

Data-Centric Middleware for Context-Aware Pervasive Computing

Guanling Chen[†], Ming Li[‡], and David Kotz[‡]

[†]Department of Computer Science, University of Massachusetts Lowell
Lowell, MA 01854
glchen@cs.uml.edu

[‡]Institute for Security Technology Studies (ISTS), Dartmouth College
Department of Computer Science, Dartmouth College
Hanover, NH 03755
{mingli, dfk}@cs.dartmouth.edu

Abstract

The complexity of developing and deploying context-aware pervasive-computing applications calls for distributed software infrastructures that assist applications to collect, aggregate, and disseminate contextual data. In this paper, we motivate a data-centric design for such an infrastructure to support context-aware applications. Our middleware system, Solar, treats contextual data sources as stream publishers. The core of Solar is a scalable and self-organizing peer-to-peer overlay to support data-driven services. We describe how different services can be systematically integrated on top of the Solar overlay and evaluate the resource-discovery and data-dissemination services. We also discuss our experience and lessons learned when using Solar to support several implemented scenarios. We conclude that a data-centric infrastructure is necessary to facilitate both development and deployment of context-aware pervasive-computing applications.

Keywords: context-aware computing, pervasive computing, data-centric middleware, smart environments, emergency response.

1 Introduction

The vision of pervasive computing consists of unobtrusively integrating computers with people’s everyday lives at home and work. The technology functions in the background, assisting users’ current tasks with minimal distraction for tedious routines. This vision, sometimes called ubiquitous computing [66] or invisible computing [51], has inspired many researchers to work on new hardware, networking protocols, human-computer interactions, security and privacy, applications, and social implications [67, 59].

Pervasive-computing applications have to gracefully integrate with human users. The non-traditional computing platform, with mobile and embedded devices and applications, may easily overwhelm a user if not carefully managed. To minimize user distraction the applications must be aware of and adapt to the situation in which they are running, such as the state of the physical space, the users, and the computational resources. When informed about such information, applications are able to modify their behaviors reactively or proactively to assist user tasks. We loosely define “context” to be the set of environmental states and interactions that either determines an application’s behavior or in which an application event occurs and is interesting to the user.

Unlike explicit user input, context represents implicit input to the application. An application typically infers context information from various sensors to bridge the gap between physical and virtual worlds. It is, however, not trivial for the applications to discover, obtain, and monitor the relevant sensor data by themselves. Researchers have responded by abstracting the common functionalities into an independent software middleware to facilitate development of context-aware applications. Schilit and his colleagues at Xerox Parc developed Active Map, which provides a service managing devices' and users' current location [60]. Dey et al. developed a Context Toolkit that further abstracts any type of context by providing a widget analogy [28]. Hong and Landy developed a Context Fabric that focused on compatibility of sensor data, with which the infrastructure could automatically create a path among existing services to derive context [35]. Gu et al. further added an ontology component into the context-managing infrastructure to enable context reasoning [33].

While these middleware architectures have provided data abstractions and communication layers to facilitate application prototyping, thus far there has been no systematic investigation of desired properties for a context-aware middleware so it can move from the lab to a deployment in the real world. In particular, we believe that a data-centric design is critical for context-aware computing in a smart environment. By *data-centric*, we mean that the contextual data is driving both application behavior and service adaptation inside the middleware system. To be more specific, sensors are treated as data-stream publishers and application developers explicitly *compose* desired sensor streams and transform low-level data to more meaningful context using *operators* (Section 3). As the contextual data flows through the infrastructure, the *values* of data items could trigger adaptations of various middleware services according to predefined rules and policies (Section 5 and 6). Note that sometimes the term “data-centric” is used to refer design decisions on how to deliver and store data, such as QoS adaptation and persistent storage in data-centric networking [29]. Our intention, however, is for the infrastructure itself to adapt to the data values, similar to the philosophy of content-based routing [30], but not constrained to routing behaviors.

We now describe two application scenarios as motivations to our work on a context-aware middleware system.

Smart-office scenario. Alice often holds meetings with her colleagues in her office. Frequently her desk phone rings during the middle of meetings, which distracts her and others in the meeting. Although she decides not to answer any call while she is in a meeting, the phone still rings for a while, interrupting the conversation, before the call is transferred to voice mail. If Alice had a SmartPhone application that could use embedded sensors to detect whether there is an ongoing meeting, the incoming calls could be routed to voice mail without causing any interruptions. The SmartPhone application should determine, without human intervention, when to turn on and off the call forwarding (to voice mail) functionality by using one or more sensors to accurately detect the beginning and ending of meetings.

Disaster-response scenario. First responders carry out search and rescue missions after man-made or natural disasters. During disaster response it is important to provide all responders with an awareness of the evolving situation. Responders could deploy environmental sensors, attach physiological sensors to victims and responders, and set up localization systems. With all these inputs whose values are constantly changing, the command and control applications could use a context-aware middleware to provide real-time situational awareness for responders and decision makers at all levels of the incident response hierarchy. For instance, the application could alert the local commander to pull out

a responder (Bob) who has been exposed to toxic smoke, based on the readings from Bob’s wearable medical sensors and smoke sensors deployed in the environment.

We have implemented a data-centric middleware system, Solar, to support context-aware pervasive-computing applications. Solar leverages an attribute-based data abstraction, on which semantics and further reasoning functionalities could be added. Solar provides a data-flow composition programming model, on which applications can specify modular computation for context inference. Solar leverages a context-sensitive resource discovery method for applications to cope with environment dynamics, and a policy-driven data-dissemination method for applications to deal with rapid data streams. Finally, Solar consists of a distributed architecture built on a peer-to-peer routing overlay, which is scalable and self-organized for easy management. With Solar, we have implemented both scenarios described above. Note that Solar is designed for applications to discover, collect, and aggregate contextual data from sensors. It is not designed for interactions with devices such as an ambient display. We expect those functions be implemented by additional service interfaces outside of Solar.

In this paper, we investigate two key aspects of our middleware design. First, the applications must have a programming model to allow dynamic composition without requiring the developer or user to identify specific sensors and devices. This design choice facilitates development of smart-environment applications. Second, the supporting system must have access to the values in the passing data to allow automated adaptation without explicit instructions from applications or human operators at runtime. This design choice facilitates deployment of pervasive-computing applications. Here we focus on architectural details and system issues to realize these two design choices and we evaluate the performance of the resource discovery and data dissemination in Solar. In this paper we make following four contributions:

- we motivate and describe new design principles of middleware support for context-aware pervasive-computing applications,
- we describe a novel middleware architecture based on a scalable and self-organizing peer-to-peer overlay,
- we present evaluation results on two key middleware services: resource discovery and data dissemination, and
- we discuss our experience and lessons learned when using Solar to implement pervasive-computing applications.

In the rest of this paper, we present Solar’s data abstraction, programming model, architecture, resource discovery, and data dissemination from Sections 2 to 6. We show how a data-centric design plays an important role in all these components. In Section 7, we study two applications implemented with Solar and discuss the limitations we encountered and some retrospective lessons learned. Finally we discuss related work in Section 8 and conclude in Section 9. In other publications, we have presented how context could be used to make authorization decisions and to build “virtual walls” for privacy protection using Solar [49, 40].

2 Data Abstraction

We treat sensors as *information sources*, which produce their data as *events*. The sequence of events produced is an *event stream*. An event *publisher* produces an event stream, and an event *subscriber* consumes an event stream. The sources are publishers, which choose when and what events to publish based on the nature of the sensors; their sub-

scribers have no control of publishers, except for manipulating received events. A subscriber may also be a publisher; for example, it is easy to write an *operator* that subscribes to a stream of interest and publishes some or all of the events after filtering and possibly transforming the events.

It is feasible to use any representation for the events as long as it is agreeable to both publisher and subscribers. There are four typical approaches used by event-distribution systems. The event may be a structure encoded in binary, a serialized object appropriate to a particular object-oriented language, a set of attribute-value pairs (usually represented as lines in an ASCII string), or a small XML document. Each representation has advantages and disadvantages [20].

While binary encoding is small and can be portable, object representation and XML encoding could be more flexible and better structured. However the parsing and encoding, for the latter two methods, could impose larger overhead. Solar takes a middle ground and all Solar components agree on an attribute-based event representation, which allows the middleware services to inspect the *values* in the passing events when necessary. By exposing the event content to middleware between sensors and applications, Solar can continuously adapt according to application-specified rules and policies, such as in resource discovery (Section 5) and context dissemination (Section 6), to meet the challenges of context-aware computing.

3 Programming Model

Pervasive-computing applications typically need high-level context rather than raw sensor data. Context may be derived by aggregating data from one or more sensors. These aggregation algorithms may not be trivial; substantial computational resources may be necessary to process incoming data streams. Solar aims to make it possible to offload this computation from the end-user application device into the middleware, running on one or more servers that host the Solar software. As a result, we can move processing closer to the data source, onto platforms that are more capable and more robust, and limit the amount of data that flows across low-bandwidth edge networks to the client.

It is almost impossible to anticipate how applications need to use what kind of data and how a piece of sensor data could be used by different applications. Instead, applications should instruct or “program” Solar on which sensor to use and how the sensor data should be aggregated into desired context. This approach deviates from a service-based approach, where the infrastructure contains dedicated services to provide predefined context through fixed APIs. The service-based approach limits flexibility and the applications may need to do significant post-processing work; in Solar, the application can meet its needs exactly by pushing its functionality into the Solar middleware by defining and deploying new operators.

In this section, we present a data-flow programming model that is a key design choice for our data-centric middleware. Applications compose data flows, rather than interacting directly with mobile and embedded sensors and devices. The applications have full control over data composition and may annotate flows with context-sensitive policies, allowing Solar to inspect data values and customize sensor selection and data dissemination behaviors.

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 reuse. We consider two kinds of reuse: with *code-based*

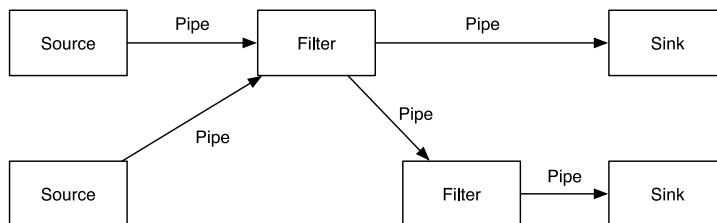


Figure 1: The filter-and-pipe software architecture style promotes reuse and composition.

reuse applications import existing modules from documented libraries, such as the one included in Java’s Development Kit; with *instance-based reuse* applications discover and use already deployed data-processing 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.

One popular software architectural pattern for data-stream oriented processing is *filter-and-pipe* [32], which supports reuse and composition naturally. In a filter-and-pipe style, as shown in Figure 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. A data flow starts from a source, through a sequence of pipes and filters, and reaches a sink.

Solar uses the filter-and-pipe pattern to support the data-flow based programming model. In our terminology, we call a filter an “operator” and a pipe a “channel”. A channel is directional and has two ends; at the input end is attached a *source*, and at the output end is attached 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. A channel connects an upstream operator to a downstream operator, and the direction of a channel indicates the direction of data flow. This simple model 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*.

Solar provides an XML-based language for operator composition. We illustrate the language using the example scenario described in Section 1. Suppose there is a dedicated meeting area in Alice’s office including a table and chairs. The smart-room application may determine the meeting status by first detecting whether there is more than one person sitting in those chairs. We attach a pressure sensor and a motion sensor to the chair. 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 passing user or when a heavy object is placed on the chair.

The operator graph used to detect human presence is straightforward (Figure 2(a)). The operator “presence fusion” takes two inputs and produces events indicating 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

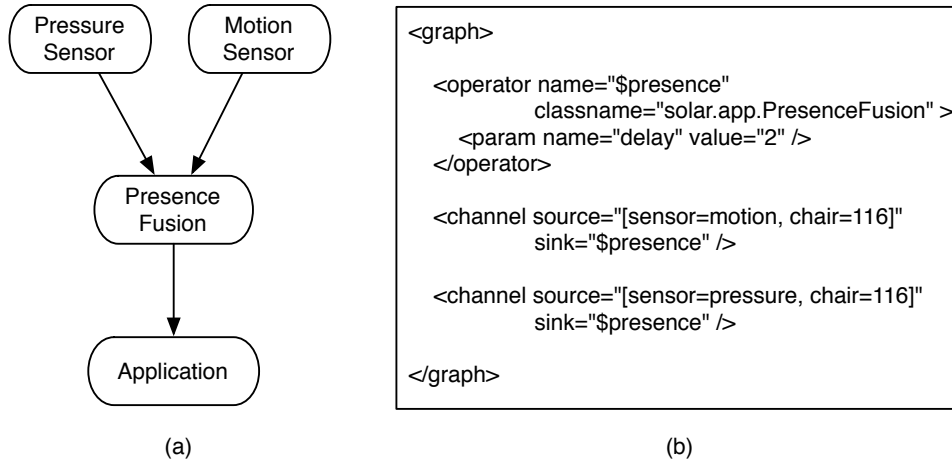


Figure 2: A simple operator graph and its XML encoding for detecting human presence in an instrumented chair.

observations. We present the details of a specific solution to detect ongoing meetings based on human presence in another paper [65].

We show how this operator graph is encoded with Solar’s XML language in Figure 2(b). We first define the fusion operator with a variable name `$presence`, which uses the specified Java class and is initialized with an algorithm-specific parameter. The channel element has source and sink labels. The source could be either a previously defined operator or a *name query* as shown in our example. 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 Section 5. Although not shown, the channel could also be associated with a flow-control policy that is discussed in Section 6.

4 Solar Architecture

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*. Planets provide some key services: operator hosting and execution, sensor/operator registration and discovery, and data dissemination through operator graphs. The more Planets are deployed, the more capacity Solar has to serve sensors and applications.

Since the Planets are functionally equivalent, Solar interconnects them using an application-level peer-to-peer (P2P) protocol. The advantage of using a P2P-based service overlay is its capability of automatic handling of Planet join and departure. The first generation of P2P protocols, such as used by Napster,¹ Gnutella,² and Kazaa,³ however, are mainly designed for file swapping over the Internet and not appropriate for Solar.

¹<http://www.napster.com>

²<http://www.gnutella.com>

³<http://www.kazaa.com>

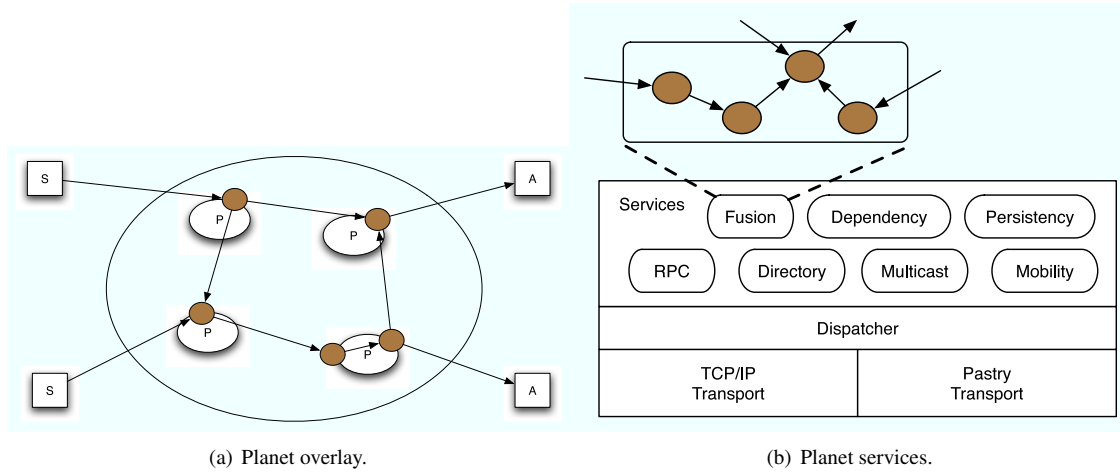


Figure 3: Solar consists a set of functionally equivalent Planets that form a service overlay.

Solar uses a Distributed Hashtable (DHT) [6], a new P2P concept used by Pastry [58], Chord [62], and Tapestry [68]. Each Planet is assigned a unique numeric ID, and the DHT interface allows components to send a message to a numeric key. The message will be delivered to a Planet with the numerically closest ID. This mechanism allows Solar services to focus on data objects instead of on where they live (IP addresses). Solar uses the Pastry library because it is implemented in Java and could be easily integrated with Solar, but it is possible to use other DHT protocols since they provide similar APIs.

It is known that DHT protocols are not efficient to handle high-rate “churn” (rapid node join and departure). This is a serious problem for applications deployed over the Internet on user PCs, which may join and leave at arbitrary times. On the other hand, Planets typically run on relatively powerful and stationary hosts. In a smart-office scenario, they can run on server clusters managed by IT staff. In a disaster-response scenario, they can run on servers in response vehicles interconnected with high-speed wireless networks.

Figure 3(a) shows relationship among sensors (S), applications (A), Planets (P), and operators (filled circles). A sensor may connect to any Planet, which serves as its *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 its 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.

4.1 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

```

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.TopTransportService
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 4: A portion of a Planet’s service configuration.

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 the 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 3(b).

A Planet has two kinds of message transports: normal TCP/IP and DHT (Pastry) transports. 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 these two transports to the appropriate Solar service, 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, so the directory users (fusion service in this case) do not need to know the internals (Section 5).

We found that our architecture based on service-level modules was 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. Jini’s local access to a remote service through a downloaded proxy shares a similar idea [64].

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 4. The configuration simply contains a set of key/value pairs, where 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 4, 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 directory service in Section 5 and the multicast service in Section 6. The dispatch service registers a callback with both transports to receive messages, which contain a multiplex header indicating the destination service.

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, it is possible to add a Web proxy service on the Planets. 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. An example for such a distributed Web cache is Squirrel [36], whose construction could be simplified by using the existing services, such as directory, multicast, and fusion.

5 Resource Discovery

To compute the desired context, Solar applications typically select some existing data sources and compose them with some operators into an operator graph (Section 3). 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.

In addition to typical `advertise` and `query` interfaces, Solar’s naming service also supports persistent queries and context-sensitive advertisements and queries. In particular, our Context-Sensitive Resource Discovery approach enables data-driven context composition, which is important for dynamic and volatile smart environments. Again, CSRD reflects a data-centric design in which the data value is used to dynamically construct operator graphs. The rationale is that a name advertisement or query itself often depends on the value of some contextual data.

Suppose Alice wants to record her activities in the office and she has one camera for the work space on her desk and one for the meeting space around the table. A SmartCamera application could then determine which camera to turn on for video recording, based on her current location or meeting status. In this case, the dynamic meeting context is used to identify appropriate cameras. This scenario requires a context-sensitive name query, that is, the results of the query for “a camera where Alice is right now” resolve to one or the other camera depending on her current location.

In another situation, the camera may be mobile. Consider the first-responder scenario, in which rescue workers might wear helmets with small attached cameras and a wireless-network interface. If these cameras are named according to their location, derived automatically from a localization system possibly using wireless triangulation, a supervisor’s monitoring application can request images of a particular area by selecting cameras whose location (in advertisement) matches the area of interest. The display automatically adjusts when a responder 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.

5.1 Name specification

There are several possible ways to represent names. The name space could be organized as a tree, as in many file systems; or as a set of descriptive attribute-value pairs [34]; a hybrid approach is also feasible [7]. We discuss their tradeoffs on complexity and expressiveness in another paper [20]. Solar simply represents a name advertisement or query as a set of attributes, and we say that an advertisement matches a query if the query’s attributes are a subset of the advertisement’s attributes. An example for a temperature sensor could be

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

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

```
[camera=$cam-filter:camera, room=116, building=Sudikoff]
```

the value of the “camera” attribute is defined by context information derived from an operator called `$cam-filter`. Specifically, the operator publishes events which, we assume, contain an attribute whose tag is “camera”. The value of the camera attribute is substituted here, so this name specification changes every time a new event arrives from the `$cam-filter` operator. Every context-sensitive name specification must be accompanied by an operator graph that defines the desired context computation.

Figure 5 demonstrates these concepts using the smart-room example. At top left, an operator determines the current meeting status and publishes an event whenever the status changes. Note that this operator could be deployed and registered in the namespace by the application that automatically mutes the phone when there is an active meeting. Its subscriber simply outputs an one-attribute event (shown in italics) indicating which camera should be used based on the meeting status. The event indicates there is an active meeting and the meeting camera should be chosen. At lower right, the SmartCamera application uses the context-sensitive query

```
[camera=$cam-filter:camera, room=116, building=Sudikoff]
```

to identify and subscribe to the camera covering the meeting space. If `$cam-filter` refers to the event stream produced by the filter, then `$cam-filter:camera` is resolved by the filter’s events. As shown, the `camera` attribute of the query is resolved to be *meeting*, which matches the advertisement of the camera in the lower left.

To arrange the context-sensitive subscription depicted in Figure 5, SmartCamera composes an operator graph using syntax described in Section 3. Note that the application could query the namespace to find out whether there is a meeting operator to reuse. If not, it can always compose a more complicated operator graph starting from motion and pressure sensors attached to chairs (see Figure 2).

The above example demonstrates the use of a context-sensitive name specification to support a context-sensitive subscription request from the SmartCamera application. A similar graph specification also could be used by an application that simply wishes to query the name service for a list of camera sources covering the meeting space, using the name specification

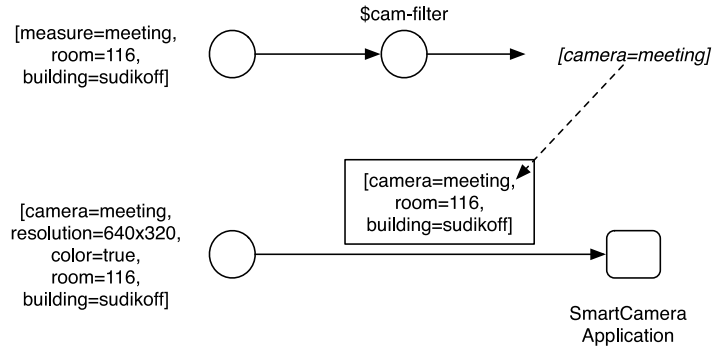


Figure 5: A smart-room example shows the use of the output from an operator graph to select an advertisement for the SmartCamera’s subscription.

```
[camera=$cam-filter:camera, room=116, 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.

5.2 Directory Service

The proposed Context-Sensitive Resource Discovery, however, places 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).

Solar uses a distributed directory service to implement the naming and discovery functionalities. It leverages our context-fusion infrastructure to help resources to make context-sensitive advertisements, and to allow applications to make persistent and context-sensitive queries. In our previous implementation, the directory service was wrapped around the Intentional Naming System (INS) [2], and we present our evaluation in [21].

There is some limitation, however, to use INS given that the namespace is fully replicated and it combines naming and messaging in ways that we do not need. We later decided to adopt a different design, to leverage our DHT-based overlay. To insert a name into namespace, Solar splits the name into attributes, which are then hashed into numeric keys. A copy of that name is then inserted into the directories on the Planets whose ID is closest to the hashed keys. Thus a name could be stored in multiple Planets for redundancy purposes. The name query, on the other hand, is sent to the Planet whose ID is closest to the hashed key of the longest attribute. The Planet receiving the query will compare the query against its local directory for any matched names. Similarly, a persistent query is registered on the Planet whose ID is closest to the hashed key of the longest attribute. For new name advertisements, a Planet will match them against registered persistent queries in local repository. This attribute-based name replication approach is similar to the one adopted in Twine [7].

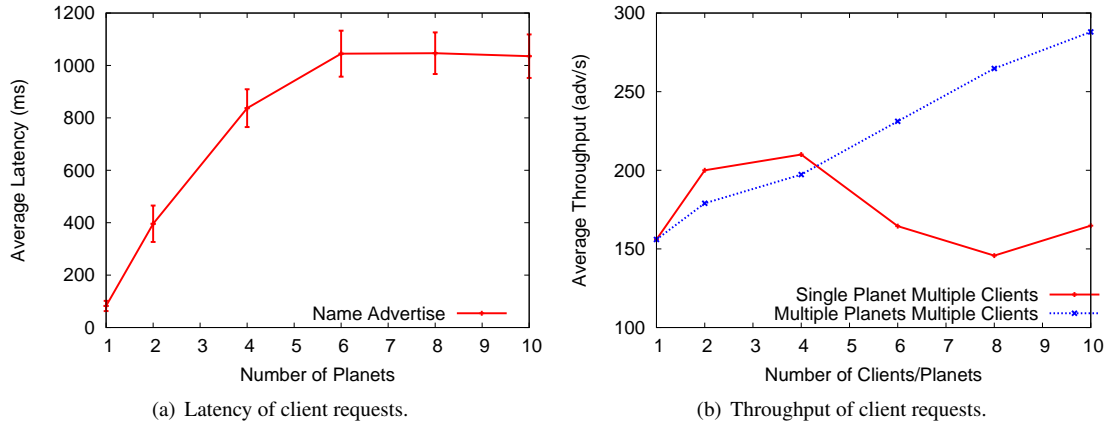


Figure 6: Performance of Solar naming service.

5.3 Performance Evaluation

Next we evaluate the performance of naming subsystem, containing multiple Planets and Solar clients, which issue name advertisement requests (name query and update involve similar attribute operations). Every client continuously issued advertisement requests using the RPC service (so requests were issued sequentially per client), which used a thread pool to handle incoming requests. We ran Planets and clients on our department’s Linux cluster, which contains 32 nodes with a head node for management. The cluster has 64 processors, that is, two processors for each node. Each node runs RedHat Linux 9 and Java virtual machine 1.5.0. Each node has 4GB RAM, an Ultra320 36GB hard drive, and gigabit Ethernet. Our Solar system uses Pastry version 1.3.2, and we use the SHA message digest to hash name attributes so they will spread evenly in the Planet ID space (160 bits).

We first measured the latency (for a client to complete a request) as we increased the number of Planets. Here a client issued a name with five attributes. Figure 6(a) shows the results. With one Planet, the mean latency was 82ms, where the client-Planet communication used TCP transport service. Interestingly, the mean latency increased to 396ms for two Planets. This is because the naming service would replicate the names according to attribute hashing, thus incurring round-trip communications between Planets. With 5 attributes, it likely triggered at least two inter-Planet requests, through the Pastry DHT transport service. As we further increased the number of Planets, the latency continued to increase until we added more than 5 Planets, after which only 5 inter-Planet requests were necessary.

This result represents a clear tradeoff of the replicated naming system. With attribute-based replication, a request needs multiple inter-Planet communications to complete the replication, thus trading performance with reliability. If both the number of name attributes and the number of Planets are large, the request latency could be large. In practice, for the applications we have developed, while the number of Planets may increase as needed, the number of name attributes seemed to be limited. One limitation of the attribute-based replication is that it is not easy to control the number of replicated copies. In future, we plan to investigate other replication methods that are independent of the number of name attributes [3, 43]. We expect changing the naming service is relatively easy given our architecture.

We can also improve our current system performance. Pastry version 1.3.2 uses Java non-blocking IO for its

network communication, which seems to be inefficient. It takes well over 100ms (with large variance) for a round-trip between two cluster nodes, even without any name processing. We plan to use the latest Pastry release that provides us a Java socket-based communication library with better performance.

Next we increased the number of clients to stress the workload on Planets and calculated the transaction throughput, the number of successfully completed requests per second for each client. Note that Solar RPC service has a 3-second timeout and the timed-out requests were not included in the calculation. Here every one client on the X axis represents a single cluster node running 50 parallel requesting processes. To avoid putting too much load on a single cluster node, we ran only 50 requesters on one node and only one Planet on a separate node.

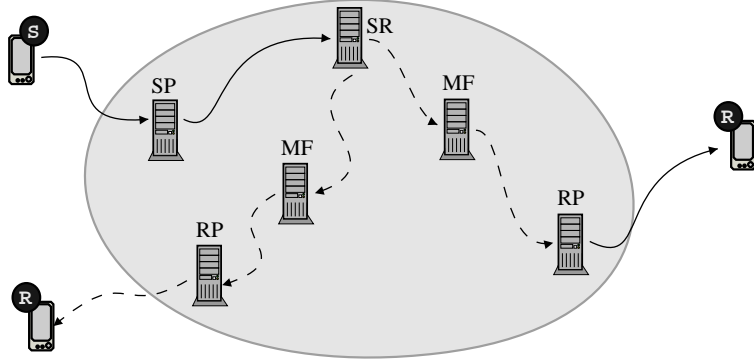
As we can see in Figure 6(b), for a single Planet, the average throughput peaked with 4 clients with a little over 200 requests per second. Then the average client throughput decreased as we added more clients and the Planet became congested. The average throughput reduced to about 150 requests per second with 8 clients, and slightly increased with 10 clients, because the timed-out requests made room for new requests. If we increased the number of Planets, in proportion to the number of clients, as shown the other curve, the average throughput achieved almost a linear increase with the number of clients. The Planet overlay builds up the capacity despite of communications between themselves. The nice thing about the self-organized overlay is that it is easy to add one Planet – just start a Planet with a specified address of an existing node for bootstrap and all Solar services automatically extend to the new Planet. One can imagine that a Solar deployment may have standby Planets and only activate them as load exceeds current overlay capacity.

6 Data Dissemination

In an operator graph, a sensor or operator may have multiple downstream subscribers, forming a fan-out structure; every time the sensor or operator publishes an event, the event has to be delivered through all the connected outbound 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 [27]. 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 [37, 57, 16, 61], buffer overflow management remains a challenge in ALM. In particular, we believe buffer overflow will be a serious problem for smart-environment applications that need to constantly monitor a large number of sensors for rapid adaptation, while the data streams may overwhelm the applications or the supporting infrastructure.

With the same data-centric design principle, we adopt a data-driven approach to handle buffer overflow on the dissemination paths. Namely, Solar supports application-specific policies that reduce the data volume based on data values, which are prioritized according to the application’s semantics. First, we present the basics of ALM over



7.a Multiple data dissemination paths converge to a multicast tree.

```

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

```

7.b An example of data-reduction policy.

Solar, and then motivate the data-reduction requirements. Then, we present our method for data-reduction policy specification and the evaluation results in the rest of this section.

6.1 Application-level multicast

The Planets of the Solar system serve many clients, each of which host sensors or applications. 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 ID randomly chosen from the same ID space. The subscriptions of a sender S are managed by S 's *root* Planet, whose ID is numerically closest to S 's ID among all live Planets. Note that the root Planet for S 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 7.a, 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 .

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 [17].

6.2 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 will run out of space. We say that a buffer overflows if it is full but new events continue to arrive (a buffer has limited space either because of memory constraint or there is a configured threshold to avoid long queueing delay). Furthermore, the events for a mobile receiver R are buffered at the proxy Planet RP during R 's disconnection. In this case 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.

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 in any case 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 [47]. 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.

We present a data-driven buffer-management module, named PACK, for the Solar 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.

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) [23], namely the Solar events. Since PACK is able to separate the events that follow a common attribute-based structure (Section 2), PACK can get the *values* inside the event object, enabling a much more flexible and expressive policy space for receivers.

6.3 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 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.

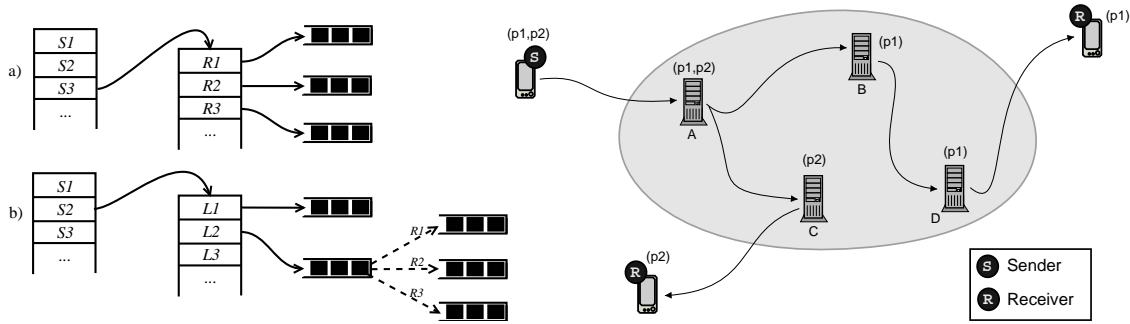
A filter is instantiated with application-defined parameters and determines what events to keep and what to drop given an event queue as input. The filters are independent, do not communicate with each other, and do not retain or share state. Since an event may contain several attributes, the filter 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 *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.

We show an example policy in Figure 7.b 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 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 semantics as their names suggest.

As indicated in Figure 7.b, 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 [12]. 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.



8.a Two-level indexes of local and remote buffers. 8.b The multicast service consists of a set of Planets with PACK policies.

6.4 Buffer management

A PACK host puts all events, either from a local sender or from the network, into its buffer waiting to be consumed by a local receiver or transmitted to the next host on the path. 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 endpoints on the same host, and the other is the *remote buffer* containing events to be transmitted to clients or some overlay node. 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 endpoints and one remote buffer for all destinations.

Both local and remote buffers adopt a two-level indexing structure (shown in Figure 8.a), 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 (overlay) 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, a private queue is created for each receiver and packed using its individual policy.

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 with 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 all the event duplication is based on references, not object instances. Figure 8.a shows private queues in the lower right.

Note a queue may be associated with multiple policies from receivers subscribed to the same sender. During queue overflow, all policies will be executed and the results are kept separated to avoid conflicts. This means that the amount of buffer state increases as the number of policies increases, posing a potential scalability limitation on PACK buffer and preventing a wide-area deployment with hundreds of thousands receivers. We are willing to pay this price to have expressive policies since most of our current applications are targeted at a campus-wide deployment with a limited

number of subscribers for individual data sources. It is possible, however, to increase scalability by limiting the policy flexibility [12].

6.5 Ladder algorithm

When packing an event queue is necessary to prevent overflow, PACK must determine which filters to apply. Packing with too many filters may unnecessarily drop many important events. On the other hand, packing with too few filters 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 k (initially one). Once k is determined given a packing request, all events in the queue pass through filters 1 to k in sequence. 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_l , 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 capture 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 historical turnaround time of 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.

6.6 Multicast service

Once the fusion service deploys operator graphs for an application, it uses the multicast service to set up the subscription channels. Note that a logical channel may actually represent a multi-hop path across Planets depending on the peer-to-peer routing protocol. In addition to the policies at the end hosts, the multicast service installs data-reduction policies on the buffers of the intermediate forwarding Planets, so they can be triggered closer to congested links or disconnected clients. Policies are installed on all the hosts along the path from sender to receiver.

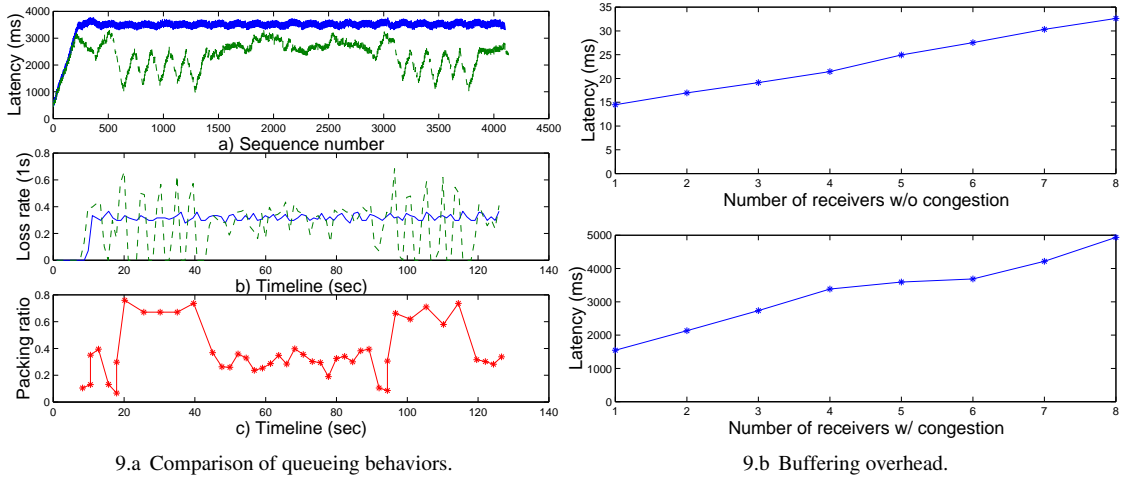


Figure 9: Performance of Solar multicast service.

Figure 8.b 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 $p2$). 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 and whether to add a digest. On the receiver’s client, PACK pulls events or digests from the queue and invokes the receiver to process each one.

6.7 Performance Evaluation

We first present experimental results using the Emulab testbed at Utah, in which we focused on measuring the performance of the PACK buffers inside the infrastructure. Next we give an application study of the PACK buffers on a client that tracked a large number of wireless devices on campus.

6.7.1 Queuing tradeoff

To measure the queuing 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-filter PACK policy, each filter randomly throwing out events (10%, 25%, and 50% respectively). We show the results in Figure 9.a (solid line for using DropTail and dashed line for PACK policy). In all the tests we turned off the just-in-time compiler and garbage collector in the Java VM.

Figure 9.a(a) shows the latency perceived by the receiver. After the buffer filled up, events in the DropTail queue had a (nearly constant) high latency because each event had to go through the full length of the queue before transmis-

sion. On the other hand, events in the queue managed by the policy exhibited lower average latency because events were pulled out of the middle of the queue, so other events had less distance to travel. From these results it is clear that PACK reduced latency by dropping data according to application’s semantics, and it is desirable for applications to use filters that are more likely to drop events in the middle (such as EVERY, RANDOM, GUNIQ) rather than at the tail (such as LATEST). On the other hand, DropTail has a roughly constant loss rate and thus has advantages under certain situations where fairness across applications is desirable. For the context-aware applications we have considered, however, we think a faster delivery of more important events is more reasonable.

Figure 9.a(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 pack operation was performed. Figure 9.a(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). Our heuristic algorithm worked reasonable well, although the filtering level varied despite the steady publication rate. The reason is that the RANDOM filter dropped varying amounts of events and our ladder algorithm adapted to longer or shorter inter-packing intervals by adjusting the filtering level.

6.7.2 Buffering overhead

We also measured the overhead posed by the two-level indexing buffer structure (see Figure 9.b). 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 publishes 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 setups, 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 drops a certain fraction of events from the tail of the queue (10%, 25%, and 50% respectively). In both setup, we measured the event delivery latencies at all the receivers and show the averages in Figure 9.b.

The first plot in Figure 9.b 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 network congestion forced a multicast stream to split into several substreams with different policies. Although still a linear increase, the added latency perceived by the receivers 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 method gives us the advantages over other approaches, such as TCP, that blindly pushes back on the sender, whose queue may eventually overflow and crash while the receiver has no method to learn about 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 [12]. 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 [46].

6.7.3 Application study

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 (APs), 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, or leaves the network.

One of our goals is to provide an IP-based location service: given a wireless IP address, the service can identify the AP where the device is currently associated. This service enables us to deploy location-based applications, often without modifying legacy software. For instance, we modified an open source Web proxy so it can 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.

To provide this kind of service, a locator subscribes to the syslog source and monitors all devices' association with the network. The 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. The locator uses a PACK policy consisting of six filters:

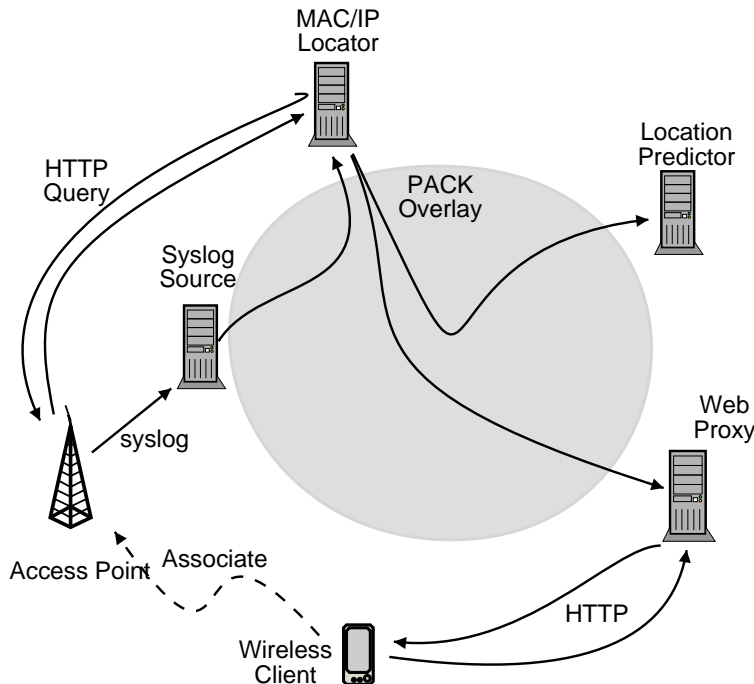


Figure 10: The MAC/IP locator monitors the syslog message stream and polls the AP for MAC-IP mapping. Then the locator publishes another stream with location updates of the mobile device. The Web proxy and a location-prediction service subscribe to the output of the locator to, for instance, push location-related content to clients.

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;
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 an hour-long run and Figure 11 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 the filter 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 workload carefully to configure efficient policies. The lower-left plot indicates that PACK triggered the policy rather frequently, with the median interval approximately 11 seconds. The lower-right plot shows the latency distribution, derived from the time the AP query is resolved and the timestamp in the original syslog event. Although

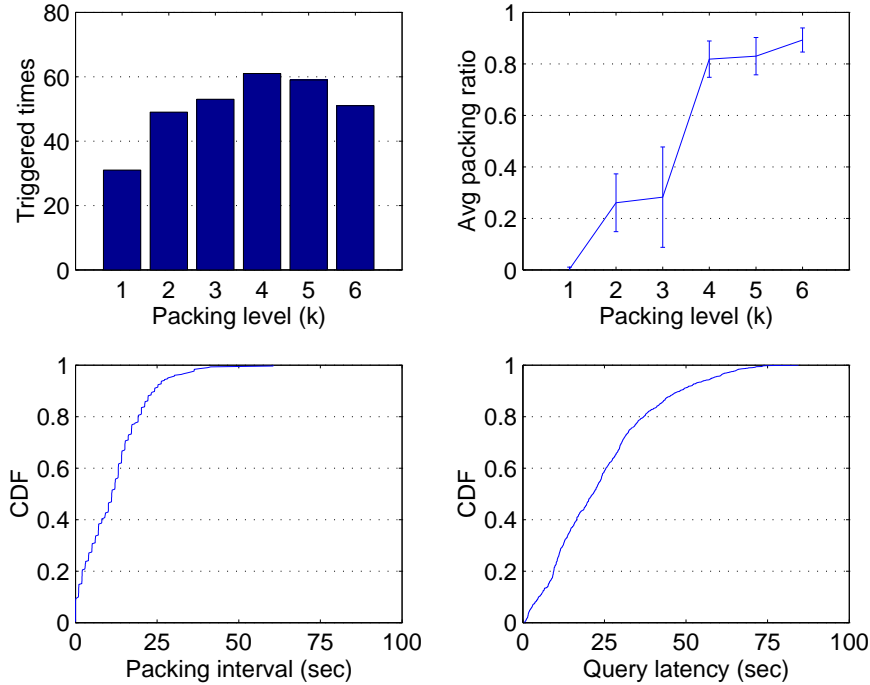


Figure 11: Statistics derived from a one-hour trace collected by the MAC/IP locator.

we set the connection timeout to be 30 seconds for each query, the longest delay to return a query was 84 seconds; some AP was under heavy load and slow to return results even after the connection was established.

The locator could adapt to situations when 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 then none of those events need to be transferred across the network. The source, however, might become the bottleneck as the number 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 [24]. PACK complements these systems to deal with queue overflow issues.

6.8 Grouped data selection service

A multicast service can save network bandwidth when disseminating a high-volume context source to multiple applications across the network. Yet, these context-sharing applications may use context in different ways. For instance, in an emergency-response scenario, suppose that a temperature sensor produces events for use by fire-trend prediction applications as well as personal-safety monitoring applications. The fire-trend prediction applications (which may re-

side on well-connected servers) may demand a temperature update on the scene every millisecond; the safety-checking applications (which may live on responders’ PDAs) may be content with a temperature update every 5 seconds. A typical Solar-based solution would have each application deploy an operator graph that includes the necessary filtering operators to select the desired subset of the data for each application’s needs. For saving network bandwidth, those filters are normally deployed near or on the source node. But then each application produces its own customized sequence of events, and the per-stream based multicast service cannot be used. The output of a source node where many filters are deployed may easily become a data dissemination bottleneck. This problem is especially prominent in a pervasive setting with wireless mesh networks where network bandwidth is precious.

The problem motivates us to provide *grouped data selection* at a node from which a high-volume streaming source is disseminated to a group of applications. This service trades off computation for bandwidth savings. Here we make use of two overlooked, yet important, observations about context aggregation: 1) many pervasive applications can tolerate some degree of “slack” in the quality of the context information, and 2) applications with quality slack may find more than one subset of a context source that satisfies their quality requirements. The middleware thus can exploit the equivalence in the potential subsets for the applications and pick the subsets that maximize the overlap among the applications. This method reduces the total output data bandwidth of the group when multicasting is used.

Here is a simple example that illustrates the basic idea of grouped data selection. Assume that all applications are interested in receiving subsets of the data stream, using a “delta” function that skips values that have changed less than a certain threshold amount from the previously reported value. Considering each application’s requirement alone, we find the set of points chosen by this delta function and call them *reference points*. Now, applications may declare a “slack” of e units, relative to these reference points. That is, they may be willing to receive some points before or after the desired reference points, as long as the value is no more than e above or below that of the reference point. For instance, consider a sequence of six temperature readings $S : (10, 13, 14, 15, 18, 20)$. The reference points for a 5-unit delta-compression are $S_1 : (10, 15, 20)$. If the application declares a 2-unit slack based on these reference points, any point that is within 2 units from a reference point can replace that point without affecting the data quality. We notate such n -unit delta-compression with e -unit slack as (n, e) compression. It is easy to see that $S_2 : (10, 14, 18)$ also meets the $(5, 2)$ compression requirements, as 14 is within 2-unit distance from reference point 15, and 18 is within 2-unit distance from reference point 20. Similarly, a 3-unit compression query would choose 10, 13, and 18 as its reference points; a $(3, 1)$ compression may choose $(10, 14, 18)$ as a valid output. If there are two applications, one using $(5, 2)$ and the other using $(3, 1)$ compression, picking $(10, 14, 18)$ for both outputs is better than picking $(10, 15, 20)$ for the first query and $(10, 13, 18)$ for the second query; we can multicast the three-tuple sequence to both applications, rather than sending separate three-tuple sequences to each, or sending the union (a five-tuple). The result is a savings in bandwidth over the original six-tuple sequence, or over the separately compressed sequences.

Our grouped data selection approach considers *group utilities* of data points. We define the group utility of a data point as the number of application selectors that would accept this data point. To compute group utilities, we must compute “candidate sets”, subsets of tuples from the source data, for replacing reference points for each application. In the previous $(5, 2)$ compression example, we can compute the candidate set for replacing reference point 15 by

including nearby data points that fall into the range of $15 - 2 = 13$ and $15 + 2 = 17$. This method gives us the candidate set $\{13, 14, 15\}$ for replacing 15. By the same token, for $(3, 1)$ compression, the candidate set for replacing the reference points 13 is $\{13, 14\}$. If the group includes only these two applications, the group utility is 2 for data points 13 and 14 respectively and 1 for data point 15. Thus, if both selectors select 13 or 14, that will reduce one data point in the final output of the source.

We have studied an extensive set of data selectors, ranging from simple value-based filters, random load shredders, to selectors as sophisticated as reservoir sampling operators. We find that the group data selection can be applied to those selectors by working through the following two phases. In the first phase, each selector gathers candidate sets for the reference points that satisfies its quality needs. In the second phase, based on the candidate sets, we compute the group utility of every data point. When a selector needs to decide which point(s) to choose in the candidate sets, it picks point(s) with highest group utility. For some selectors, such as heavy hitter sampling, it may be beneficial to be group-aware even in the first phase when it randomly selects candidate points. We are currently working on extending this framework for these more general cases. Currently, the subscription file of an application consists of its filter type and the filter's parameters such as delta and slack values in the case of a delta-compression type filter. For future work, it is desirable to make the filter parameters adaptive to the constraints of the system resources.

7 Solar Applications

In this section we first present a set of pervasive-computing applications we built with Solar to realize the scenarios described in Section 1. We then discuss our experience and lessons learned using a data-centric middleware to support context-aware applications.

7.1 Smart building

We have installed an infrared-based badge-tracking system that covers our department building, to identify the location of badge-wearing people on a room-by-room granularity. Solar was then used by students to develop several applications, such as a wall-mounted Web portal that shows content personalized for the approaching user, and a SmartReminder application that alerts its user about upcoming events at a time that depends on her current location and the location of next appointment [45].

We instrumented an office by attaching motion and pressure sensors to the chairs, then aggregated their outputs to detect whether a meeting was in progress, based on which a SmartPhone application automatically routed incoming calls to voice mail without interrupting the meeting. By aggregating two simple sensors using a state-machine based algorithm, we were able to significantly increase the detection accuracies and decrease detection delays. Interested readers can find more details about this implementation and performance evaluations in our earlier paper [65].

Note that the SmartPhone application aims to adjust phone settings without requiring a user remember to press/reset a “mute” button before/after the meetings; it works automatically with minimal distraction. Although we had a single-office deployment, for which a simple centralized solution may be sufficient, we believe a middleware like Solar is necessary when deploying in multiple offices and when the meeting-detecting sensors may be shared by

other applications. In fact, the motion sensor attached to the chair is embedded in the IR badge we use for a location tracking system. An assert management application may also want to subscribe to the output of the IR badge and its sensor through Solar.

7.2 Emergency response

As an infrastructure for information aggregation and dissemination, Solar was used in the Automated Remote Triage and Emergency Management Information System (ARTEMIS) [26], an ongoing project sponsored by U.S. Department of Homeland Security since 2003. The long-term goal of the project is to build a remote triage system that can expedite and improve care of the wounded in small- or large-scale emergency situations, and ultimately to provide an unprecedented degree of medical situational awareness at all levels of the first-responder command hierarchy.

To evaluate and validate Solar in such mission-critical settings, we started by deploying it in a small scenario that involves local emergency management services with tens of responders, casualties and sensors in an accident that lasts for several hours. Specifically, it is a scenario based on a real rescue and triage exercise by the local police department and fire department at Lebanon airport (NH) in the spring of 2004, simulating an airplane crash during takeoff in which dozens of passengers were injured.

In this scenario, responders deployed many kinds of sensors and applications. Pulse oximeters, which were attached to responders, measured blood oxygen saturation and effectively captured the cardiac and respiratory states of the sensed responders. These real-time physiological states were continuously pumped into the PDA that each responder carried, providing continuous input to a triage status classification algorithm running on the PDA. Each GPS-enabled PDA provided responders' health and location information to remote subscribers through the wireless network. To monitor the surroundings of the incident, hundreds of environmental sensors, such as temperature sensors, were also deployed on the scene. Along with the infrastructural sensors built in the airport runway, these environmental sensors monitored the relevant environmental states and sent the data to nearby sensor gateways.

The ARTEMIS command and control applications include 1) those that continuously monitor the fire propagation status, track the location of each first responder and report injured responders and their physiological status, etc; and 2) those that can later playback the history of the location or physiological status of the responders in question; and 3) those that probes the history of sensor data and answer one-shot queries by the commanders for situation assessment purposes. Figure 12 and Figure 13 show an application GUI running on a PDA carried by responders and at a remote EOC (Emergency Operation Center), respectively.

In the spring of 2006, we used Solar in a larger-scale disaster recovery scenario. We simulated a disaster where a train containing chlorine derailed in downtown Baton Rouge, Louisiana. In this scenario, we tested out the grouped data-selection service on a fast-rate chlorine-monitoring source shared by several delta-compression applications. The source data was simulated according to a diffusion model that was carefully engineered for this scenario. The model considered many factors such as wind direction, wind speed, and the density of the sensors. Based on the functions of the roles, we specified realistic data granularity requirements of the applications used by the roles. In this testing, we found that the grouped data selection had a further bandwidth saving of about 15% over the independent data selection,



Figure 12: A front-end application running on the mobile device carried by first responders.

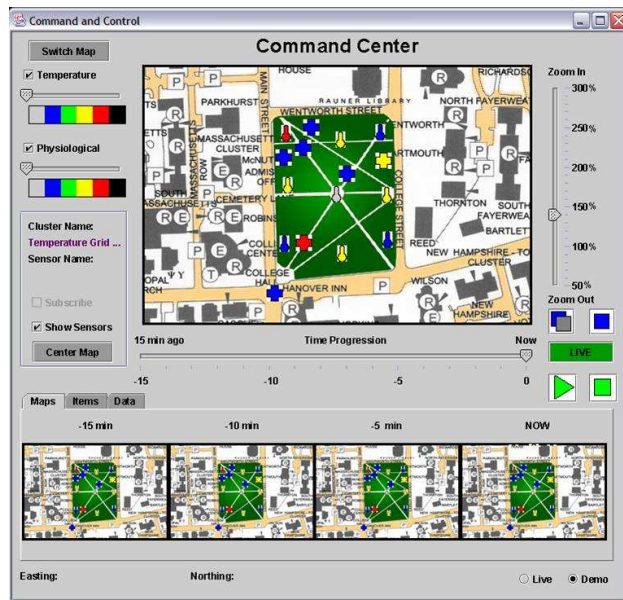


Figure 13: A command and control application running at remote EOC to provide situational awareness.

and that the group selection algorithm was also efficient: processing sixty tuples took less than a quarter of a second.

7.3 Lessons learned

We note that applications may use context *actively* or *passively*. In smart-building scenarios, many applications take actions directly, according to context changes: the SmartPhone 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 likely caused by the complexity of decision making and the cost of mistakes in different environments.

Our original design on name representation supports attribute hierarchy, namely, the value of an attribute could be another attribute-value pair. In practice, we found Solar developers simply choose flat attribute structures, not bothering with a name hierarchy. This suggests that a flat attribute name is sufficiently flexible and expressive. A hierarchal name representation could be more efficient for processing and give more structure [20], but the developers do not seem to need it.

As described in Section 3, the channels connecting operators are push based. The downstream subscribers have no control over upstream publishers except processing received events. In some cases, we found that a pull-based channel might be more efficient. Namely, the sink could inform the source when to send the data. For instance, a meeting-fusion operator may only check the pressure sensor when the motion sensor reports activity (Figure 2(a)) to reduce data traffic. We are considering ways to support both pull and push channels, similar to the abstraction used by iQL [24].

We also observed that sometimes the Solar programmers desired to have more control over operators, which are passive event handlers. One drawback of Solar was a lack of built-in timeout facility, so an operator could not gain control unless it receives an event; on the other hand, some operators need to react to a *lack* of events as well as new events. Another drawback is that applications cannot adjust the behaviors of a deployed operator graph, such as changing a filtering condition. Solar could be extended to provide a control mechanism on the channels so the downstream subscribers can instrument upstream publishers. Like adding a pull interface to the channel, however, an operator is no longer a passive event handler. Instead, it needs to be able to identify and communicate with upstream publishers. As a tradeoff, the complexity for the operator composition and the supporting system increases significantly.

Solar operators may have transient state, which will be lost if the operators are reclaimed after the application quits. Some applications, on the other hand, need to query some historical state information. We are adding a storage service to Solar so operators have access to persistent storage. Our DHT-based P2P routing substrate could be leveraged to easily build a redundancy-enabled storage service. The tricky challenge is to identify which state from which operator instance should be retrieved.

Currently Solar does not explicitly support any ontology and thus cannot guarantee semantic compatibility between heterogeneous events. We believe that an ontology service, such as the one described in [33], could be easily added into Solar's service-oriented architecture (Section 4). Such a change in protocol would not require rewriting of applications

given Solar’s layered architecture.

8 Related Work

Xerox PARC’s ActiveMap was one of the first middleware 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 [60]. 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.

The Context Toolkit seeks to simplify the development of context-aware applications by wrapping sensors with a *widget* abstraction and by providing pre-defined aggregators for commonly used context [28]. The widget is a powerful analogy that hides the details of heterogeneous sensors and exposes standard interfaces. Solar’s data abstraction could also be built on top of sensor widgets, but Solar further regulates a common structure in sensor events to allow data-driven system adaptation. The Context Toolkit has a data-flow representation for its aggregators that bears some similarity to Solar’s operator graph; the system structure, however, is established by administrators and becomes static at runtime. Solar allows application-specific context customization at runtime to meet the diverse needs of different applications.

The Context Fabric also focuses on the interface between system and applications [35]. 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 [41] and CANS [31]. 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 scalable architecture, dynamic resource discovery, and flow control for sensor data.

In industry, ContextSphere (formerly called iQueue) shares some similar goals with Solar [25]. 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 [24]. 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 [20]. The current 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 [38]. This loosely-coupled coordination model reduces component interdependency and allows easy recovery from crashes. But, the simple interface of a tuplespace, with tuple retrieval based on pattern matching, limits the expressiveness of data processing.

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 [42], specification toolkits [54], support systems [15, 14, 56], and load balancing and stability algorithms [55]. 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 a certain quality of composed service.

Other emergency response and battlefield projects, such as Rescue [48], EventWeb [19], and JBI [10, 9], 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 [11, 34, 44]. 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.

Recent work using an overlay of event brokers to provide a content-based publish/subscribe service has been focused on routing and matching scalability and has largely ignored end-to-end flow control [8, 13]. Pietzuch and Bhola, however, study the congestion-control issues in the context of the Gryphon network during the course [53]. Congestion in the whole system can not 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. Solar, on the other hand, does not employ classic congestion control to stop senders, which may not be feasible for sensors. Instead, Solar uses application policies for dynamic data reduction.

Solar’s context aggregation is similar to continuous query processing [1, 50, 63] in that the traditional store-index-then-process model does not fit the fast-rate streaming data. Thus STREAM [4] and TelegraphCQ [18] extend the SQL language to manipulate streaming data by introducing window-based relational operators such as *join* for approximate query results. Unlike them where a query plan is generated based on the SQL-like language, Aurora [5] and Solar let applications compose box-and-arrow graph (boxes are the operators, possibly defined by applications, and arrows are pipes between a pair of operators) to describe the data flow. One of the challenges in stream data processing is to manage the data load at the operators. STREAM lets applications specify a sampling rate in their SQL-like language to shed the load. Aurora dynamically adds load shredders that randomly shed data load based on simple QoS functions. Johnson et al. [39] generalize the processing of a sampling operator to a four-stage process. Solar’s group-aware data reduction can be seen as a special form of sampling, as the outputs are selected from a candidate set of outputs, yet our

data reduction approach leverages the approximate nature of applications' quality requirements and is group-oriented, while the sampling or filtering operators elsewhere [52, 22, 39] are oriented to a single application.

9 Conclusion

We believe that a data-centric infrastructure is necessary to support context-aware pervasive-computing applications. In the Solar infrastructure our implementation systematically integrates several data-driven design choices. We present our data abstraction, programming model, architecture design, and two key services (resource discovery and data dissemination) and their evaluation results. Solar has achieved our goal as a data collection, processing, and dissemination infrastructure for context-aware applications. It appears to be reasonably efficient, scalable, and fault tolerant. We have deployed Solar to support both smart-building and emergency-response applications. The data-centric design facilitates application development with a data-flow composition model, and allows the system to be more adaptive by exposing structure and values in the data flow. Our experience with Solar also shows that our data-flow model worked well, though it could be enhanced by adding state-storage capabilities and support for a semantic model of context and application state.

Acknowledgments

We gratefully acknowledge the support of the Cisco Systems University Research Program, Microsoft Research, the USENIX Scholars Program, DARPA contract F30602-98-2-0107, and DoD MURI contract F49620-97-1-03821. This research program is a part of the Institute for Security Technology Studies, supported under Award number 2000-DT-CX-K001 from the U.S. Department of Homeland Security, Science and Technology Directorate and by Grant number 2005-DD-BX-1091 awarded by the U.S. Department of Justice, Bureau of Justice Assistance. Points of view in this document are those of the authors and do not necessarily represent the official position of the U.S. Department of Homeland Security or its Science and Technology Directorate, the United States Department of Justice, or any other sponsor.

We also want to thank those who have used Solar to build interesting applications, such as a location-aware personal event reminder (Arun Mathias), the meeting detector for smart office (Jue Wang), and the ARTEMIS command and control for first responders (Ronald Peterson, Michael De Rosa, Christopher Carella, and project director Dr. Susan McGrath).

References

- [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] 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, SC, December 1999.

- [3] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. [A survey of peer-to-peer content distribution technologies](#). *ACM Computing Survey*, 36(4):335–371, 2004.
- [4] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. Stream: the stanford stream data manager (demonstration description). In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 665–665, New York, NY, USA, 2003. ACM Press.
- [5] Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Eddie Galvez, Jon Salz, Mike Stonebraker, Nesime Tatbul, Richard Tibbetts, and Stan Zdonik. Retrospective on Aurora. *VLDB Journal*, January 2004.
- [6] 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, 2003.
- [7] 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.
- [8] 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.
- [9] United States Air Force Scientific Advisory Board. [Report on Building the Joint Battlespace Infosphere, Volume 2: Interactive Information Technologies](#), December 1999.
- [10] United States Air Force Scientific Advisory Board. [Report on Building the Joint Battlespace Infosphere, Volume 1: Summary](#), December 2000.
- [11] 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.
- [12] 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, 2000. ACM Press.
- [13] 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), 2001.
- [14] 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.
- [15] 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.
- [16] 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.
- [17] 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.
- [18] S. Chandrasekaran. [Telegraphcq: Continuous dataflow processing for an uncertain world](#), sp2003.
- [19] 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.

- [20] 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.
- [21] 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, TX, March 2003.
- [22] Reynold Cheng, Ben Kao, Sunil Prabhakar, Alan Kwan, and Yicheng Tu. Adaptive stream filters for entity-based queries with non-value tolerance. In *VLDB '05: Proceedings of the 31st International Conference on Very Large Data Bases*, pages 37–48. VLDB, 2005.
- [23] 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, PA, 1990. ACM Press.
- [24] 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.
- [25] 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.
- [26] Dartmouth College. [Automated remote triage and emergency management information system](#). <http://www.ists.dartmouth.edu/projects/frsensors/artemis/>.
- [27] 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.
- [28] 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.
- [29] Mike Esler, Jeffrey Hightower, Tom Anderson, and Gaetano Borriello. [Next century challenges: data-centric networking for invisible computing: the Portolano project at the University of Washington](#). In *Proceedings of the Fifth Annual International Conference on Mobile Computing and Networking*, pages 256–262, Seattle, WA, August 1999.
- [30] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. [The many faces of publish/subscribe](#). *ACM Computing Surveys*, 35(2):114–131, June 2003.
- [31] 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.
- [32] David Garlan and Mary Shaw. [An introduction to software architecture](#). Technical Report CMU-CS-94-166, Carnegie Mellon University, January 1994.
- [33] Tao Gu, Hung Keng Pung, and Da Qing Zhang. [Toward an OSGi-Based Infrastructure for Context-Aware Applications](#). *IEEE Pervasive Computing*, 3(4), October 2004.
- [34] 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, Canada, October 2001.
- [35] Jason I. Hong and James A. Landay. [An infrastructure approach to context-aware computing](#). *Human-Computer Interaction (HCI) Journal*, 16(2-4), 2001.

- [36] Sitaram Iyer, Antony Rowstron, and Peter Druschel. [Squirrel: A decentralized peer-to-peer web cache](#). In *Proceedings of the 21th Annual Symposium on Principles of Distributed Computing*, pages 213–222, Monterey, CA, July 2002.
- [37] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and James W. O’Toole 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.
- [38] 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.
- [39] Theodore Johnson, S. Muthukrishnan, and Irina Rozenbaum. Sampling algorithms in a stream operator. In *SIGMOD ’05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 1–12, New York, NY, USA, 2005. ACM Press.
- [40] Apu Kapadia, Tristan Henderson, Jeffrey Fielding, and David Kotz. [Virtual walls: Protecting digital privacy in pervasive environments](#). In *Proceedings of the Fifth International Conference on Pervasive Computing (Pervasive)*, volume 4480 of *Lecture Notes in Computer Science*, pages 162–179. Springer-Verlag, May 2007. Honorable Mention for Best Paper.
- [41] 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.
- [42] Frank Leymann. [Web Services Flow Language \(WSFL 1.0\)](#).
- [43] Xuezheng Liu, Guangwen Yang, and DingXing Wang. [Stationary and adaptive replication approach to data availability in structured peer-to-peer overlay networks](#). In *Proceedings of the 11th IEEE International Conference on Networks (ICON2003)*, pages 265–270, Sydney, Australia, September 2003. IEEE.
- [44] 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.
- [45] Arun Mathias. [SmartReminder: A case study on context-sensitive applications](#). Technical Report TR2001-392, Dartmouth College, June 2001. Senior Honors Thesis.
- [46] 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, CA, August 1996.
- [47] 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.
- [48] 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.
- [49] Kazuhiro Minami and David Kotz. [Secure context-sensitive authorization](#). *Journal of Pervasive and Mobile Computing*, 1(1):123–156, March 2005.
- [50] 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.
- [51] Donald A. Norman. [The Invisible Computer: Why Good Products Can Fail, the Personal Computer Is So Complex, and Information Appliances Are the Solution](#). The MIT Press, August 1999.

- [52] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *SIGMOD '03: Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 563–574, San Diego, California, June 2003.
- [53] 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–, Rio de Janeiro, Brazil, June 2003.
- [54] Shankar R. Ponnekanti and Armando Fox. [Sword: A developer toolkit for web service composition](#). In *Proceedings of the 11th International World Wide Web Conference*, Honolulu, Hawaii, May 2002.
- [55] 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.
- [56] Bhaskaran Raman and Randy H. Katz. [An architecture for highly available wide-area service composition](#). *Computer Communication Journal*, 26(15):1727–1740, September 2003.
- [57] 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.
- [58] 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.
- [59] M. Satyanarayanan. [Pervasive computing: Vision and challenges](#). *IEEE Personal Communications*, 8(4):10–17, August 2001.
- [60] William Noah Schilit. *A System Architecture for Context-Aware Mobile Computing*. PhD thesis, Columbia University, May 1995.
- [61] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. [Internet indirection infrastructure](#). In *Proceedings of the 2002 ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 73–86, Pittsburgh, PA, August 2002.
- [62] 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.
- [63] 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.
- [64] Jim Waldo. [The Jini architecture for network-centric computing](#). *Communications of the ACM*, 42(7):76–82, July 1999.
- [65] 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.
- [66] Mark Weiser. [The computer for the 21st century](#). *Scientific American*, 265(3):66–75, January 1991.
- [67] Mark Weiser. [Some computer science issues in ubiquitous computing](#). *Communications of the ACM*, 36(7):75–84, July 1993.
- [68] 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.