group when that Planet fails. As shown in Figure 5.1, the black circle is a representative of Planet P and sends aggregated tokens to monitor P_M for all the operators on P. CP is a proxy operator for client C; the proxy is monitored just as with other operators.

When starting up, a Planet P first tries to find another Planet as its monitor, by sending a request to a random key through Pastry; the receiving Planet responds with its IP address. When the request successfully returns, P starts to send an aggregated token, including the key and configuration of its operators, to its monitoring Planet P_M using a direct IP connection. On the other hand, P_M also will send an acknowledge token to P at a fixed rate. Both P and P_M maintain a timer that is updated whenever a token is received from its peer and is expired when it is not updated for a certain period of time.

If the token sending interval at P is t, P_M assumes that if it has not received any token for a period of kt, P has crashed. Similarly, P assumes P_M is lost if it has not received any token for that period. By tuning the parameter k, the protocol becomes more resilient to intermittent token loss or more quickly able to recover from failures.

When the timer at P_M expires, it starts the recovery process of all operators hosted originally by P. It chooses a random Planet by sending a request to a random key; the request contains the key and configurations of the operators to start. The receiving Planet restarts all the operators locally (optionally, it could forward some or all of the request to another Planet, if it is overloaded). Then P_M removes P from the list it monitors. If P detects that P_M has failed, it sends a monitoring request to a random key until a Planet other than itself is found.

To move an operator X from Planet P to P', P first removes X from its local repository and then requests P' to install a new copy of X. If P fails during migration process, P_M eventually times out and triggers the process of recovering X. Once P' has successfully initiated X, it requests P_M to remove X's key and configuration. From then on X will be watched by P''s monitor.

We now discuss how to monitor and recover an external client C. The idea is similar; client C and its serving Planet P run a dual monitoring protocol that monitors each other's liveness. If C has failed, P tries to restart it if C's configuration contains instructions. Otherwise, it simply removes CP and de-registers C.

On the other hand, the proxy operator CP registers with the dependency service and also is being monitored by P_M. When P has failed, P_M does not try to recover CP but simply removes CP from the list of operators to be restored. If C is still alive, it will ask P to restart CP. If C is still alive while P has failed, C may go through a discovery protocol to find another Planet for service.

Note that P and P_M form a two-node ring to monitor each other. If both P and PM fail simultaneously, our protocol is not able to recover the lost operators on P. Assuming a Planet fails with probability p and there are no co-related failures, the probability of simultaneous failure is p^2. We can further reduce this probability by inserting more monitors to expand the circle. Each monitor P_M periodically sends a token to its adjacent neighbor on the ring, which effectively monitors P_M and restarts it when it failed in a similar way to previous discussion. The token contains current addresses of all participating monitors, so a failure of non-adjacent members does not break the ring. With m nodes in the circle, the protocol failure probability exponentially reduces to p^m, with cost of linearly increased total token rate mr (assuming r is the token rate on one edge).

5.1.3 Tracking dependencies

To facilitate coordination between component X and those on which it depends, we define a *root* object for X and denote it R_X. The root always runs on the Planet responsible for X's key, so any party can communicate with X by sending messages to K_X without knowing its network address. R_X tracks the current location of X, and forwards the message to X. X's monitor retains X's restart configuration, and the root R_X keeps X's dependency policies by receiving periodic updates from X as shown in Figure 5.2.

Upon receiving one of X's periodic update messages, R_X records X's current location and list of dependencies. If any dependency is name-based, R_X queries the directory service and evaluates the selector function on the results. The output determines the component(s) X should use; X is

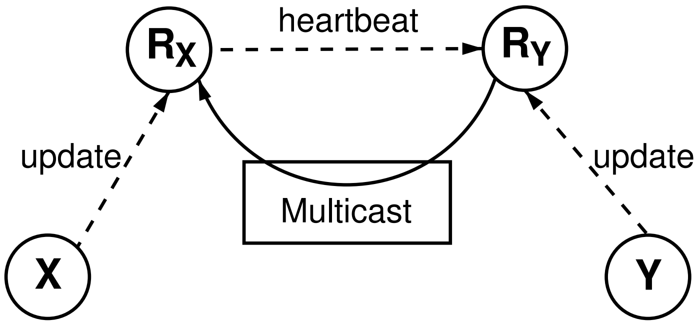


Figure 5.2: A component's root tracks its dependencies. Here we assume X depends on Y.

notified about any changes in the selection and X updates R_X if it indeed made those changes. Thus R_X contains a list of keys of the components X currently depends on, either explicitly specified or resolved from the name query.

Root R_X receives notification from either P_X or M_X whenever X fails, restarts, or migrates. Then R_X publishes these notifications as events through the multicast service with publishing key $g(K_X)$, where g is a deterministic function to return the publishing key. The purpose of the mapping function is to allow subscriptions based on X's key K_X, instead of the key of R_X. R_X itself subscribes to the mapped keys of all the components on which X depends, such as $g(K_Y)$. These events trigger R_X to take actions specified in X's dependency policies, for instance, to re-evaluate a selector function to get other usable components if Y has failed, or ask P_X to reboot X if Y has rebooted. Note that for scalability reasons, R_X does not keep the keys of its dependents. Instead, it simply publishes events using a single mapped key to which X's dependents subscribe.

Given the list of keys on which X currently depends, where we expect the list to be relatively small, R_X sends a periodic *heartbeat* message to all the keys through P2P at a low frequency. The root of each components receives the heartbeat and resets its timer; the timer fires when it has not heard any heartbeat from any dependents for a long time. Then that root, say R_Y for component Y, assumes there are no dependents for Y anymore. If Y's configuration permits, R_Y requests P_Y to delete Y and de-register it from the dependency service.

Note the state kept by each root is soft and can be recreated from its component's periodic updates. This soft state helps the situation when P_{R_X} crashes, or R_X has moved to a different Planet due to overlay evolution. If the R_X times out on X's updates, it simply removes itself.

5.1.4 Protocol optimizations

In many cases we desire X be restarted on a Planet near the original P_X to maintain network proximity to other components. To achieve this goal, we add one more field to the monitoring tokens; this field contains a set of neighbor nodes N_X of P_X. When P_X fails, M_X may request a random Planet from N_X to start X instead of sending a request to a random key. It also is possible to make M_X run on a Planet nearby to P_X using the same approach, but with increased possibility that P_X and P_{M_X} fail together if "nearby" nodes may have a correlated failure.

Yet another optimization can further reduce the traffic of monitoring and recovery by aggregating protocol messages that are being sent to the same IP or Pastry key. There are two places where the dependency service can group the messages. A Planet as the message originator may group token messages and heartbeat messages sent to the same IP address or Pastry key. On the other hand, a Planet as an intermediate node on the transmission path may check the destination key of passing messages and group them by key. Simulations on realistic network topologies show that the paths for messages sent to the same key from nearby nodes in the underlying network converge quickly after a small number of hops [28], so this message aggregation may significantly suppress protocol overhead.

5.1.5 Evaluation

In this section we present some results on monitoring and migration protocols. We performed the experiments on seven Linux-based workstations,[1] all connected using a 100Mbps switch. The average round-trip delay between any two hosts is about 0.25ms. On each workstation, we run a single Planet that hosts several operators. We set the token rate to be once per 3 seconds, and the monitoring timeout to be 9 seconds. The root update rate is also once per 3 seconds.

First we measure how long it takes Solar to recover from a Planet crash and restore the operators on another Planet. We deliberately crashed a Planet and report the interval between when its monitor detected the failure until the monitor's recover request returned. The recover request was sent to a random key whose responsible Planet restarted the operators from the crashed Planet and registered them with the local dependency service. We show the result in Figure 5.3, with the recovery time measured against the number of operators on the crashed Planet. We notice the time grows linearly as the number of operators to recover increases.

We then measure the time it takes for an operator to migrate from one Planet to another. In this test, every 5 seconds each Planet moved a random operator it currently hosted to another Planet. The migration first requested the current Planet to remove the information about the moving operator, and then sent the migration request to a random key. The Planet received the request to restart the operator using the configuration in the request. We measure the time between the migration request and the time the request successfully returned to original Planet. We recorded the delay numbers on all seven Planets during a 10-minute run, and Figure 5.4 shows the distribution. The median migration time is about 12 milliseconds. The more hops (P2P) a migration request had to go through, the more latency it incurred. We also saw a long tail indicating a couple of 120 milliseconds delay, which might be caused by the combined effect of a large number of hops and thread scheduling effects at the destination Planet.

5.1.6 Related work

The concept of data fusion is essential for component-based context-aware systems, such as Context Toolkit [51] and ContextSphere [44]. Until now, we have not seen a general service, like Solar provides, to manage the dependencies between the distributed data-fusion components. We believe our service abstracts away many complexities for coordination in a heterogeneous and volatile pervasive-computing environment.

Our dependency service mainly concerns itself with the temporally-coupled components, and does not directly apply to other coordination models [20]. For instance, Stanford's Intelligent Room system provides temporally-decoupled communication over a tuple space [74]. Here we can only say a component depends on the tuple-space service while each component may be individually

[1] Dell GX260, 2.0 GHz CPU, 512 MB RAM, and running Red Hat Linux 9

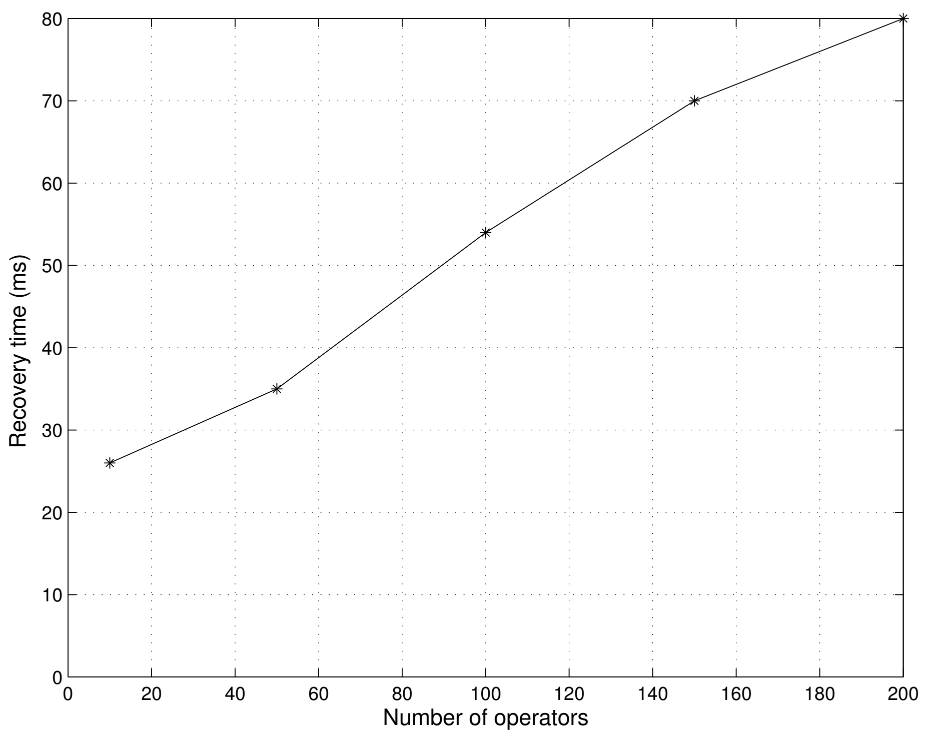


Figure 5.3: Operator recovery time in milliseconds after a Planet crash.

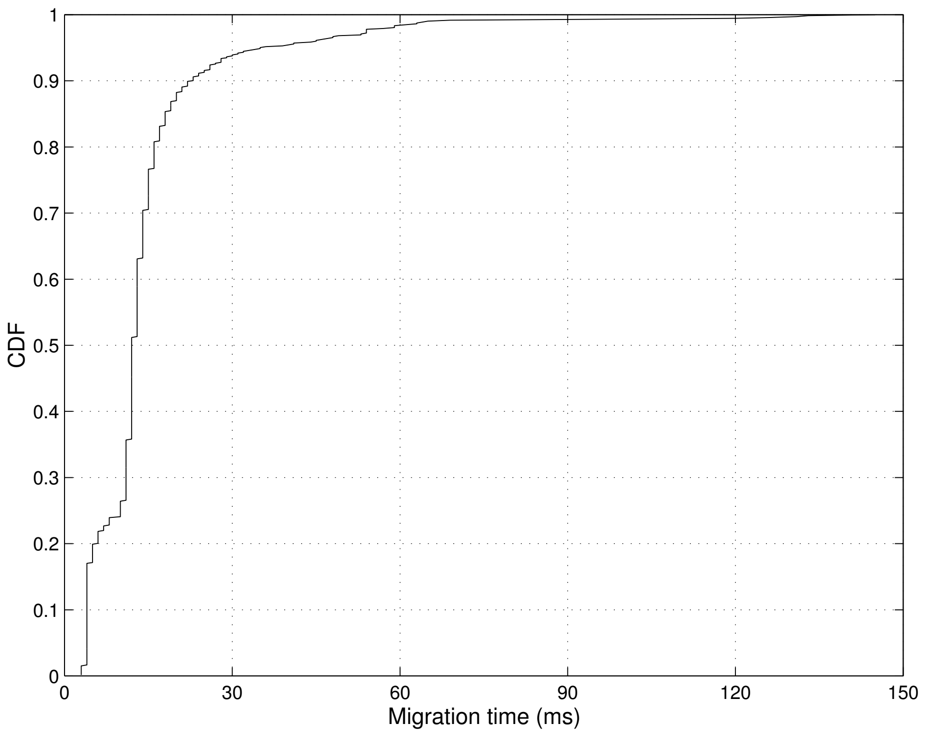


Figure 5.4: Distribution of operator migration time for 10-minute run.

monitored and recovered. The functional and restart dependencies between components, however, are not clearly defined.

Recovery oriented computing (ROC) aims to reduce recovery time and thus offer higher availability [101]. One ROC technique is recursive restartability (RR), which groups components by their restart dependencies instead of functional dependencies [21]. When error or malfunction is detected, including component failure, the RR system proactively restarts the minimum component group containing the offending one. Our approach has a smaller scope and is focused on the distributed dependency management protocol. Solar supports restart dependency, in addition to functional dependency, by requiring explicit component registration.

Solar uses a rendezvous point in the infrastructure to manage a component's dependencies. Given a relatively stable overlay network and the soft-state based root, the root becomes a natural entry point for inter-component coordination. This indirection-based technique has been used to manage large-scale event multicast [26] and host mobility [144].

5.2 Load balancing

In this section, we briefly discuss the load balancing service in Solar. Most of this work is done by Ming Li.

When an application requests the deployment of an operator graph, Solar launches the operators on available Planets. Solar attempts to deploy operators, and re-arrange them as necessary, to balance the computational load on Planets and to reduce inter-Planet communication traffic. This iterative re-deployment process is self-monitoring and self-tuning. The Planet profiles the load of each operator, measuring its CPU usage and the event-arrival rate for each incoming channel.

We can abstract the load-balancing problem into a k-way min-cut graph-partitioning problem, where k is the number of Planets. We abstract the operator graph into a flow graph where nodes represent operators and edges represent channels; node weights represent operator CPU usage and edge weights represent event flow on that channel. We used MeTis,[2] a fast and scalable graph-partitioning software package, to compute an approximate partitioning that attempts to minimize the communication cut and balance the load.

To balance CPU load requires centralized knowledge of the CPU load of every operator and traffic flow on every channel, information that is expensive to collect. Fortunately, this global load-balancing effort is only mandatory when the CPU load of some Planet(s) is above a high threshold. More often we apply a "local" effort focused only on reducing traffic among Planets. In this case, we apply the same min-cut algorithm to the graph corresponding to a single application or a small group of applications. This local tuning reduces inter-Planet traffic globally as well as locally, without substantially harming global CPU balance. We use this local tuning frequently, and the global tuning occasionally.

[2]http://www-users.cs.umn.edu/~karypis/metis

Chapter 6

Sensor Analysis

To experiment with location-aware Solar applications, we deployed an indoor location tracking system covering our department building (Section 6.1). Our campus-wide deployment of WiFi access points also allows us to estimate the locations of wireless devices (Section 6.2). Although the location infrastructure has been in place, we believe that it is critical to analyze the characteristics of these services quantitatively because of the tight interactions between location-provision systems and location-aware applications.

Although some evaluations of experimental location systems are available, we have seen few reports with a detailed analysis of data collected from a deployed location system. In this chapter, we present a first attempt to both quantify the data quality and characterize user-movement patterns based on location traces from off-the-shelf location systems that are in daily usage. Although our study is inevitably based on a particular deployment and biased toward a certain user population, the results still serve as feedback on how a location system behaves and as a guideline for the design of both a middleware system, such as Solar, and location-aware applications in general.

6.1 Versus sensors

We installed a commercial locating system, VIS (Versus Information System),[1] in our department building. Our installation of VIS contains 79 IR and 7 RF (operating at 433.92 MHz) ceiling-mounted *sensors*, which are wired into 4 *collectors* (one collector handles up to 24 sensors). The collectors are then daisy-chained into a *concentrator* that interfaces with the Ethernet.

An IR sensor has a range of 15 feet and an RF sensor has a range of 90 feet. A personnel badge periodically emits IR and RF signals (as a 42 bit packet with 16 bit ID), which are picked up by IR and RF sensors respectively. The packets are forwarded to collectors for re-formatting and finally relayed to the concentrator. A badge contains an IR transmitter, an RF transmitter, a motion sensor, and optionally a push button. The badge emits an RF signal about every 2 minutes, and emits an IR signal more frequently (about every 3.5 seconds) when it is on the move than when it is stationary (about every 4 minutes). The RF channel serves as a backup indicating the badge is still in range (180 feet diameter with RF sensor as the center) even though the IR signals might be lost (for instance, due to IR's line-of-sight problem). The badge uses its internal motion sensor to adjust how often to send IR signals, as a way to prolong the battery's life time.

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

badge_num	the unique ID of the sighted badge
col_num	the unique ID of the collector for the reporting sensor
sen_num	the ID of the sensor under that collector
tcount	the sequence number (4 bits) of the badge signals
motion	whether the badge is in movement or stationary
button	whether the button (if the badge has one) is pressed
battery	whether the badge's battery is low
timestamp	the time when the report arrived at the logger

Table 6.1: List of the data fields for each record in VIS location trace.

Our 3-story department building has about 60 offices, labs, and classrooms. We deployed 61 IR sensors to cover most rooms (except one office due to the resident's request, four machine and storage rooms, and six restrooms), and 18 IR sensors to cover hallways and stairways. Several large labs were partitioned into 2 or 3 zones with one IR sensor installed for each zone. The 7 RF sensors were distributed evenly to cover the whole building.

We captured the output of the Versus concentrator during an academic term (January 5 to March 12, 2003). During the period for this study, the author and one other person wore a badge "all the time". Although we distributed the badges to 12 other students, enrolled in a "pervasive computing" seminar during the term, they did not wear their badges consistently. These students were divided into six groups to develop location-aware applications. About half of the class had offices in our building and the others did not. We also attached badges to printers, chairs, laptops, and so on. Thus the peak workload for the VIS was about 25 badges in the building, and the peak load for one sensor was about 10 badges (for example, on final project demo day).

Our 67-day trace data contains about 1.5 million records, collected from January 5 to March 12, 2003. Each record represents one sensor sighting of one badge, as shown in Table 6.1. There are four holes in our collected data, one on January 13 due to a software upgrade (5 minutes) and the other on February 25 due to a hardware upgrade (2 hours). On March 6 and 7, we migrated the source collection process onto a dedicated server and lost a few hours of data on each of these two days. Our study results do not include any data from these four days.

Although the RF and IR sensors were wired together in the Versus system, we can think of them as two layers, one consisting of IR sensors only and the other consisting of RF sensors. These two layers have different characteristics such as update rate and granularity. It is interesting to study them separately and to compare them shoulder-to-shoulder. Thus for our study, we separated the trace into two parts: one is generated by IR sensors, and the other by RF sensors. In the rest of discussion, we refer the IR layer as **Versus/IR** and the RF layer as **Versus/RF**.

6.2 WLAN sensors

Our campus is compact, with over 161 buildings on 200 acres, including administrative, academic, residential, and athletic buildings. Our school installed more than 550 access points from Cisco Systems, mostly an Aironet model 350, to provide 11 Mbps coverage to nearly the entire campus. Each access point (AP) has a range of about 130 to 350 feet indoors, so there are several APs in all but the smallest buildings. Although there was no specific effort to cover outdoor spaces, the campus is compact and the interior APs tend to cover most outdoor spaces. Thus each 802.11 device can be roughly located on campus by relating its currently associated access point.

We describe two approaches to infer with which access point a 802.11 card is currently associated: one is push-based (syslog) and the other is pull-based (SNMP). For both approaches, we recorded traces for the 65 days from January 5, 2004 through March 9, 2004, inclusive. Note that this data was collected one year later than the Versus data.

We configured the access points to transmit a syslog message every time a client card authenticated, associated, reassociated, disassociated, or deauthenticated with the access point (see definitions below). The syslog messages arrived via UDP at a server in our lab, which recorded all 23,548,687 of them for the 65-day period of our analysis.

Most APs contributed to the syslog trace as soon as they were configured and installed. We saw 506 of the 550 APs and 5936 unique MAC addresses in our trace. Although some APs appear to have never been used, many were misconfigured and did not send syslog messages. Since syslog uses UDP it is possible that some messages were lost or misordered. As a result of these spatial and temporal holes in the trace, some of our statistics will undercount actual activity.

Our syslog-recording server added a timestamp to each message as it arrives. Each message contained the AP name, the MAC address of the card, and the type of message:

- *Authenticated.* Before a card may use the network, it must authenticate. We ignore this message.

- *Associated.* After authentication, a card chooses one of the in-range access points and associates with that AP; all traffic to and from the card goes through that AP.

- *Reassociated.* The card monitors periodic beacons from the APs and (based on signal strength or other factors) may choose to reassociate with another AP. This feature supports roaming. Unfortunately, cards from some vendors apparently never use the Reassociate protocol, and always use Associate.

- *Roamed.* When a card reassociates with a new AP, the new AP broadcasts that fact on the Ethernet; upon receipt, the old AP emits a syslog “Roamed” message. We ignore this message; because it depends on an inter-AP protocol below the IP layer, it only occurs when a card roams to another AP within the same subnet.

- *Disassociated.* When the card no longer needs the network, it disassociates with its current AP. We found, however, that the syslog contained almost no such messages.

- *Deauthenticated.* While it is possible for the card to request deauthentication, this almost never happened in our log. Normally, the associated AP deauthenticates the card after 30 minutes of inactivity. In our log it is common to see several deauthentication messages for a widely roaming card, one message from each subnet visited in the session; we ignore all but the message from the most recent AP.

We also used the Simple Network Management Protocol (SNMP) to periodically poll the APs. We chose to poll every 5 minutes to obtain information reasonably frequently, within the limits of the computation and bandwidth available on our polling workstation. Each poll returned the MAC addresses of recently associated client stations, and the current value of two counters, one for inbound bytes and one for outbound bytes. Our 65-day trace period includes 40,010,036 of these SNMP records, of which we extracted 20,430,974 messages for localization purpose. We saw 543 Of the 550 APs and 6006 unique MAC addresses in our trace.

Our network does not use MAC-layer authentication in the APs, or IP-layer authentication in the DHCP server. Any card may associate with any access point, and obtain a dynamic IP address. We thus do not know the identity of users, and the IP address given to a user varies from time to time and building to building. We make the approximating assumption to equate cards with users, although some users may have multiple cards, or some cards may be shared by multiple users.

In the rest of discussion, we refer to the campus-wide localization mechanism using the syslog message stream as **Campus/syslog** and the other as **Campus/SNMP**.

6.3 Measurements and results

In this section, we present the analysis of location-update traffic. We focus the discussion on the data quality and volume (Section 6.3.1 and Section 6.3.2), load disparity across users and zones (Section 6.3.3), sighting intervals of users (Section 6.3.4), and user prevalence (Section 6.3.5). As a reminder, the **users** are approximated by the badges (in Versus) and 802.11 cards (in Campus). The **zones** are represented by the name of sensors (in Versus) and name of access points (in Campus).

6.3.1 Data quality

Unfortunately none of the four systems generates perfect location data for direct usage.

As a dedicated and commercial locating system, Versus performed most reliably, and we could easily parse the data stream to obtain which badge was sighted at which sensor. Curiously, however, the system did generate some errors reporting a non-existent badge or a badge at a non-existent sensor. There are 11 such errors out of 978,469 in the Versus/IR trace and 57 out of 550,000 in the Versus/RF trace. The software part of the system has to check a human-maintained table to detect these errors.

Campus/syslog sometimes reported incomplete messages (probably caused by the syslog daemon), 183 out of the whole trace. While it is more light-weight to transmit messages using UDP, it may have caused the packets to arrive out-of-order or not arrive at all. This problem was difficult to detect and impossible to recover from. We also ignored the warning messages about access point conditions, which represent about 21 percent of the trace.

Campus/SNMP used one polling station to poll the access points every 5 minutes. A SNMP query included statistics about the AP's interfaces and about its associated client cards. We extracted only the list of cards, which represented about 51 percent of the whole trace.

There was a need to detect and remove errors from the two Versus systems and Campus/syslog. A 21% and 49% data reduction could be gained by applying simple filtering for the two Campus systems respectively. In all four systems, timestamps were added by the collection processes using the host's own clock when data arrives. Considering daylight savings time, the timestamp may need to be transformed to avoid confusion (non-continuous near clock changes) for timestamp-sensitive applications. Merging and correlation was necessary for reliability (Campus/syslog) and scalability (Campus/SNMP). It is possible to set up several Versus systems (although we did not) to cover a large building, and then the location system has to merge all the data streams from every root

	Maximum	Minimum	Mean	Std. dev.	C.V.
Versus/IR	29,341	3,093	15,531	6,993	45.0%
Versus/RF	11,395	6,347	8,730	1,095	12.5%
Campus/syslog	563,940	83,344	299,650	81,130	27.1%
Campus/SNMP	365,290	48,896	314,320	51,541	16.4%

Table 6.2: Statistics of daily location-update traffic. C.V. is the standard deviation divided by the mean value (coefficient of variation).

concentrator to answer queries such as "Where is badge A?". *In summary, data pre-processing plays a vital role in localization systems*.

6.3.2 Data volume

We compare the location-update traffic generated by the four systems in Table 6.2.

We see a relatively small variation in the daily location updates for Versus/RF and Campus/SNMP. Considering that the update rate was more-or-less fixed for each (pushed by badge every 2 minutes in Versus/RF and pulled by the SNMP poller every 5 minutes in Campus/SNMP), this indicates that the number of badges and cards present daily had small variation too. On a typical day, Campus/SNMP produced more than 3.6 messages per second, given a total 6006 cards and 543 APs seen during the whole trace.

Campus/syslog produced a smaller amount of update traffic than Campus/SNMP, given that the two traces were collected for same period. It is not surprising, since syslog only produced updates when the card associated and disassociated with an access point, and such visits often lasted longer than the 5-minute polling interval.

Versus/IR had a relatively large variance on daily updates, due to the significant difference in update rate when the badge is in motion (every 3.5 seconds) or stationary (every 4 minutes). In theory, N badges can produce $(N/3.5)$ updates per second if there are no missed pings caused by interference and line-of-sight problems. That is 10 updates per second with 350 active badges. Clearly, tracking all the assets and people in a multi-site large organization will be a challenge for a location service.

Location-aware applications resident on a mobile device will not have the resources to handle this amount of traffic. *Instead, a software infrastructure is necessary to collect, process, and disseminate the location updates to applications.* This infrastructure needs to keep up with the update arrival rate and control the amount of information delivered to applications. It can shield the data pre-processing from applications, and share the results with multiple applications. Several major research efforts specifically target this direction [119, 50, 68]. In the case where the data rate still may outrun the capability of the infrastructure, approximation techniques have to be applied to reduce the data stream (Chapter 3).

6.3.3 Load disparity

Many location-aware applications may only care about location updates from a particular zone or from a particular user, to answer queries such as "Who is in zone A?" or "Where is user X?". We decompose the trace to show the load distribution of such queries. We compute the fraction of

Figure 6.1: [Versus] Distribution of location updates across badges and sensors.

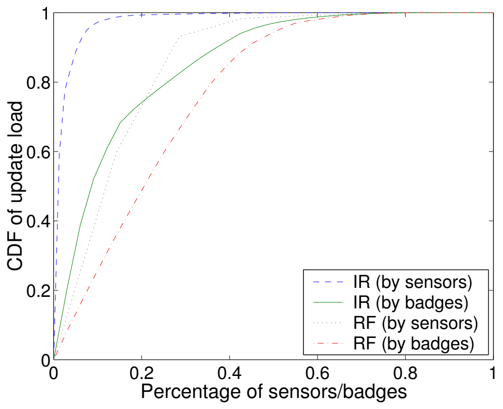
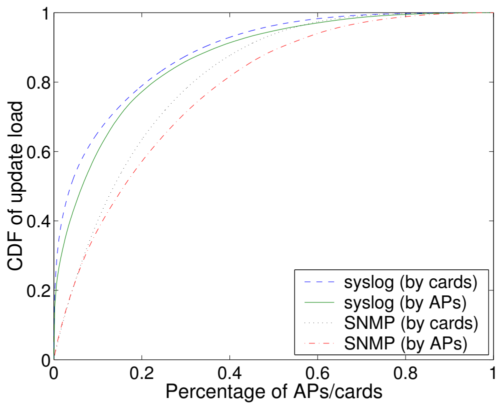


Figure 6.2: [Campus] Distribution of location updates across cards and access points.



load updates generated by the individual users, and sort them in decreasing order. We then plot the cumulative sum of the sorted vector as the Y axis. A point is then read as "the top x fraction of users generated fraction y of location updates", resulting in a Cumulative Distribution Function (CDF) plot. We did the same thing to separate the traffic for the zones.

Figure 6.1 shows the results for the two Versus systems. In Versus/IR 8% of the sensors received about 95% of the updates, indicating about 7 hot zones in our building. In fact, these were the offices of the badge owners. Versus/RF, however, does not have such a deep curve, since although the user population was concentrated on a small number of rooms, they are more distributed among the radio cells. About 40% of the badges generated 90% of the updates in Versus/IR; these more "active" badges were worn by the author and attached to some office chairs. By active, we mean the motion sensor in the badge reported badge movements. Interestingly, one badge attached to a stationary laptop and another to a stationary water cup, both resting on the author's desk, also generated a large number of updates. This result indicates that the motion sensor in the badge was rather sensitive, so these two badges appeared to be active when the user was working on the desk. The number of updates by Versus/RF was insensitive to the badge state (active or not) since the update rate was fixed (every 2 minutes), so its curve is closest to linear. The small variation was caused by occasional absence of some badges from the building.

Figure 6.2 shows load disparity across access points and cards on campus. Campus/syslog has a knee near 10% of the cards that generated 65% of the location updates, while the Campus/SNMP has a smoother curve. The syslog curve is more skewed than the SNMP curve because it captures short-term mobility caused by a few cards that are changing APs frequently. These cards were indeed mobile, or stationary but trying to "walk" through adjacent access points to find one that has better signal strength/quality. Load distribution across access points by Campus/syslog and Campus/SNMP is comparable, while the former is more skewed than the latter, for a similar reason that the APs in hotspots captured more short-term sessions.

We observe this location-update load disparity across both zones and users in all of the four systems. The localization systems themselves need to be aware of this situation. A sensor hierarchy like Versus's sensor-collector-concentrator may experience hot paths in the system. The buffer size along these paths need to handle peak traffic not seen in a test phase. Any software infrastructure, such as a lattice-based location service [76], also will see similar phenomena. A virtual counterpart,

Figure 6.3: [Campus] Distribution of average sighting intervals in minutes.

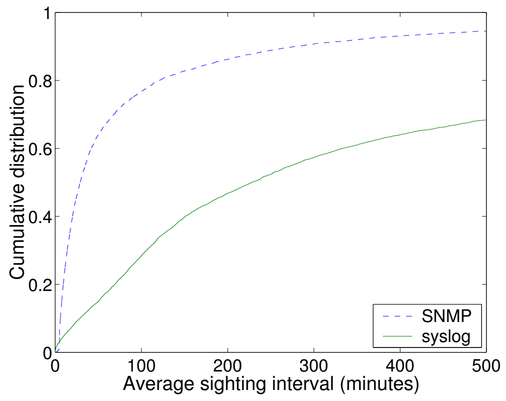
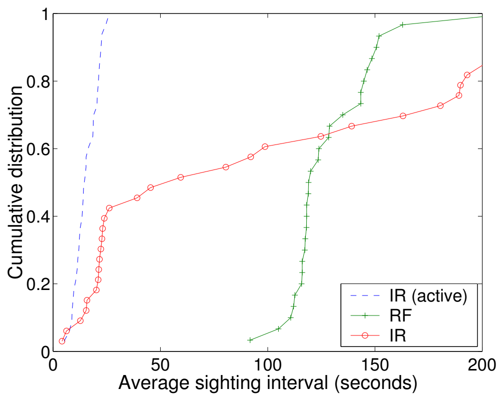


Figure 6.4: [Versus] Distribution of average sighting intervals in seconds.



such as an agent, for these "hot" zones and active users will experience more location updates and possibly more location queries due to popular interest. On the other hand, applications also could take advantage of this locality property. For example, popular information could be cached or prefetched.

6.3.4 Sighting interval

A *sighting interval* is defined as the period between two consecutive location updates for one user. For the Versus systems, we apply a cut-off period of 30 minutes, since longer intervals indicate the badge was out of the building. We also took advantage of the motion field in the Versus events. All the intervals during an idle period (defined by a sequence of 3 or more consecutive motionless reports) were thrown away, until the badge was active again. We used this heuristic to exclude the intervals when the badge was placed on table while the user was away. We apply this approach to Versus/IR and call it "Versus/IR (active)". For Campus/syslog, we use "associated" and "reassociated" to demark the boundaries of sighting. We also treat two messages as a special location update: "deauthenticated" and "disassociated". All other syslog messages are ignored in this analysis. We did not apply any cut-off period for the Campus/SNMP trace. Figure 6.3 and Figure 6.4 show the cumulative distribution of the average sighting interval (across cards and badges) for Campus and Versus systems.

The two Campus systems have rather different sighting intervals due to the nature of their location-update mechanism. We found the mean of Campus/SNMP to be 145 minutes, median to be 29 minutes, and the maximum to be more than 18 days. The mean of Campus/syslog was 830 minutes, the median was 228 minutes, and the longest interval is more than 21 days. One interpretation of the difference is that, although we purposely collected the traces over the same period of time for these two campus traces, the set of access points showing up in the two traces are not the same as we mentioned before. Another reason is that the SNMP could not detect card departure, leading to much longer interval when the card came back later.

The mean of the average sighting interval for Versus/RF was 129 seconds, roughly about the fixed update rate (every 2 minutes). The mean average sighting interval for Versus/IR was 106 seconds with a tail out to 511 seconds. Notice this tail is much smaller than our 30-minute cut-off period, indicating that if the badge happened to leave the building, it always stayed out for more

Figure 6.5: [Versus] Distribution of missed pings for various badges.

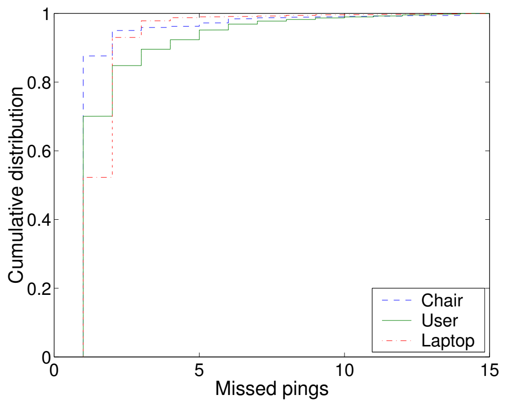
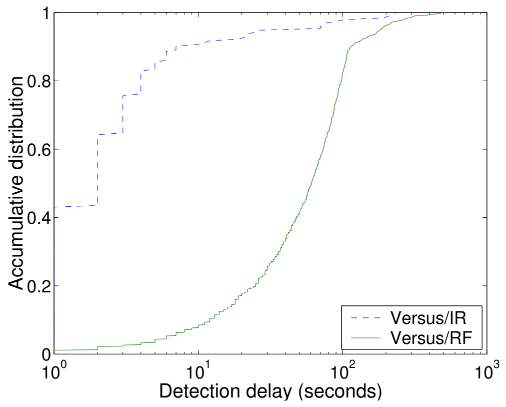


Figure 6.6: [Versus] Distribution of detection delay for two badges.



than half an hour. The mean of the average sighting interval for Versus/IR (active) was 15 seconds, an order of magnitude smaller than the previous two.

Infrared-based localization systems have a well-known line-of-sight problem. Obstacles block the IR signals and potentially increase the sighting intervals. The Versus system encodes a built-in sequence number in location updates that can be used to detect the number of missed pings. Figure 6.5 shows the cumulative distribution of missed pings for three badges: one attached to an office chair, one worn by a user, and another one attached to a laptop. The zero value of missed pings are not counted since they were dominant in the trace (for instance, the total missed pings for the badge worn by the user is about 17 percent). We see that a small number of missed pings (1 or 2) are frequent. This estimation, however, is only an approximation, given that the sequence number is only 4 bits. A shielded active badge will easily outrun the sequence space in $(16 * 3.5) = 56$ seconds. Starting in March, the author wore two badges, one on the wrist and the other pinned on his chest. We found that the chest badge lost about 18 percent of the pings, similar to the one on the wrist (17 percent loss).

We now do a simple analysis of detection latency. Assume that a random variable X, denoting the time between a badge entering a zone and the system receiving its ping, is uniformly distributed over 0 to n seconds. The mean of X then is $n/2$. Now assume that two statistically independent badges enter the zone simultaneously with detection time X and Y, each following the same uniform distribution over 0 to n seconds. It is provable that the mean of the detection-time difference of the two badges $|X - Y|$ is $n/3$. We compute the empirical detection difference of the two badges worn by the same user whenever he enters a new zone (so both badges are active) and plot the CDF in Figure 6.6 (note x axis is in log scale). The n for Versus/IR (active) and Versus/RF can be approximated doubling the mean in Figure 6.4, which are respectively 31 seconds and 258 seconds. The mean detection differences from Figure 6.6 are 9.6 seconds for Versus/IR and 71 seconds for Versus/RF, roughly following the $n/3$ rule. This significant detection lag should be taken into account by any application that requires spontaneous interaction using collocation information. In particular, applications like a memory-aid [113] or active reminder [49] may miss some events and fail to act if two people only briefly meet in the hallway.

The sighting interval is also a useful metric to measure the freshness or confidence of the location information. The more frequently the system obtains a user's location updates, the more confident it can be of the user's current location. One active map application uses image fading and frame shading during the sighting interval to degrade the information freshness gracefully [90]. The detection latency also may have impacts on many applications, such as a reminder application that uses collocation [49], teleporting [114], and call forwarding application that follows user's current location [137]. None of our four localization systems provided an explicit location update when the user left the covered area (the "deauthenticated" and "disassociated" messages in Campus/syslog are not reliable, and the most reliable message arrives 30 minutes after the user leaves). Thus applications have to infer this situation by applying some threshold on how long there was no update before the user's location becomes unknown. If not arbitrary, this threshold often relates to the expected value of the sighting interval of that localization system.

The sighting interval is a complicated metric and is jointly determined by the localization system design, the characteristics of the deployed environment, and user behavior. While in Versus/IR the percentage of missed pings was similar with a badge worn on the chest or on the wrist, it is conceivable that a user wearing the badge at the belt will suffer more lost pings. Depending on the arrangement of access points, the location of the wireless card and the environmental interference, the card may cause more location updates by trying to associate with an access point with the best signal quality. While Campus/SNMP had a fixed 5-minute polling interval, a card can be seen more often, or less often, if it moves around. Versus may report arbitrarily long sighting intervals if the badge is taken out of the building, or put in a drawer by the user, or if the battery died without being noticed.

6.3.5 Prevalence

The *prevalence* of a zone in a user's trace is the measure of the fraction of time that the user spends within that zone. So for one user, there is one prevalence value for every zone she ever visited in the trace, and the sum of all the prevalence values for that user is 1. This metric is defined and used to characterize user mobility in a corporate wireless LAN study [9]. In their study, a user is categorized as "highly mobile, somewhat mobile, regular, occasionally mobile, and stationary" by segmenting maximum and median prevalence into several bins to see to which bin that user belongs. We compute the maximum and median prevalence for each user, and Figures 6.7–6.10 show the scatterplots of every user for the four localization systems. The segmentation lines (using the values from [9]) are drawn in dashed lines. To avoid overplotting of same-valued points, we added randomized jitter in range of $[-0.01, 0.01]$ on both x and y axis for every point.

We first look at the two plots for the Versus systems (Figures 6.7 and 6.8). The total time for a user is the time the badge was active in the system. So if the badge was left on the table unused, we excluded the period of idle time from the prevalence computation. Due to the definition of "median," there will be no users with median prevalence greater than 0.5 except for one special case: both the median and maximum prevalence were 1 (the user only visited one place). We see that there are more such stationary badges in Versus/IR than Versus/RF. At first this may seem odd since if one badge is stationary in one IR zone, it should also be stationary in a bigger RF zone. The explanation is that we have overlapping RF zones, so a badge's ping received by more than one RF sensor can be reported to be any one of the RF sensors receiving that ping, although the exact policy about when and which sensor in the group Versus would choose is unclear to us. The badges with median prevalence equal to 0.5 means that the badge only visited two zones, and the maximum prevalence shows how biased the badge was toward one of them. The author, and another user who

Figure 6.7: [Versus/IR] Scatterplot of prevalence of badges.

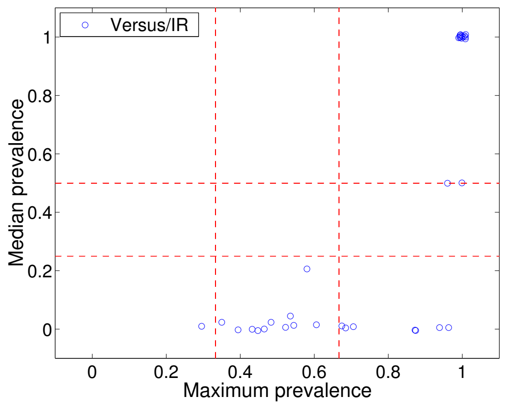
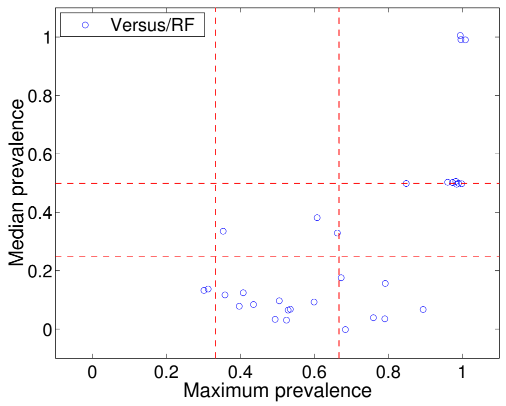


Figure 6.8: [Versus/RF] Scatterplot of prevalence of badges.



wore a badge consistently, show up in the bottom right corner, indicating they spent most of their time in their offices, while they spent relatively little time on other sensors (such as in hallways). Other badges with low median and maximum prevalences indicate that their time was more evenly distributed across several sensors. The badges seen by Versus/RF have relatively higher prevalence, since there were only 7 radio cells.

The two plots for Campus (Figures 6.9 and 6.10) bear strong similarity and show some obvious patterns. First, there were about 11 percent of the cards that never moved during the lifetime of the two traces. Second, nearly 9 percent of the cards had 0.5 median prevalence, which means that they spent their time across two access points. While some of these cards may indeed be mobile, we believe many of them just flickered between two access points when neither signal quality was good enough. Third, the line $P_{median} = 1 - P_{max}$ bounds all the points on the right side when P_{median} is smaller than 0.5. This rule always holds, although we skip the proof here. It is interesting to see that points actually flock to this line. The reason, we believe, is that the card flickered through 3 APs and was biased strongly towards two of them; thus P_{median} approaches $1 - P_{max}$. There is another common line in both plots: $P_{median} = (1 - P_{max})/2$. This phenomena was caused either by the card flickering between three APs while spending even time on the two besides the one with P_{max}, or the card flickering through 4 APs with one of them getting very little time.

The explanation above makes the point that a card may change its associated AP when it was not actually moving, which is a significant phenomena. The network-based optimizations used by the cards actually cause complications and confusion for localization systems trying to determine the current place of a flickering card. *Using a system designed for a different purpose to infer object location sometimes conflicts with our goal and complicates the task.* This suggests that our take-as-is approach to these systems has limited usage, if without advanced processing such as correlation between several APs [79]. Other techniques that explicitly involve user interaction during these indeterministic times also have been investigated in various applications [42].

We compute the percentage of the users in each mobility category (defined using median and maximum prevalence, as in [9]) for all four location systems and show the results in Table 6.3. Each entry has the following format: (Versus/IR, Versus/RF), (Campus/syslog, Campus/SNMP). Versus/IR and Versus/RF traced over the same user population over the same period, and we can see they categorize the users comparably. The main difference seems to be that Versus/IR had more

Figure 6.9: [Campus/syslog] Scatterplot of prevalence of cards.

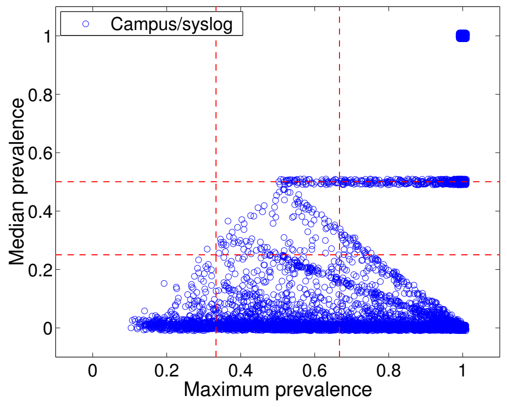
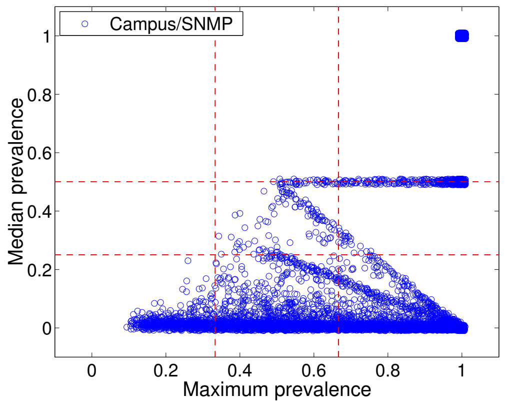


Figure 6.10: [Campus/SNMP] Scatterplot of prevalence of cards.



	Maximum Prevalence (P_{max})		
Median Prevalence (P_{median})	**Low** $P_{max} \in [0, 0.33)$	**Medium** $P_{max} \in [0.33, 0.66)$	**High** $P_{max} \in [0.66, 1)$
High $P_{median} \in [0.5, 1)$	N/A	N/A (0%,0%), (1%,1%)	stationary (43%,32%), (19%,20%)
Medium $P_{median} \in [0.25, 0.5)$	N/A	regular (0%,6%), (2%,2%)	N/A (0%,3%), (1%,0%)
Low $P_{median} \in [0, 0.25)$	highly mobile (3%,6%), (9%,11%)	somewhat mobile (33%,33%), (28%,28%)	occ mobile (21%,20%), (40%,38%)

Table 6.3: Mobility matrix of four localization systems using prevalence metric.

stationary users while Versus/RF had more highly mobile users. This result is not too surprising given that a badge may be reported to show up in more than one RF cell even if it is stationary in the intersections of RF zones. We also captured the traces for Campus/syslog and Campus/SNMP over the same time period with roughly the same user population (the set of APs seen by two systems are slightly different; possibly some are misconfigured and not responding to either syslog or SNMP queries). Again, these two systems divide the user population into mobility categories similarly. This analysis leads us to conclude that *user mobility was well captured by these heterogeneous localization systems*, even though they had different granularity (Versus) and location update mechanisms and rate (both Versus and Campus). The perceived user mobility between Versus and Campus also seems comparable, though they operated at a different scale and tracked a different user population.

Finally, we note that all of the four localization systems really are tracking devices, not humans. The devices are either badges, laptops, or handhelds, and the users may sometimes not carry the device when they move around. We are unable to draw any conclusions about this behavior without a more careful study of user behavior.

Chapter 7

Application Studies

Many undergraduate and graduate students have used Solar to develop applications for their pervasive-computing seminars, senior honors theses, and other research projects. In this chapter we give a detailed case study of one Solar application, which automatically controls a desk telephone based on whether there is an ongoing meeting in the office. We discuss our experiences and lessons learned with this application. Finally we also briefly go over some other Solar applications in the domains of *smart spaces* and *emergency response*, which we have been developing.

7.1 Meeting detection

We are interested in detecting meeting status in an office environment, a capability that we expect will be useful for two classes of applications: 1) applications that help the user to control devices in the room, such as programming the phone to send incoming calls to voice-mail, using audio and video recorders to record the proceedings of a meeting, or controlling the projector, lights, microphone, or other devices in the meeting room; and 2) applications that help a meeting-room scheduling system monitor every room's situation, to know if a meeting is actually in progress. This scheduling system can allow last-minute bookings based on real-time information about availability.

Detecting high-level context such as user meetings is a challenging problem. Although a user's calendar may provide some hints, it is an unreliable source since many users fail to update their calendars consistently and promptly. Instead, we choose to detect meetings with embedded sensors. In this section, we present the sensors we selected, the detection algorithm, and the performance evaluation of our prototype. We built a telephone controller that redirects all incoming calls to a pre-configured voice mailbox so that meetings will not be interrupted. The telephone controller makes a decision to switch to voice mail or not based on the output from our meeting detector. Our telephone controller works with Cisco IP phones by logging into a configuration website and reconfiguring the call forwarding settings whenever a meeting begins or ends.[1]

7.1.1 System design

Our goal is to detect the beginning and ending of a meeting, in near real time, without expecting every user that might attend a meeting to wear special hardware such as location-tracking badges.

[1]http://www.cisco.com/en/US/products/hw/phones/ps379/

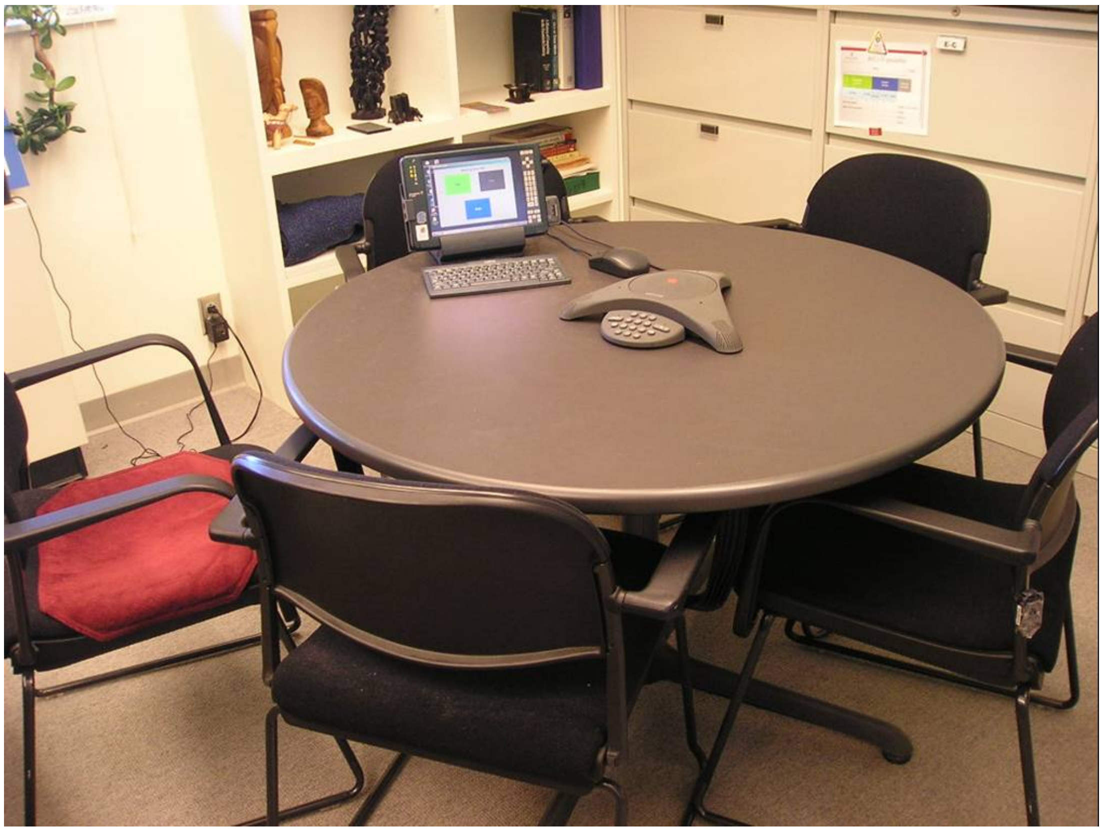


Figure 7.1: An instrumented meeting space with a controlled telephone that automatically transferred incoming call to voice mail without interrupting ongoing meetings.

It is not surprising that many regular building occupants refuse to wear a badge [60], and visitors in our open academic building, of course, never wear a badge. We designed the meeting detector to detect a meeting's status by aggregating two kinds of sensors input. We first introduce the system architecture and hardware setup, and then describe the operator graph in detail and discuss the fusion algorithm. (Most of this work was done by Jue Wang).

The meeting detector is designed to detect a meeting in a user's office or a conference room (see Figure 7.1). Our prototype is designed for a small meeting area in a typical office, which contains a table and some chairs. We define "a meeting in progress" when at least two chairs are occupied. We detect chair occupation using pressure and motion sensors attached to the chairs. This approach assumes that the motion or the pressure are produced by human occupation, which may fail in the cases that somebody bumps the chair, causing the motion signal, or somebody places a heavy object on the chair, causing the pressure signal. It is thus necessary to combine these two sensors to achieve a more reliable result.

We added both sensors to the chairs to detect the pressure of a seated person, and the motion of the chair caused by a seated person (see Figure 2.3). To reduce cost, we use only one pressure mat (on the chair that is always occupied in meetings by the office resident). The pressure mat on the seat of one chair uses a wireless transmitter to communicate its status to a nearby receiver every

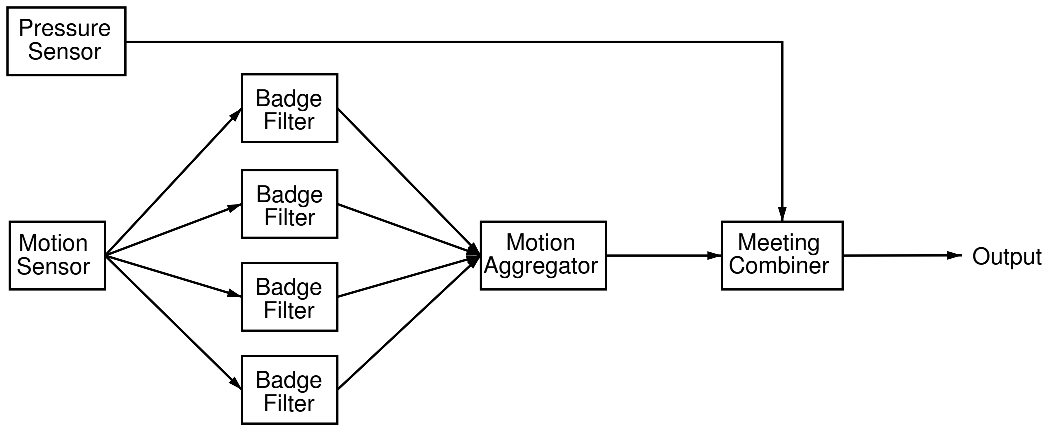


Figure 7.2: An operator graph used for meeting detection with sensor fusion. The meeting table had four chairs with motion sensors, and one chair with pressure sensor.

120 seconds (preset in hardware).[2] Software on the receiver computer sends each pressure reading into Solar as an event and logs these data automatically. We call this log the "pressure log."

To detect chair motion status we used an existing commercial location system,[3] in which "personnel badges" periodically send IR updates to ceiling-mounted sensors. These badges happen to have an embedded motion sensor, intended to conserve power when the badge is stationary. A badge update packet contains the status of the motion sensor so we can tell if the badge has recently moved. We monitor the ceiling detectors and translate badge packets into Solar events. The motion sensor is quite sensitive and a seated user will usually trigger the sensor except when sitting very still. The badge sends updates every 3.5 seconds when it is moving and every 2 minutes when it is stationary. All the motion data is logged at a server and we thus obtain a "motion log."

Using these two kinds of sensor data, an operator graph running on Solar determines the meeting's status. The operators filter the data from the motion sensors and combine it with the data from the pressure sensor. We describe the operator graph first and then the fusion algorithm. The meeting detector is an operator graph, which mainly consists of two layers of filters and a combiner (see Figure 7.2).

Before Solar can use raw data from sensors and other data collection devices, the data must be converted into attribute-value pairs within a Solar event. The meeting detector has two Solar sources, a motion source and a pressure source; each reads raw sensor data and uses the Solar API to publish an event object for each sensor reading. Events from the pressure source have two attributes, indicating whether pressure is detected by the sensor and a timestamp. Events from the motion source contain a badge number, its motion state, and a timestamp. Since the motion source collects the updates from all badges, even those on chairs in other offices or those not attached to chairs, we use the Badge Filter to pass on events only from the badges we used in the meeting room (each filter matches a particular badge number). The Motion Aggregator keeps an internal state about whether there is any movement of chairs and publishes an event whenever the number of moving chairs changes.

The Combiner is the key to the operator graph in meeting detector. Its goal is to output an event

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

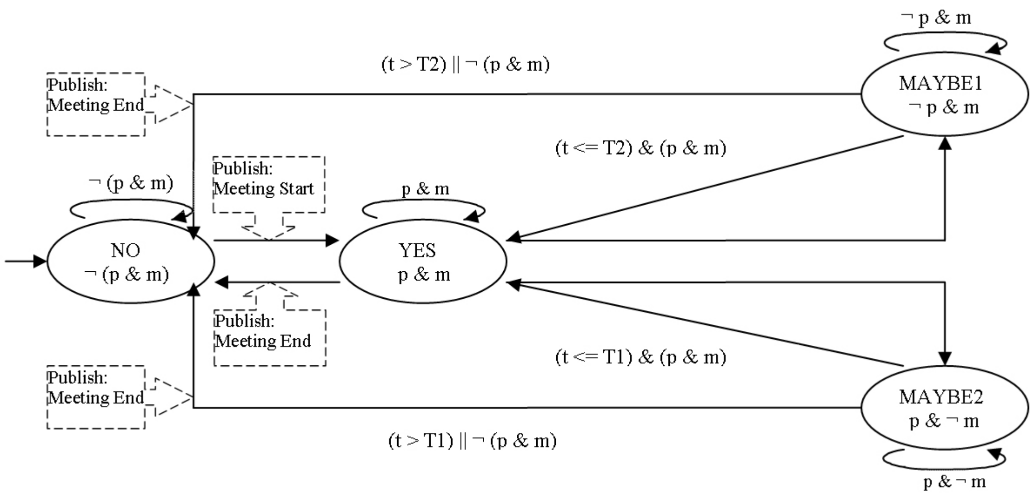


Figure 7.3: A state machine illustration on sensor combination algorithm. The meeting table had four chairs with motion sensor, and one chair with pressure sensor.

whenever a meeting begins or ends. It receives simple events from the filters above, and internally records the most recent observed state: p means there is pressure on the mat and $\neg p$ means no pressure; m means there is motion in some chair and $\neg m$ means no motion. Figure 7.3 shows how we combine the pressure and motion data to detect meeting start and end, using a state machine. Here t is the time since the last state change.

We have four states: NO, YES, MAYBE1 and MAYBE2. NO means no meeting is in progress and YES means a meeting is in progress. Initially, we wait until there is pressure and motion at the same time before we decide that a meeting has started. We declare the meeting over if there is neither pressure nor motion in the most recent readings. MAYBE1 and MAYBE2 mean the meeting may have ended, but we hesitate to make a decision immediately. These “hesitation” states deal with the intermittent reports of no pressure or no motion even when a meeting is in progress, because of unreliable sensor outputs. We use two thresholds to control transitions out of these hesitation states. Threshold T_1 is for the state that there is pressure but no motion, and T_2 is for the state that there is motion but no pressure. In MAYBE1 if t exceeds T_1, or in MAYBE2 if t exceeds T_2, we assume the meeting has ended despite one sensor indicating a meeting is in progress. These timeouts were necessary because sensors occasionally “stick”. We set T_1 to 10 minutes because we found that the motion sensors often report a chair as stationary even when occupied: sometimes people sit very still. We set T_2 to 1 minute because our pressure mat had several unexplained 1-minute gaps even when the chair was occupied. These thresholds help reduce false reports of meeting end.

Essentially, we decide that a meeting is in progress whenever both pressure and motion are detected (p & m). Because our sensors occasionally report intermittent lack of pressure or motion, the state machine hesitates to declare “no meeting” immediately when one is false, so we include two MAYBE states for the cases $p \ \& \ \neg m$ and $\neg p \ \& \ m$. If the $p \ \& \ m$ condition returns before a timer exceeds a threshold, the meeting continues in state YES. If too much time elapses, we declare the end of the meeting.

7.1.2 Evaluation

In this section, we report the performance of the meeting detector. For this study, we test the sensitivity and accuracy of the meeting detector by matching the detected meeting records from the meeting detector and the real meeting records from a manual log of a several-week period of actual meetings.

We tested the meeting detector in a professor's office with real meetings and real people during two periods in September 2003 and February 2004. To obtain the "ground truth" about actual meeting start and end times, we implemented a simple Meeting Recorder, which we installed on a tablet placed on the meeting table. This application allows the professor to, with a single button, manually log every meeting's start and end. This results in a "meeting log." This log has some incorrect records due to human mistakes (failing to record a meeting, or starting and ending late). We discard these incorrect data and the corresponding data in the "pressure log" and the "motion log."

In total we have three log files: the "meeting log," the "pressure log," and the "motion log." By matching the output of the meeting detector (using pressure and motion logs as input) with the meeting log, we can measure how well the meeting detection works using three different detectors: 1) we used only pressure data to detect meetings; 2) we used only motion data to detect meetings, that is, when at least one chair is in motion; and 3) we combined motion data and pressure data using the algorithm in the previous section. We compare the outputs of all three detectors with the real meeting log.

It was not immediately clear what might be an appropriate metric to measure how well and how quickly detection occurs. For a context classification component, we are interested in both its accuracy and sensitivity. We define several metrics to measure the matching accuracy comprehensively, as shown below and illustrated in Figure 7.4.

1. Delta Start, the difference between real meeting start time and detected meeting start time.
2. Delta End, the difference between real meeting end time and detected meeting end time.
3. Gap Length, the time length of a gap between two detected meetings, where one real meeting is divided into two or more detected meetings.
4. Gap Number, the number of gaps between detected meetings for one real meeting.
5. Missed Gap Length, the time length of a gap, which is not detected, between two real meetings.
6. Missed Gap Number, the number of gaps, which are not detected, between real meetings.
7. Extra Meeting Length, the total time of extra detected meetings (which map to no real meeting) per day.
8. Extra Meeting Number, the number of extra meetings per day.
9. Missed Meeting Length, the time length of a real meeting thoroughly undetected.
10. Missed Meetings, the number of missed meetings, which are thoroughly undetected, per day.

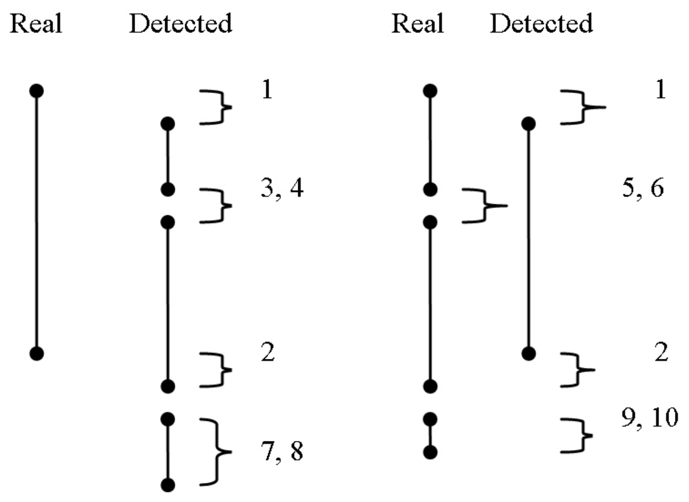


Figure 7.4: Definitions of evaluation metrics for meeting detection.

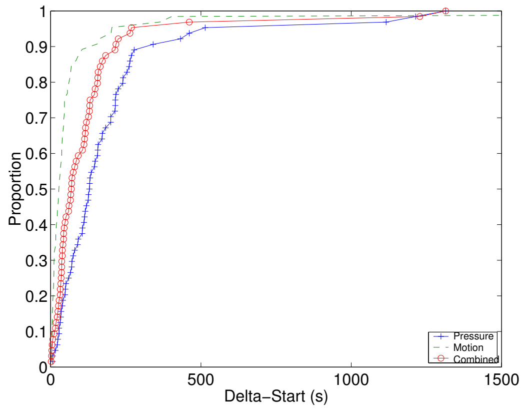
In Figure 7.5 we present results of metrics (we only show six here because cases 5, 6, 9 and 10 happened rarely so there is little data to present). Each plot has three curves, which show the cumulative distribution function (CDF) of the quality (one of the metrics) of a meeting detector on the set of points representing real meetings. In all metrics, smaller (to the left) represents better detection performance.

We show the performance of the three detectors, for the different metrics, in Figure 7.5.

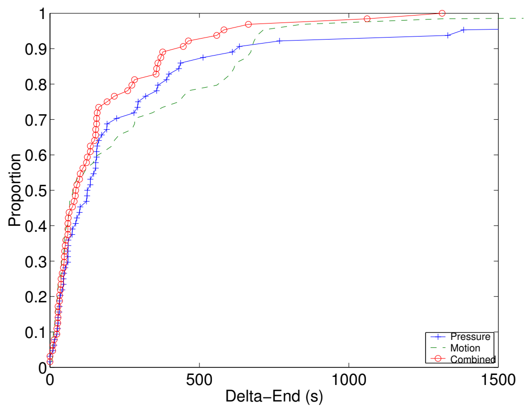
Note that plot (a) clearly shows that the motion detector most quickly detected meetings with a median of 28 seconds,[4] which is 41 seconds earlier than the combined detector and 101.5 seconds earlier than the pressure detector alone. This 41 second latency is the price we pay to avoid the extraneous meetings and gaps, a reasonable trade-off for many applications. The combined detector shows its advantage in (b) with a median Delta End of 88 seconds. Overall, we found that our algorithm did not work as well in detecting meeting end time as it did the start time. This asymmetry arose because of the necessary MAYBE states and their associated thresholds. Those hesitation states did effectively reduce extra meetings and gaps, however, in both length and number, as shown in (c), (d), (e) and (f). Plot (c) shows that the pressure detector had the longest extra meeting length because sometimes the pressure mat was stuck and thus produced a long extra meeting. Plot (e) shows that the motion detector had the longest gap length, because of the motion sensor's limitation mentioned above. Plots (d) and (f) show that the combined detector was effective, reporting 5% fewer extra meetings and 11% fewer gaps than the motion detector, 8% fewer extra meetings and 16% fewer gaps than the pressure detector. These results show clear advantages of a combined detector that overcomes the limitations of a single sensor and enhances the performance significantly.

Generally, a single sensor was not a good indicator for meetings because of its reaction time, sensitivity or inherent hardware limitations. Although the motion scheme was the most likely to catch the time of meeting start or end due to its sensitivity, that same sensitivity led it to detect

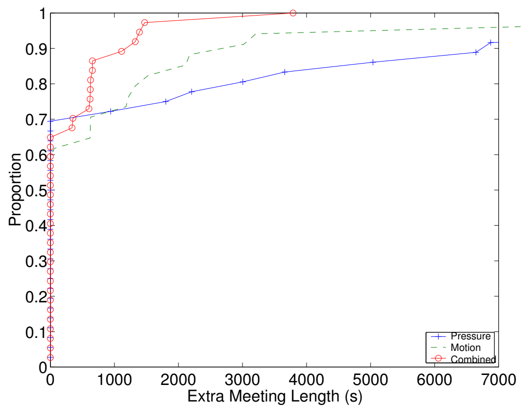
[4]We note that 28 seconds is well within the noise, since the basis for comparison is the manual meeting log and it was not unusual for the professor and others to sit down and get settled for a few seconds before clicking the "meeting start" button.



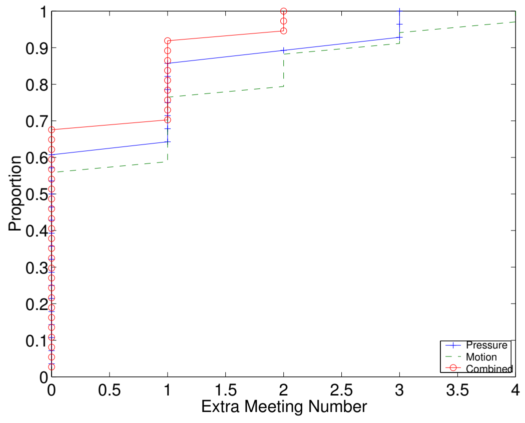
(a)



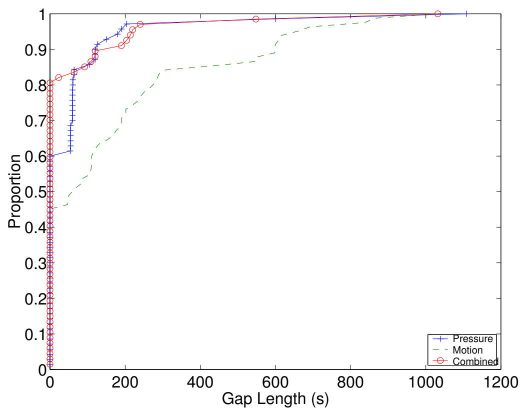
(b)



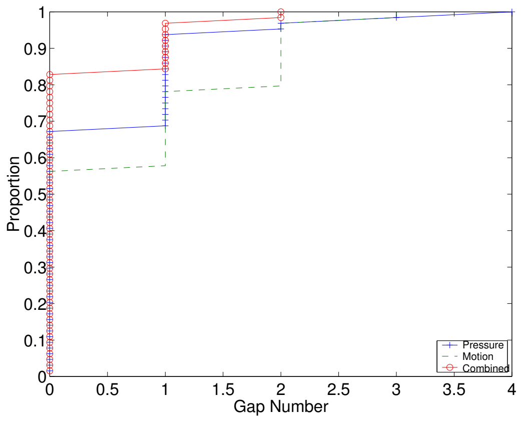
(c)



(d)



(e)



(f)

Figure 7.5: These CDF plots represent results over several different metrics.

meeting end prematurely unless one or more people were moving frequently. It also tended to detect extra meetings when a passersby bumped the chairs. The overall result was many extra meetings and many gaps. On the other hand, the pressure mat was less sensitive than the motion sensors, but avoided the problem of accidental bumped chairs. We found that the pressure mat occasionally got stuck, though, and produced some excessively long meetings. Naturally, a combination was best overall, although it was not the best one for the metric like Delta Start.

7.1.3 Discussion

We expect that our approach to detect meeting context could be implemented at low cost, and would be easy to use and maintain. Wireless pressure and motion sensors, tied to a simple wireless receiver in the room, could easily collect and forward the information via a WiFi or Ethernet connection. If the sensors could be integrated into the chair seat cushion, they should be rugged and easy to maintain. Additionally, no complex sensing infrastructure (such as a location trackers) is required, although we happened to leverage an existing such system for our test. The fusion logic is easily implemented in software or hardware, without any training necessary. Although training may improve accuracy somewhat, it is inconvenient to retrain whenever the meeting environment changes, the room is visited by users with different habits, or when the system is ported to other rooms.

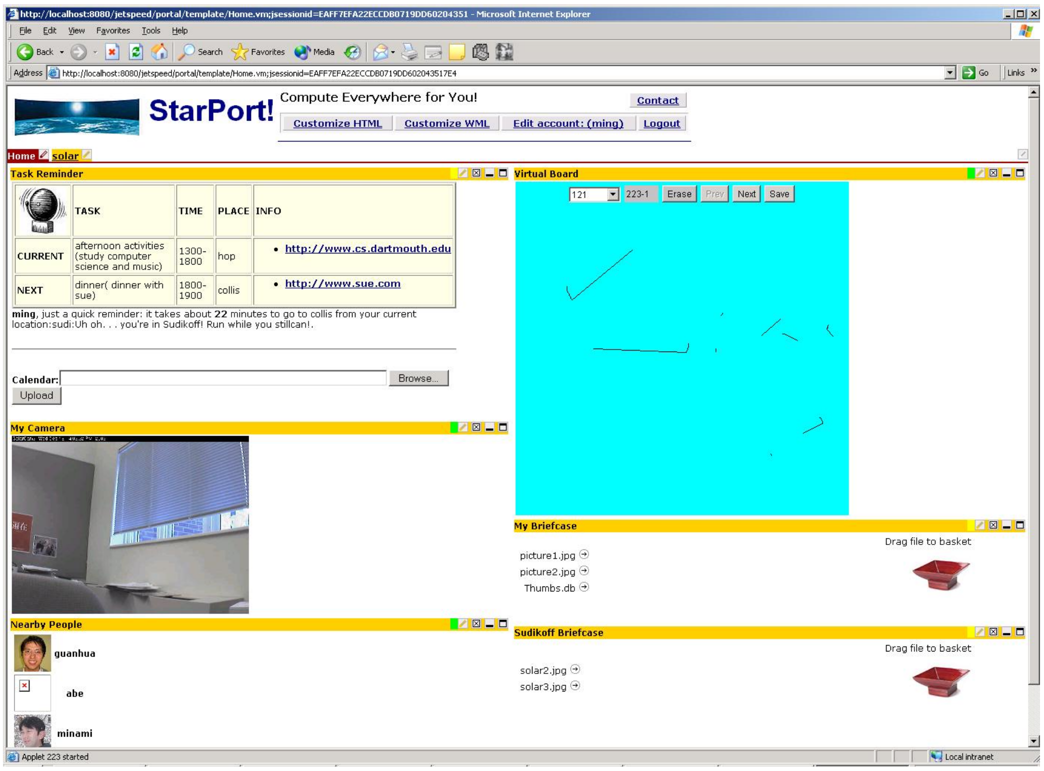
In a typical enterprise, office or conference room chairs cost several a hundred dollars apiece. Recent progress in sensor and wireless network technology should drive the price of our needs (a pressure sensor, motion sensor, and wireless transmitter) to a few dollars per chair and a few dozen dollars per meeting room, in quantity, within a few years. The deployment cost of our approach may well be reasonable since it has many applications.

There are limitations, however. First, detection quality is determined by sensor limitations. The low cost means we sacrifice performance to some extent. Since different applications of a meeting detector require different detection accuracies and sensitivities, the adequate accuracy we achieved in our prototype may not be sufficient for other applications. Some applications may require a highly sensitive and accurate detection within several seconds; but some scheduling applications may be satisfied with accuracy within several minutes. Again, this argues for Solar's approach to allow applications to customize desired information instead of to pre-compute context for all applications.

Second, we note that we chose to define "motion" when one or more chairs moved. Defining "motion" to be when two or more chairs moved might avoid detecting extra meetings, but in our experience this approach does not change Delta Start and Delta End significantly. Careful tuning of this parameter and others should lead to better performance. Finally, in some environments, where many short meetings occur close together, such as frequent five-minute meetings with one-minute intervals, our meeting detector did not work as well as it did in the ordinary case, since our pressure sensor is not sensitive enough to react to these short intervals in time. A better sensor could easily avoid that problem.

7.2 Other applications

We have been developing some other Solar applications, in the domains of "smart spaces" and "emergency response," which are enabled by our building-wide (Section 6.1) and campus-wide (Section 6.2) location-tracking infrastructure.


http://localhost:8080/jetspeed/portal/template/Home.vm;jsessionid=EAFF7EFA22ECCDB0719DD60204351 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost:8080/jetspeed/portal/template/Home.vm;jsessionid=EAFF7EFA22ECCDB0719DD602043517E4
Go Links
StarPort!
Compute Everywhere for You!
Contact
Customize HTML
Customize WML
Edit account: (ming)
Logout
Home
solar
Task Reminder
Virtual Board
121
223-1
Erase
Prev
Next
Save
Calendar:
Browse...
Upload
My Camera
My Briefcase
Drag file to basket
Nearby People
guanhua
abe
minami
Sudikoff Briefcase
Drag file to basket
Applet 223 started
Local intranet

	TASK	TIME	PLACE	INFO
CURRENT	afternoon activities (study computer science and music)	1300-1800	hop	• http://www.cs.dartmouth.edu
NEXT	dinner(dinner with sue)	1800-1900	collis	• http://www.sue.com

ming, just a quick reminder: it takes about **22** minutes to go to collis from your current location:sudi:Uh oh. . . you're in Sudikoff! Run while you stillcan!.

picture1.jpg
picture2.jpg
Thumbs.db

solar2.jpg
solar3.jpg

Figure 7.6: An active Web portal shows customized content to approaching user.

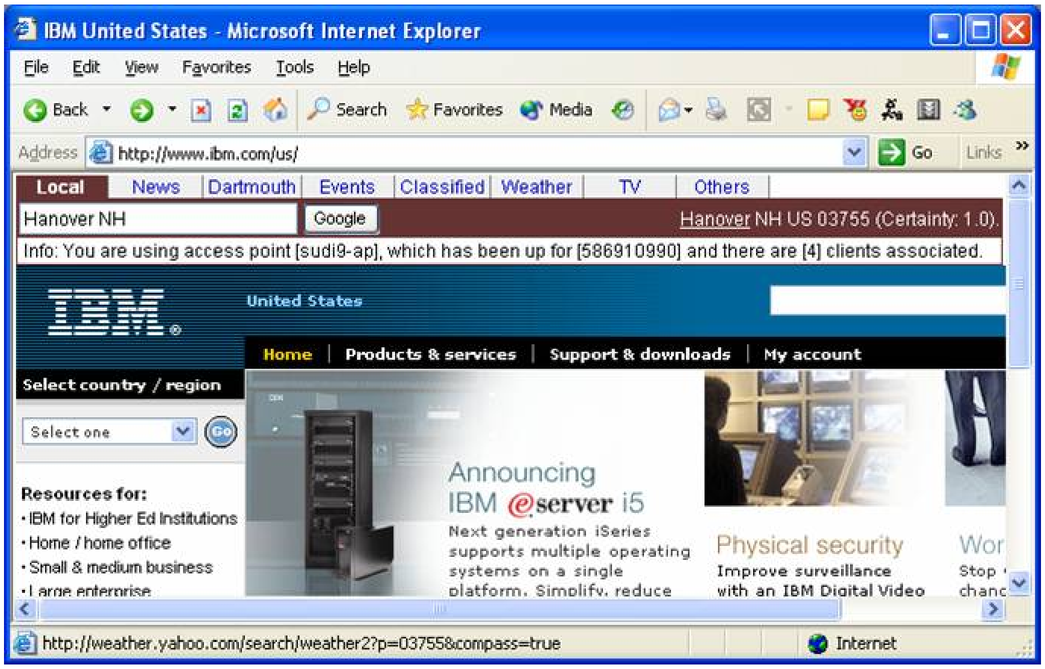
Earlier, we built a "smart reminder" application using Solar [88]. The goal of this application was to determine the appropriate time to alert its user about the next appointment based on the user's current location and the location of the next task on the calendar. This approach is more friendly and efficient than alerting the user, say, 10 minutes prior to the next meeting regardless of the distance to that meeting site. (Arun Mathias implemented this application.)

In the summer of 2002, we assembled a project team to build an active information portal that could customize its layout and content based on the identity of the user and other context information. Figure 7.6 shows a prototype GUI that displays a user's calendar, people nearby, a virtual board associated with the current location, and an interface allowing the user to exchange files between her personal folder and a public folder associated with the current location. The prototype is built with the Apache JetSpeed portal framework.[5] (Adrian Hartline, Ming Li, Kazuhiro Minami, Cal Newport, Libo Song helped to write this application.)

We built a Web proxy that determined the associated access point given the IP address of a wireless client. When the browser on a WiFi device initiated an HTTP request, the proxy determined its rough location based on the IP address in the HTTP header. Thus the browser can push location dependent information to the client; see the top of the returned page in Figure 7.7. The proxy may also automatically fill in some Web forms given the user's current location. (Guanling Chen wrote this application.)

We built a location-aware graffiti application, shown in Figure 7.8, which runs on a user's mobile

[5]http://portals.apache.org/



Info: You are using access point [sudi9-ap], which has been up for [586910990] and there are [4] clients associated.

Next generation iSeries supports multiple operating systems on a single platform. Simplify, reduce

Improve surveillance with an IBM Digital Video

Figure 7.7: The browser on a wireless client can get location-dependent information through an instrumented Web proxy.

device and allows the user to attach scribbles and text messages to her current location. The user also may retrieve graffiti left by others at the current location. A map interface also allows the user to explicitly navigate to find and author graffiti. (Lin Zhong wrote this application.)

Solar currently is being deployed for a new suite of applications, based on an earlier system called the Automated Remote Triage and Emergency Management Information System (ARTEMIS).[6] These emergency-response applications use a large number of data streams from environmental sensors, physiological sensors (attached to victims and responders), and human observer inputs. Related command and control applications use Solar to compute high-level and customized contextual information for the decision makers from different participating organizations. Figure 7.9 and Figure 7.10 show an early application GUI running at a remote EOC (Emergency Operation Center) and on a mobile device carried by first responders, respectively. The goal of this project is to provide situational awareness at all levels of the incident command hierarchy. (Christopher Carella, Michael DeRosa, and Aaron Fiske wrote this application.)

Based on our field studies with responders and local authorities, we have learned that the environment and situation can change quickly. Some of the changes cause automatic operator-graph reconfiguration using our naming and discovery service and dependency management service. On the other hand, some context needs may not be foreseen beforehand. The decision makers may have to quickly deploy new operator graphs in the Solar network using Solar's development and deployment toolsets.

The integration of Solar into this project has only recently begun.

[6]http://www.ists.dartmouth.edu/projects/frsensors/artemis/

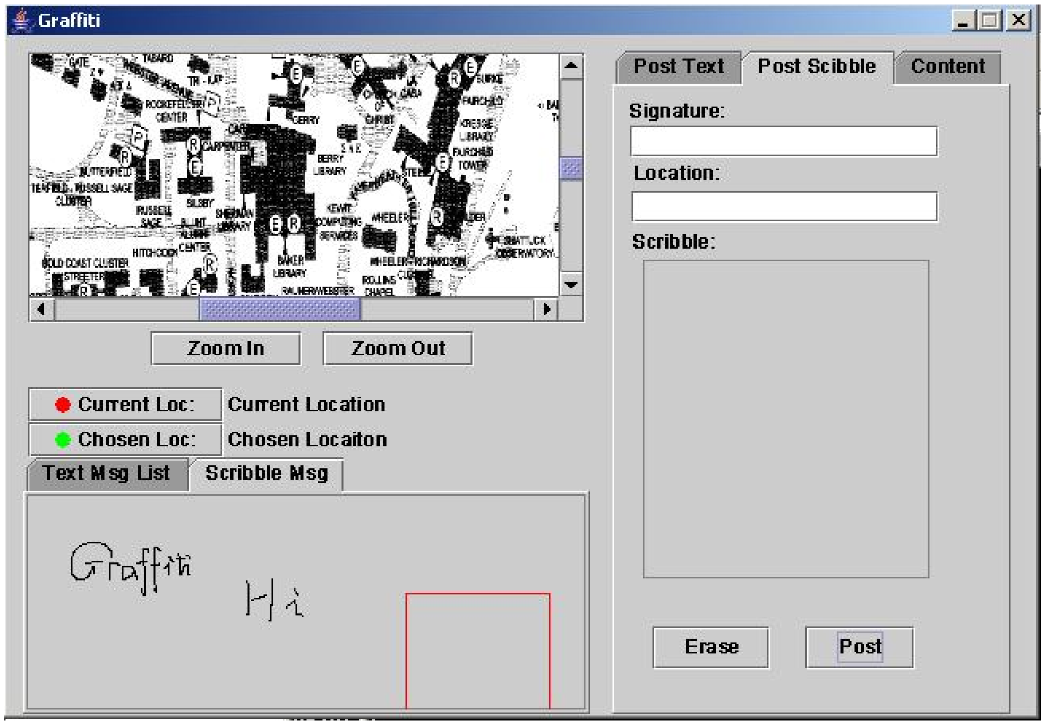


Figure 7.8: The graffiti application running on mobile devices allows user to leave and retrieve scribbles based on current location.

7.3 Experiences and lessons

We note that applications may use context *actively* or *passively*. In smart-space scenarios, many applications take actions directly, according to context changes: the phone controller changed phone-answering behavior based on meeting context. Other applications present the right context at the right granularity to the right people, with decision-making mostly left to human users; the emergency response applications fit in this category. The difference is caused by the complexity of decision making and the cost of mistakes in different environments.

We found that it was relatively easy to convey the operator-composition model to the students in the pervasive-computing seminar course. We have seen few cases of code-based operator reuse, however, except for simple location filtering and transforming. Student project groups made progress in parallel with little coordination, and sometimes ended up with multiple versions of similar operators. For instance-based reuse, we provided a deployed operator that computed all badges' current location and every group used it as a primary source in their operator graphs. Another reason we did not see much reuse is that we did not have many sensors at that time and many projects only needed a location filter. We expect Solar's strength to become more obvious when the environment is instrumented with more sensors and more applications are developed. For instance, we can easily replicate the meeting detection hardware in other offices, and other applications may use the context of meeting status.

The first prototype of Solar was a pure event-driven system, in which each operator was a passive event handler. While this approach is natural for many physical sensors, we found it difficult to

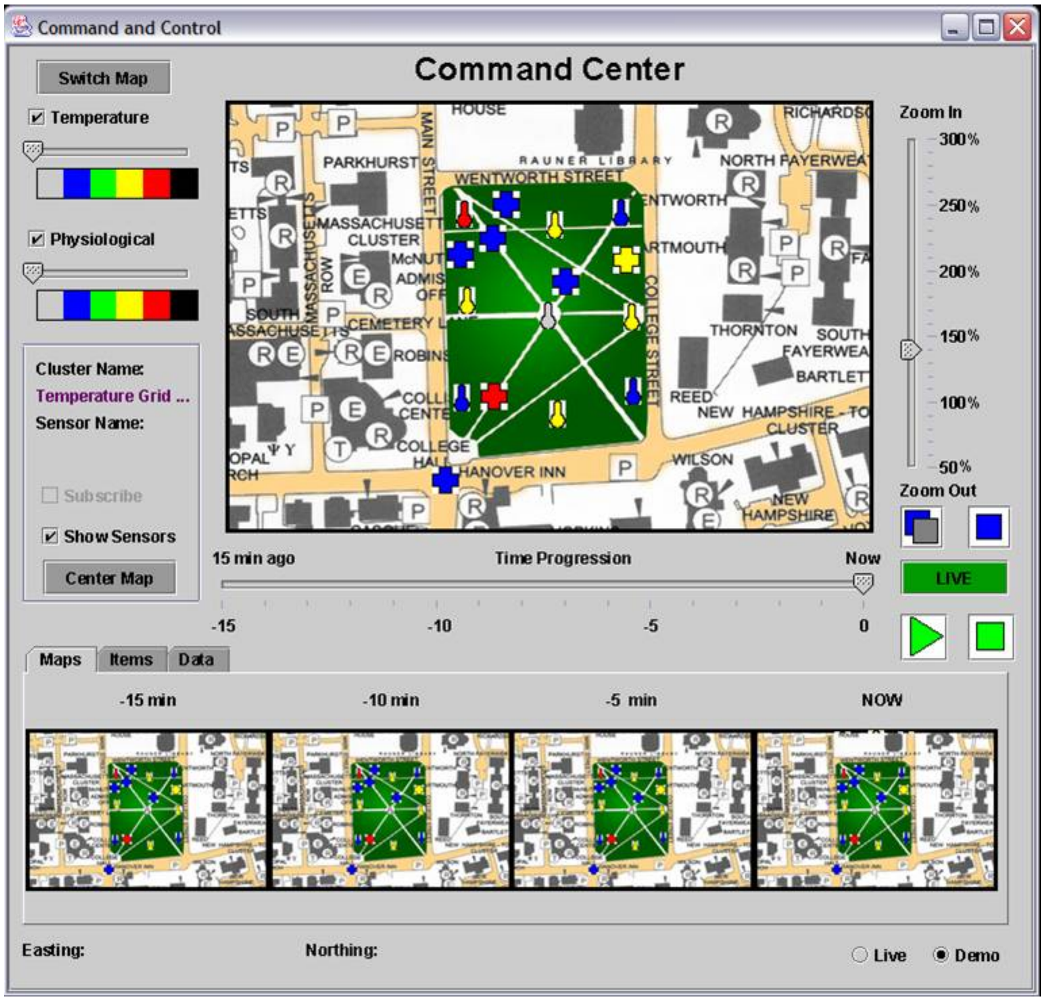


Figure 7.9: A command and control applications running at remote EOC to provide situational awareness for decision makers.

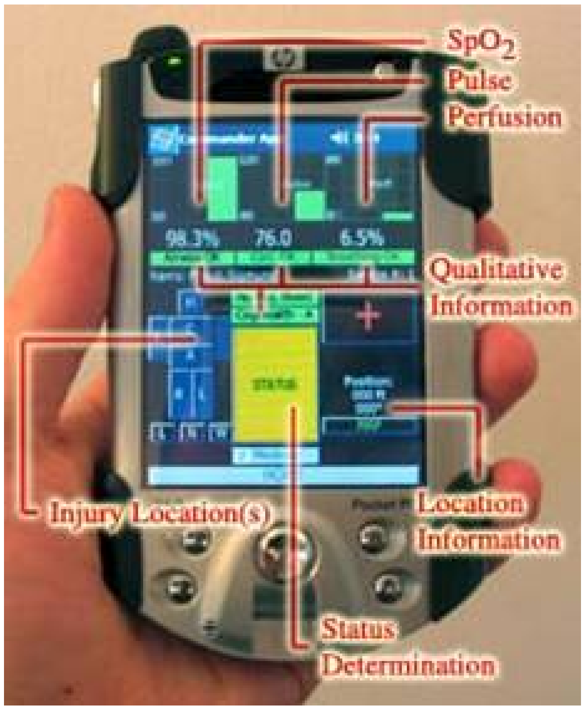


Figure 7.10: A front-end application running on the mobile device carried by first responders.

incorporate other types of online sources, such as calendar information. The query interface over pull channels in the current version meets this need. Another drawback was a lack of a built-in timeout facility, preventing an operator from gaining control until it received an event: some fusion logic needs to react to a *lack* of events as much as it reacts to new events.

We learned several lessons from our experience building various Solar applications. First, a stable, flexible, and efficient context-fusion platform (such as Solar) is a great help to application developers. It significantly reduced the developer's programming efforts. Solar's ability to reuse both classes and instances makes it easier to extend applications. Second, we found that a single type of sensor was unable to provide the desired accuracy, since every sensor has its limitations. It helped, although not always, to combine multiple sensor types, since some sensors are complementary. This observation again justifies the importance of context fusion. Finally we need to point out that easy and effective fault-tracking is necessary in the design of any "invisible" pervasive-computing systems. For instance, if the phone rings during a meeting, is it because the detector made a wrong decision, or is there something wrong with the hardware? Which sensor went wrong, the motion badge or pressure mat? How can you tell if they are embedded? Is there a hardware failure, or do they just need a new battery? We had some system failure cases in our experiments and they required significant effort to debug.

Chapter 8

Conclusions and Future Work

Pervasive computing is a relatively young research field with many challenging issues to be solved. This dissertation addresses the system infrastructure needed to support adaptive context-aware applications. We abstracted the application requirements into a data-fusion problem, and we built a flexible and scalable middleware infrastructure to allow applications to specify their data and computation needs.

8.1 Contributions

In this dissertation, we presented Solar, an infrastructure that connects sensors and applications while allowing applications to specify customized operators to tailor the received contextual information. Solar provides a suite of services to manage these operators and to meet the challenges of a heterogeneous and dynamic pervasive-computing environment. The primary contributions of this dissertation are:

- a flexible and scalable context-fusion system for a pervasive computing environment with a programming model based on an operator graph compositional model. This model allows applications to deploy specific data-fusion functionality inside the network;

- a policy-driven data-reduction technique that allows loss-tolerant context-aware applications to trade completeness for fast delivery in case of buffer overflows caused by rapid data streams and slow receivers;

- a scalable naming service that supports persistent queries and context-sensitive resource discovery, which takes advantage of our context-fusion facility and allows applications to offload most traffic and computation into the infrastructure;

- a set of component monitoring and dependency tracking protocols that automatically recover operators that are lost due to host failures and that automatically adjust the operator graphs according to changes in resource availability;

- a detailed analysis of several traces collected from location tracking systems that are in daily usage, which provides insights on typical pervasive-computing characteristics; and

- experience building pervasive-computing applications and lessons learned. Together with an open-source software package, Solar provides valuable contributions to the community.

8.2 Limitations and future work

Although Solar's operator-composition language is easy to learn and use, some users may prefer a descriptive language such as iQL [44]. With some effort, we believe that Solar can support both interfaces, for instance, by automatically parsing an iQL program into an operator graph that is deployable on Solar.

Solar does not allow loops in an operator graph, since most applications we built do not need this feature. We expect some applications may need loops in their data-flow, which will require Solar to be extended in future. Currently there are no tools to validate a specified operator graph, so errors such as connecting different pull/push ports cannot be detected before deployment. We also lack a GUI-based tool to help users specify an operator graph visually.

Solar's operator library is quite limited, except for some common filters and aggregators that work with our deployed sensors. We are currently expanding our operator library to support advanced fusion algorithms that follow some common models, somewhat similar to PQS's pluggable processes [13]. We believe that this will be an important step to increase both code-based and instance-based operator reuse.

Solar is designed to work in a LAN environment where Planets run inside a single organization, packet drops and host failures are not frequent, and latency is relatively low. It is natural to extend Solar to run in WAN settings, and we plan to develop Internet-based Solar applications and evaluate Solar's performance on a wide-area testbed called PlanetLab.[1]

Solar currently does not have explicit support for context history. Some operators may make decisions based on historical observations and thus may require a repository for history data. We plan to develop a persistence service that exposes a set of integrated APIs to allow time-based data storage and retrieval.

At this point, Solar is not secure and is vulnerable to malicious attacks, and there is no access control enforced to guard information privacy. We plan to integrate an automated access control propagation facility [93] and provide some degree of security by encrypting the data communications or leveraging existing solutions such as PKI (Public-Key Infrastructure). It is generally not easy to add security mechanisms into an already implemented system. We expect, however, Solar's service-oriented design may facilitate this process.

8.3 Conclusion

One barrier preventing pervasive-computing from faster adoption is the lack of a flexible and scalable infrastructure designed to meet the challenges of a heterogeneous and volatile environment. To support context-aware applications, our Solar system employs a flexible operator-composition programming model that facilitates both development and deployment of these applications. To manage application-specified operators on a set of overlay nodes called Planets, Solar provides several unique services such as application-level multicast with policy-driven data reduction to handle

[1]http://www.planetlab.org/

buffer overflow, context-sensitive resource discovery to handle environment dynamics, and proactive monitoring and recovery to handle common failures. Experimental results show that these services perform well on a typical DHT-based peer-to-peer routing substrate. Our experience from building applications with Solar shows that developers benefit from our infrastructure, allowing them to focus on task-specific functionalities without being overwhelmed by the complexity of a pervasive-computing environment.

Bibliography

[1] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *International Conference on Very Large Data Bases*, 12(2):120–139, August 2003.

[2] 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.

[3] 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 System Principles*, pages 186–201, Charleston, South Carolina, United States, 1999. ACM Press.

[4] Larry Arnstein, Chia-Yang Hung, Robert Franza, Qing Hong Zhou, Gaetano Borriello, Sunny Consolvo, and Jing Su. Labscape: A Smart Environment for the Cell Biology Laboratory. *IEEE Pervasive Computing*, 1(3):13–21, July-September 2002.

[5] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice Hall, 1986.

[6] P. Bahl and V.N. Padmanabhan. RADAR: an in-building RF-based user location and tracking system. In *Proceedings of the 19th Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 775–784, Tel Aviv, Israel, March 2000. IEEE Computer Society Press.

[7] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking up data in P2P systems. *Communications of the ACM*, 46(2):43–48, February 2003.

[8] Magdalena Balazinska, Hari Balakrishnan, and David Karger. INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery. In *Proceedings of the First International Conference on Pervasive Computing*, pages 195–210, Zurich, Switzerland, August 2002. Springer-Verlag.

[9] Magdalena Balazinska and Paul Castro. Characterizing Mobility and Network Usage in a Corporate Wireless Local-Area Network. In *Proceedings of the First International Conference on Mobile Systems, Applications, and Services*, pages 303–316, San Francisco, CA, May 2003. USENIX Association.

[10] Guruduth Banavar, Marc Kaplan, Kelly Shaw, Robert E. Strom, Daniel C. Sturman, and Wei Tao. Information flow based event distribution middleware. In *the Middleware Workshop at the ICDCS 1999*, Austin, Texas, 1999.

[11] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 2003.

[12] Francesco Bellotti, Riccardo Berta, Alessandro De Gloria, and Massimiliano Margarone. User Testing a Hypermedia Tour Guide. *IEEE Pervasive Computing*, 1(2):33–41, April-June 2002.

[13] Vincent Berk, Wayne Chung, Valentino Crespi, George Cybenko, Robert Gray, Diego Hernando, Guofei Jiang, Han Li, and Yong Sheng. Process query systems for surveillance and awareness. In *Proceedings of the 7th World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, FL, July 2003.

[14] Samrat Bhattacharjee, Kenneth L. Calvert, and Ellen W. Zegura. An architecture for active networking. In *Proceedings of High Performance Networking (HPN'97)*, White Plains, NY, April 1997.

[15] United States Air Force Scientific Advisory Board. Report on Building the Joint Battlespace Infosphere, Volume 2: Interactive Information Technologies, December 1999.

[16] United States Air Force Scientific Advisory Board. Report on Building the Joint Battlespace Infosphere, Volume 1: Summary, December 2000.

[17] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards Sensor Database Systems. In *Proceedings of the Second International Conference on Mobile Data Management*, pages 3–14, Hong Kong, China, January 2001. Springer-Verlag.

[18] Barry Brumitt, Brian Meyers, John Krumm, Amanda Kern, and Steven Shafer. EasyLiving: Technologies for Intelligent Environments. In *Proceedings of the Second International Symposium on Handheld and Ubiquitous Computing*, pages 12–29, Bristol, UK, September 2000. Springer-Verlag.

[19] Jenna Burrell, Geri K. Gay, Kiyo Kubo, and Nick Farina. Context-Aware Computing: A Test Case. In *Proceedings of the Fourth International Conference on Ubiquitous Computing*, pages 1–15, Goteborg, Sweden, September-October 2002. Springer-Verlag.

[20] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. Mobile-Agent Coordination Models for Internet Applications. *IEEE Computer*, 33(2):82–89, February 2000.

[21] George Candea and Armando Fox. Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, pages 125–130, Elmau, Germany, May 2001. IEEE Computer Society Press.

[22] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving scalability and expressiveness in an Internet-scale event notification service. In *Proceedings of the 19th Annual Symposium on Principles of Distributed Computing*, pages 219–227, Portland, Oregon, United States, 2000. ACM Press.

[23] 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, 2001.

[24] Fabio Casati, Ski Ilnicki, Li-Jie Jin, Vasudev Krishnamoorthy, and Ming-Chien Shan. Adaptive and Dynamic Service Composition in eFlow. Technical Report HPL-2000-39, HP Labs, 2000.

[25] Fabio Casati, Ski Ilnicki, Li-Jie Jin, Vasudev Krishnamoorthy, and Ming-Chien Shan. eFlow: a Platform for Developing and Managing Composite e-Services. Technical Report HPL-2000-36, HP Labs, 2000.

[26] M. Castro, P. Druschel, A.-M. Kermarrec, and A.I.T. Rowstron. Scribe: a large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8):1489–1499, October 2002.

[27] M. Castro, M.B. Jones, A.-M. Kermarrec, A. Rowstron, M. Theimer, H. Wang, and A. Wolman. An evaluation of scalable application-level multicast built using peer-to-peer overlays. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 1510–1520, San Francisco, CA, April 2003. IEEE Computer Society Press.

[28] Miguel Castro, Peter Druschel, Y. Charlie Hu, and Antony Rowstron. Topology-aware routing in structured peer-to-peer overlay networks. Technical Report MSR-TR-2002-82, Microsoft Research, 2002.

[29] Paul Castro, Patrick Chiu, Ted Kremenek, and Richard Muntz. A Probabilistic Room Location Service for Wireless Networked Environments. In *Proceedings of the Third International Conference on Ubiquitous Computing*, pages 18–34, Atlanta, GA, September-October 2001. Springer-Verlag.

[30] Paul Castro, Benjamin Greenstein, Richard Muntz, Parviz Kermani, Chatschik Bisdikian, and Maria Papadopouli. Locating application data across service discovery domains. In *Proceedings of the Seventh Annual International Conference on Mobile Computing and Networking*, pages 28–42, Rome, Italy, 2001. ACM Press.

[31] K. Mani Chandy, Brian Emre Aydemir, Elliott Michael Karpilovsky, and Daniel M. Zimmerman. Event-driven architectures for distributed crisis management. In *Proceedings of the 15th IASTED International Conference on Parallel and Distributed Computing and Systems*, November 2003.

[32] Alvin Chen, Richard R. Muntz, Spencer Yuen, Ivo Locher, Sung I. Park, and Mani B. Srivastava. A Support Infrastructure for the Smart Kindergarten. *IEEE Pervasive Computing*, 1(2):49–57, April-June 2002.

[33] 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.

[34] Guanling Chen and David Kotz. Solar: Towards a flexible and scalable data-fusion infrastructure for ubiquitous computing. In *Proceedings of the Workshop on Application Models and Programming Tools for Ubiquitous Computing at Third International Conference on Ubiquitous Computing (UbiComp 2001)*, October 2001.

[35] Guanling Chen and David Kotz. Context Aggregation and Dissemination in Ubiquitous Computing Systems. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, pages 105–114, Callicoon, New York, June 2002. IEEE Computer Society Press.

[36] Guanling Chen and David Kotz. Context-Sensitive Resource Discovery. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, pages 243–252, Fort Worth,Texas, March 2003. IEEE Computer Society Press.

[37] Guanling Chen and David Kotz. Application-controlled loss-tolerant data dissemination. Technical Report TR2004-488, Dept. of Computer Science, Dartmouth College, February 2004.

[38] Guanling Chen and David Kotz. A case study of four location traces. Technical Report TR2004-490, Dept. of Computer Science, Dartmouth College, Februray 2004.

[39] Guanling Chen and David Kotz. Dependency management in distributed settings (poster abstract). In *Proceedings of the First International Conference on Autonomic Computing (ICAC 2004)*, pages 272–273, New York City, NY, May 2004. IEEE Computer Society Press.

[40] Guanling Chen, Ming Li, and David Kotz. Design and implementation of a large-scale context fusion network. In *Proceedings of the First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous 2004)*, Boston, MA, August 2004.

[41] Yan Chen, Randy H. Katz, and John D. Kubiatowicz. SCAN: A Dynamic, Scalable, and Efficient Content Distribution Network. In *Proceedings of the First International Conference on Pervasive Computing*, pages 282–296, Zurich, Switzerland, August 2002. Springer-Verlag.

[42] 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, Massachusetts, United States, 2000. ACM Press.

[43] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the 1990 ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 200–208, Philadelphia, Pennsylvania, United States, 1990. ACM Press.

[44] Norman H. Cohen, Hui Lei, Paul Castro, John S. Davis II, and Apratim Purakayastha. Composing Pervasive Data Using iQL. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, pages 94–104, Callicoon, New York, June 2002. IEEE Computer Society Press.

[45] Norman H. Cohen, Apratim Purakayastha, Luke Wong, and Danny L. Yeh. iQueue: A Pervasive Data Composition Framework. In *Proceedings of the Third International Conference on Mobile Data Management*, pages 146–153, Singapore, January 2002. IEEE Computer Society Press.

[46] Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz. An architecture for a secure service discovery service. In *Proceedings of the Fifth Annual International Conference on Mobile Computing and Networking*, pages 24–35, Seattle, Washington, United States, 1999. ACM Press.

[47] Stephen E. Deering and David R. Cheriton. Multicast routing in datagram internetworks and extended LANs. *ACM Transactions on Computer Systems*, 8(2):85–110, May 1990.

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

[49] Anind K. Dey and Gregory D. Abowd. CybreMinder: A Context-Aware System for Supporting Reminders. In *Proceedings of the Second International Symposium on Handheld and Ubiquitous Computing*, pages 172–186, Bristol, UK, September 2000. Springer-Verlag.

[50] Anind K. Dey, Gregory D. Abowd, and Daniel Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction (HCI) Journal*, 16(2-4), 2001.

[51] Anind K. Dey, Daniel Salber, Masayasu Futakawa, and Gregory D. Abowd. An architecture to support context-aware applications. Technical Report GIT-GVU-99-23, Georgia Institute of Technology, College of Computing, June 1999.

[52] Ludger Fiege, Felix C. Gärtner, Oliver Kasten, and Andreas Zeidler. Supporting Mobility in Content-Based Publish/Subscribe Middleware. In *Proceedings of the 2003 International Middleware Conference*, pages 103–122, Rio de Janeiro, Brazil, June 2003. Springer-Verlag.

[53] Margaret Fleck, Marcos Frid, Tim Kindberg, Eamonn O'Brien-Strain, Rakhi Rajani, and Mirjana Spasojevic. From Informing to Remembering: Ubiquitous Systems in Interactive Museums. *IEEE Pervasive Computing*, 1(2):13–21, April-June 2002.

[54] Xiaodong Fu, Weisong Shi, Anatoly Akkerman, and Vijay Karamcheti. CANS: Composable, Adaptive Network Services Infrastructure. In *Proceedings of the 3rd USENIX Symposium of Internet Technologies and Systems*, San Francisco, CA, March 2001. USENIX Association.

[55] David Garlan and Mary Shaw. An introduction to software architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University, January 1994.

[56] Hans-W. Gellersen, Michael Beigl, and Holger Krull. The MediaCup: Awareness Technology Embedded in an Everyday Object. In *Proceedings of the First International Symposium on Handheld and Ubiquitous Computing*, pages 308–310, Karlsruhe, Germany, September 1999. Springer-Verlag.

[57] Max K. Goff. *Network Distributed Computing: Fitscapes and Fallacies*. Prentice Hall, 2003.

[58] William G. Griswold, Robert Boyer, Stephen W. Brown, Tan Minh Truong, Ezekiel Bhasker, Gregory R. Jay, and R. Benjamin Shapiro. Activecampus— sustaining educational communities through mobile technology. Technical Report CS2002-0714, University of California, San Diego, July 2002.

[59] Erik Guttman. Service Location Protocol: Automatic Discovery of IP Network Services. *IEEE Internet Computing*, 3(4):71–80, July/August 1999.

[60] Richard H.R. Harper. Why do people wear active badges? Technical Report EPC-1993-120, Rank Xerox Research Center, 1993.

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

[62] Nicholas J. A. Harvey and Alec Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *Proceedings of the 4th USENIX Symposium of Internet Technologies and Systems*, Seattle, WA, March 2003. USENIX Association.

[63] Mike Hazas and Andy Ward. A Novel Broadband Ultrasonic Location System. In *Proceedings of the Fourth International Conference on Ubiquitous Computing*, pages 264–280, Goteborg, Sweden, September-October 2002. Springer-Verlag.

[64] John Heidemann, Fabio Silva, Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, and Deepak Ganesan. Building efficient wireless sensor networks with low-level naming. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, pages 146–159, Banff, Alberta, Canada, 2001. ACM Press.

[65] Jeffrey Hightower and Gaetano Borriello. Location Systems for Ubiquitous Computing. *IEEE Computer*, 34(8):57–66, August 2001.

[66] Jeffrey Hightower, Barry Brumitt, and Gaetano Borriello. The Location Stack: A Layered Model for Location in Ubiquitous Computing. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, pages 22–28, Callicoon, New York, June 2002. IEEE Computer Society Press.

[67] Fritz Hohl, Uwe Kubach, Alexander Leonhardi, Kart Rothermel, and Markus Schwehm. Next century challenges: Nexus – an open global infrastructure for spatial-aware applications. In *Proceedings of the Fifth Annual International Conference on Mobile Computing and Networking*, pages 249–255, Seattle, Washington, United States, 1999. ACM Press.

[68] Jason I. Hong and James A. Landay. An infrastructure approach to context-aware computing. *Human-Computer Interaction (HCI) Journal*, 16(2-4), 2001.

[69] Yang hua Chu, Sanjay G. Rao, and Hui Zhang. A case for end system multicast (keynote address). In *Proceedings of the 2000 ACM Conference on Measurement and Modeling of Computer Systems*, pages 1–12, Santa Clara, California, United States, 2000. ACM Press.

[70] Stephen S. Intille. Designing a Home of the Future. *IEEE Pervasive Computing*, 1(2):76–82, April-June 2002.

[71] V. Jacobson. Congestion avoidance and control. In *Symposium Proceedings on Communications Architectures and Protocols*, pages 314–329, Stanford, California, United States, 1988. ACM Press.

[72] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, James W. O'Toole, and Jr. Overcast: Reliable Multicasting with an Overlay Network. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000. USENIX Association.

[73] Changhao Jiang and Peter Steenkiste. A Hybrid Location Model with a Computable Location Identifier for Ubiquitous Computing . In *Proceedings of the Fourth International Conference on Ubiquitous Computing*, pages 246–263, Goteborg, Sweden, September-October 2002. Springer-Verlag.

[74] Brad Johanson and Armando Fox. The Event Heap: A Coordination Infrastructure for Interactive Workspaces. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, pages 83–93, Callicoon, New York, June 2002. IEEE Computer Society Press.

[75] Brad Johanson, Armando Fox, and Terry Winograd. The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms. *IEEE Pervasive Computing*, 1(2):67–74, April-June 2002.

[76] R. José and N. Davies. Scalable and Flexible Location-Based Services for Ubiquitous Information Access. In *Proceedings of the First International Symposium on Handheld and Ubiquitous Computing*, pages 52–66, Karlsruhe, Germany, September 1999. Springer-Verlag.

[77] Emre Kcman and Armando Fox. Using Dynamic Mediation to Integrate COTS Entities in a Ubiquitous Computing Environment. In *Proceedings of the Second International Symposium on Handheld and Ubiquitous Computing*, pages 211–226, Bristol, UK, September 2000. Springer-Verlag.

[78] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[79] Andrew M. Ladd, Kostas E. Bekris, Algis Rudys, Lydia E. Kavraki, Dan S. Wallach, and Guillaume Marceau. Robotics-based location sensing using wireless ethernet. In *Proceedings of the Eighth Annual International Conference on Mobile Computing and Networking*, pages 227–238, Atlanta, Georgia, USA, 2002. ACM Press.

[80] Ulf Leonhardt and Jeff Magee. Multi-sensor location tracking. In *Proceedings of the Fourth Annual International Conference on Mobile Computing and Networking*, pages 203–214, Dallas, Texas, United States, 1998. ACM Press.

[81] Anthony Levas, Claudio Pinhanez, Gopal Pingali, Rick Kjeldsen, Mark Podlaseck, and Noi Sukaviriya. An Architecture and Framework for Steerable Interface Systems. In *Proceedings*

of the Fifth International Conference on Ubiquitous Computing, pages 333–348, Seattle,WA, October 2003. Springer-Verlag.

[82] Frank Leymann. Web Services Flow Language (WSFL 1.0).

[83] Henry Lieberman and Ted Selker. Out of context: Computer systems that adapt to, and learn from, context. *IBM Systems Journal*, 39(3-4):617–632, 2000.

[84] Henning Maaß. Location-aware mobile applications based on directory services. In *Proceedings of the Third Annual International Conference on Mobile Computing and Networking*, pages 23–33, Budapest, Hungary, 1997. ACM Press.

[85] Henning Maaß. Location-aware mobile applications based on directory services. *Mobile Networks and Applications*, 3(2):157–173, August 1998.

[86] Samuel Madden, Michael J. Franklin, and Joseph M. Hellerstein. TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 131–146, Boston, MA, December 2002. USENIX Association.

[87] Christopher P. Masone. Role Definition Language (RDL): A language to describe context-aware roles. Technical Report TR2002-426, Dartmouth College, May 2002.

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

[89] Steven McCanne, Van Jacobson, and Martin Vetterli. Receiver-driven layered multicast. In *Proceedings of the 1996 ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 117–130, Palo Alto, California, United States, 1996. ACM Press.

[90] Joseph F. McCarthy and Eric S. Meidel. ActiveMap: A Visualization Tool for Location Awareness to Support Informal Interactions. In *Proceedings of the First International Symposium on Handheld and Ubiquitous Computing*, pages 158–170, Karlsruhe, Germany, September 1999. Springer-Verlag.

[91] S. Mehrotra, C. Butts, D. Kalashnikov, N. Venkatasubramanian, R. Rao, G. Chockalingam, R. Eguchi, B. Adams, and C. Huyck. Project Rescue: Challenges in responding to the unexpected. In *Proceedings of 16th Annual Symposium on Electronic Imaging Science and Technology*, San Jose, CA, January 2004.

[92] Sun Microsystems. Jini architecture specification, December 2001.

[93] Kazuhiro Minami. Controlling access to pervasive information in the Solar system. Technical Report TR2002-422, Dartmouth College, February 2002.

[94] 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.

[95] M.J. Monteiro, J. Pereira, and L. Rodrigues. Integration of flight simulator 2002 with an epidemic multicast protocol. In *Proceedings of International the Workshop on Large-Scale Group Communication*, October 2003.

[96] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, resource management, and approximation in a data stream management system. In *Proceedings of the First Biennial Conference on Innovative Database Systems*, January 2003.

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

[98] M. Nidd. Service discovery in DEAPspace. *IEEE Personal Communications*, 8(4):39–45, 2001.

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

[100] Joseph C. Pasquale, George C. Polyzos, Eric W. Anderson, and Vachaspathi P. Kompella. Filter propagation in dissemination trees: Trading off bandwidth and processing in continuous media networks. In *Proceedings of the 4th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 93)*, pages 259–268, Lancaster, UK, November 1993. Springer-Verlag.

[101] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, and Noah Treuhaft. Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Technical Report CSD-02-1175, U.C. Berkeley, 2002.

[102] Peter R. Pietzuch, Brian Shand, and Jean Bacon. A Framework for Event Composition in Distributed Systems. In *Proceedings of the 2003 International Middleware Conference*, pages 62–82, Rio de Janeiro, Brazil, June 2003. Springer-Verlag.

[103] Gopal Pingali, Claudio Pinhanez, Anthony Levas, Rick Kjeldsen, Mark Podlaseck, Han Chen, and Noi Sukaviriya. Steerable Interfaces for Pervasive Computing Spaces. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, pages 315–322, Fort Worth,Texas, March 2003. IEEE Computer Society Press.

[104] Sridhar Pingali, Don Towsley, and James F. Kurose. A comparison of sender-initiated and receiver-initiated reliable multicast protocols. In *Proceedings of the 1994 ACM Conference on Measurement and Modeling of Computer Systems*, pages 221–230, Nashville, Tennessee, United States, 1994. ACM Press.

[105] Shankar R. Ponnekanti and Armando Fox. Sword: A developer toolkit for web service composition. In *Proceedings of the Eleventh International World Wide Web Conference*, Honolulu, Hawaii, May 2002.

[106] 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, Massachusetts, United States, 2000. ACM Press.

[107] B. Raman and R.H. Katz. Load balancing and stability issues in algorithms for service composition. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 1477–1487, San Francisco, CA, April 2003. IEEE Computer Society Press.

[108] Bhaskaran Raman and Randy H. Katz. An architecture for highly available wide-area service composition. *Computer Communication Journal*, 26(15):1727–1740, September 2003.

[109] Suchitra Raman and Steven McCanne. A model, analysis, and protocol framework for soft state-based communication. In *Proceedings of the 1999 ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 15–25, Cambridge, Massachusetts, United States, 1999. ACM Press.

[110] Cliff Randell and Henk Muller. Low Cost Indoor Positioning System. In *Proceedings of the Third International Conference on Ubiquitous Computing*, pages 42–48, Atlanta, GA, September-October 2001. Springer-Verlag.

[111] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *Proceedings of the 2001 ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 161–172, San Diego, California, United States, 2001. ACM Press.

[112] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker. Application-level multicast using content-addressable networks. In *Third International Workshop on Networked Group Communication (NGC 2001)*, pages 14–29, London, UK, November 2001. Springer-Verlag.

[113] Bradley J. Rhodes. The wearable remembrance agent: a system for augmented memory. In *Proceedings of The First International Symposium on Wearable Computers (ISWC '97)*, pages 123–128, Cambridge, MA, 1997. IEEE Computer Society Press.

[114] T. Richardson, F. Bennett, G. Mapp, and A. Hopper. A ubiquitous, personalized computing environment for all: Teleporting in an X Window System Environment. *IEEE Personal Communications*, 1(3):6, 1994.

[115] Antony Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the 2001 International Middleware Conference*, pages 329–350, Heidelberg, Germany, November 2001. Springer-Verlag.

[116] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. Scribe: The design of a large-scale event notification infrastructure. In *Third International Workshop on Networked Group Communication (NGC 2001)*, pages 30–43, London, UK, November 2001. Springer-Verlag.

[117] M. Satyanarayanan. Pervasive computing: vision and challenges. *IEEE Personal Communications*, 8(4):10–17, 2001.

[118] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *Proceedings of the First IEEE Workshop on Mobile Computing Systems and Applications*, pages 85–90, Santa Cruz, CA, December 1994. IEEE Computer Society Press.

[119] Bill Schilit and Marvin Theimer. Disseminating active map information to mobile hosts. *IEEE Network*, 8(5):22–32, 1994.

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

[121] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[122] Chia Shen, Katherine Everitt, and Kathleen Ryall. UbiTable: Impromptu Face-to-Face Collaboration on Horizontal Interactive Surfaces. In *Proceedings of the Fifth International Conference on Ubiquitous Computing*, pages 281–288, Seattle,WA, October 2003. Springer-Verlag.

[123] Frank Siegemund and Christian Flörkemeier. Interaction in Pervasive Computing Settings Using Bluetooth-Enabled Active Tags and Passive RFID Technology Together with Mobile Phones. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, pages 378–387, Fort Worth,Texas, March 2003. IEEE Computer Society Press.

[124] Vince Stanford. Using pervasive computing to deliver elder care. *IEEE Pervasive Computing*, 1(1):10–13, January-March 2002.

[125] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, February 2003.

[126] Nesime Tatbul, Uğur Çetintemel, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. Load Shedding in a Data Stream Manager. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 309–320, Berlin, Germany, September 2003. Morgan Kaufmann.

[127] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survey of active network research. *IEEE Communications*, 35(1):80–86, January 1997.

[128] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the 2002 International Conference on Compiler Construction*, pages 179–196, Grenoble, France, April 2002. Springer-Verlag.

[129] Amin Vahdat and Amit Aggarwal. Active Names: Flexible Location and Transport of Wide-Area Resources. In *Proceedings of the 2nd USENIX Symposium of Internet Technologies and Systems*, Boulder, CO, October 1999. USENIX Association.

[130] Jim Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82, July 1999.

[131] Jue Wang. Performance evaluation of the resource discovery service in Solar. Technical Report TR2004-513, Dartmouth College, August 2004.

[132] Jue Wang, Guanling Chen, and David Kotz. A sensor-fusion approach for meeting detection. In *Proceedings of the Workshop on Context Awareness at the Second International Conference on Mobile Systems, Applications, and Services (MobiSys 2004)*, Boston, MA, June 2004.

[133] Roy Want and Andy Hopper. Active badges and personal interactive computing objects. *IEEE Transactions on Consumer Electronics*, 8(1):10–20, February 1992.

[134] Roy Want, Trevor Pering, Gunner Danneels, Muthu Kumar, Murali Sundar, and John Light. The Personal Server: Changing the Way We Think about Ubiquitous Computing. In *Proceedings of the Fourth International Conference on Ubiquitous Computing*, pages 194–209, Goteborg, Sweden, September-October 2002. Springer-Verlag.

[135] Roy Want, Bill N. Schilit, Norman I. Adams, Rich Gold, Karin Petersen, David Goldberg, John R. Ellis, and Mark Weiser. The ParcTab ubiquitous computing experiment. In Tomasz Imielinski and Henry F. Korth, editors, *Mobile Computing*, pages 45–101. Kluwer Academic Publishers, 1996.

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

[137] Mark Weiser. The computer for the 21st century. *Scientific American*, 265(3):66–75, January 1991.

[138] Mark Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):75–84, July 1993.

[139] Matt Welsh, David Culler, and Eric Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, pages 230–243, Banff, Alberta, Canada, 2001. ACM Press.

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

[141] A. Abram White. Performance and interoperability in Solar. Technical Report TR2002-427, Dartmouth College, May 2002.

[142] A. Abram White. XSLT and XQuery as operator languages. Technical Report TR2002-429, Dartmouth College, May 2002.

[143] B.Y. Zhao, L. Huang, J. Stribling, S.C. Rhea, A.D. Joseph, and J.D. Kubiatowicz. Tapestry: A Resilient Global-Scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.

[144] Shelley Zhuang, Kevin Lai, Ion Stoica, Randy Katz, and Scott Shenker. Host Mobility Using an Internet Indirection Infrastructure. In *Proceedings of the First International Conference on Mobile Systems, Applications, and Services*, pages 129–144, San Francisco, CA, May 2003. USENIX Association.