8-2004

Design and Implementation of a Large-Scale Context Fusion Network

Guanling Chen
Dartmouth College

Ming Li
Dartmouth College

David Kotz
Dartmouth College

Design and Implementation of a Large-Scale Context Fusion Network

Guanling Chen, Ming Li, and David Kotz
Department of Computer Science, Dartmouth College
Hanover, NH 03755, USA

{glchen, mingli, dfk}@cs.dartmouth.edu

Abstract

In this paper we motivate a Context Fusion Network (CFN), an infrastructure model that allows context-aware applications to select distributed data sources and compose them with customized data-fusion operators into a directed acyclic information fusion graph. Such a graph represents how an application computes high-level understandings of its execution context from low-level sensory data. Multiple graphs by different applications inter-connect with each other to form a global graph. A key advantage of a CFN is re-usability, both at code-level and instance-level, facilitated by operator composition. We designed and implemented a distributed CFN system, Solar, which maps the logical operator graph representation onto a set of overlay hosts. In particular, Solar meets the challenges inherent to heterogeneous and volatile ubicomp environments. By abstracting most complexities into the infrastructure, Solar facilitates both the development and deployment of context-aware applications. We present the operator composition model, basic services of the Solar overlay network, and programming support for the developers. We also discuss some applications built with Solar and the lessons we learned from our experience.

1 Introduction

Applications in a computation and communication saturated ubicomp environment have to gracefully integrate with human users. The sheer amount of context information produced by a sensor-rich environment may 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. The context is time-sensitive, which means that the environmental states and interactions could either be historically recorded or currently happening.

Unlike explicit user input, context represents implicit input to the application. An application typically infers context information from various sensors. An inherent challenge for context-aware applications is that they have to work with knowledge of world state derived from potentially incomplete and error-prone sensory data. To remedy this situation, applications may have to aggregate data from multiple sources (either similar or different types) to improve the quality of computed context. The aggregation algorithms for data fusion may be simple logical combinations based on current sensory values, or may be more sophisticated machine-learning techniques based on historical observations. We call this process, computing higher-level understanding from lower-level sensory data, "context fusion". The output of the fusion process is the context to be fed into applications.

It is possible to integrate all the sensors on a single platform where an application runs, such as an augmented mobile phone [29]. We are, however, concerned about larger scenarios with physically distributed sensors and other online sources, while multiple applications on different devices may benefit from information aggregated from the sensor data. These applications require customized context, but may also share similar data-fusion steps over shared sensors.

Given the overwhelming complexity of a heterogeneous and volatile ubicomp environment, it is not acceptable for individual applications to maintain connections to sensors and to process the raw data from scratch.

Proceedings of the First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous'04)
0-7695-2208-4/04 $20.00 © 2004 IEEE

On the other hand, it is not feasible to deploy a common context service that could meet every application's needs either. Instead, we envision an infrastructure that allows applications to reuse the context-fusion services already deployed, and to inject additional aggregation functions for context customization and user personalization where necessary.

The thesis of our research is thus to provide such a system that is both flexible to meet diverse application needs and scalable to service hundreds and thousands of sensors, applications, and users. This system is not just a Content Delivery Network (CDN), which connects multiple sources and sinks; instead, the system needs also to accommodate application-specific data-fusion functions that deliver higher-level context to applications. We call the resulting system a Context Fusion Network (CFN). Compared to a CDN, a CFN needs to provide additional functionalities of managing application-specific data-fusion functions.

We consider two application areas that may take advantage of a CFN, *smart spaces* and *emergency response*. In both environments a fleet of applications derive their contextual information from many shared and distributed sensors. Consider an intelligent building with sensors embedded in every room, hallway, and appliance. Applications in such a space may want to track and react to users' location, activities, and so on. Consider also disaster and crisis response. The situation may involve a large number of sensors and devices carried by responders and victims, and deployed in environments beforehand or on-demand. People with different roles at local and remote sites need different contextual information to support their decision making. As the response evolves, the decision makers may use a CFN to quickly deploy new fusion functions that are not expected beforehand.

There are four challenges to be addressed, however, to provide a CFN in a heterogeneous and volatile ubicomp environment. First, a CFN must be *flexible*. It must allow both deployment of well-known context services and application specified customization and personalization. A CFN should also be flexible to accommodate arbitrary context-fusion algorithms chosen by individual applications. Second, a CFN must be *scalable* to handle a large number of sensors, devices, applications, and users. It should be easy to increase the CFN capacity to handle increased load as necessary. Third, a CFN must explicitly support *mobility*, both at physical and logical level. A moving device connecting to a CFN may traverse both geographic and networking boundaries. The context-fusion components may also have to migrate in the infrastructure to balance the load, or to efficiently use the bandwidth. Finally, the overall complexity of a ubicomp environment requires a CFN to be *self-managed* with minimum user intervention necessary. A dependable CFN must proactively monitor node failures, automatically recover lost components, and garbage-collect application-specific fusion components that are no longer in use.

Note that the sensor providers and application developers may not be the same party. Thus another set of challenges for a CFN is how to deal with data heterogeneity, both at the syntax and semantics level so interoperability can be achieved. The solution to this problem requires both tools and standards, for instance, from the ontology and semantic Web research community. We assume that the semantic functionality resides at a higher layer above the CFN, and we do not address it in this paper.

We have implemented an overlay-based distributed CFN, named *Solar*, to meet the four research challenges. The rest of paper is organized as follows. We present an operator composition model in Section 2. Sections 3 and 4 describe Solar's overlay platform and a set of basic services that are used to manage operator graphs. We discuss programming support for Solar applications in Section 5, and the applications that have been or are being built with Solar in Section 6. Finally we discuss related work in Section 7 and conclude in Section 8.

2 Operator composition

One of Solar's goals is to facilitate the development and deployment of large-scale context-aware applications. We note that many applications using the same or overlapping set of data sources need to go through similar data-processing steps to get usable contextual information. It is thus critical for Solar to provide a composable framework that promotes software re-usability.

We consider two kinds of reuse: one is *code-based reuse* that allows applications to import existing modules from documented libraries; and the other is *instance-based reuse* that allows applications to discover and use the service provided by already deployed data-processing components. From the application's viewpoint, Solar encourages a modular structure and reduces the programmer's task through code-based reuse. From the system's viewpoint, Solar minimizes redundant computation and network traffic to increase scalability through instance-based reuse.

2.1 Architecture style

One popular software architectural style for data-stream oriented processing is *filter-and-pipe* [30], which supports reuse through composition. In the 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

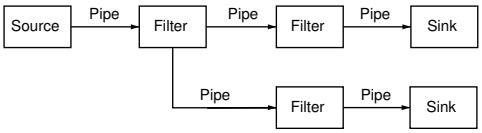

Figure 1. The filter-and-pipe software architecture style promotes reuse and composition.

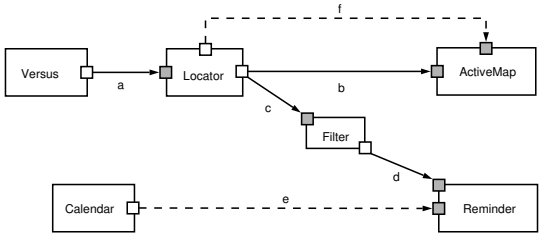

Figure 2. An example operator graph with two sensors and two applications. The shaded squares are inports while unfilled squares are outports. The dashed lines are pull channels and the solid lines are push channels.

streams of data on its outputs. A connector (pipe) serves as conduits for the streams, transmitting outputs of one filter to inputs of another. The data flow starts from a source, through a sequence of pipes and filters, and finally reaches a sink.

Some advantages of the filter-and-pipe style are [30]: 1) filters are independent and can be treated as black boxes to promote functionality isolation; 2) filters typically do not know the identities of their upstream and downstream components, which helps to ensure low coupling; 3) pipes and filters can be hierarchically composed; 4) the construction of the pipe and filter sequence can often be delayed until runtime (late binding) based on the current application state; and 5) it is relatively easy to run a pipe-and-filter system on parallel processors or in multiple threads on a single processor. Sample implementations of the filter-and-pipe style in practice, often centralized, include Unix pipes [1], structured Web servers, compiler construction, language design [31], and software routers [20].

2.2 Operator graph

We now present Solar's context fusion model, following the filter-and-pipe style. First we define some terms for the following discussions. In our terminology, a filter is an "operator" and a pipe is a "channel". A channel connects to a *source* at one end, and to a *sink* at the other end. A *sensor* provides (raw) data to Solar while an *application* consumes (contextual) data from Solar. A sensor is also a source and an application is also a sink. An operator is both a source and a sink.

An operator is a self-contained data-processing component, which takes one or more data sources as input and acts as another data source. Each operator has a set of *input ports* and a set of *output ports*. For simplicity, we call an input port an *inport* and an output port an *outport*. Each port has a unique identifier within its operator to distinguish it from other ports. A channel connects an outport of an upstream source to an inport of a downstream sink, and all data flows from source's outport to the sink's inport.

A port can be either *push-based* or *pull-based*. A channel connecting a push-based outport and a push-based inport is a *push channel*. A channel connecting a pull-based outport and a pull-based inport is a *pull channel*. With a push channel, the source voluntarily passes data units, which we call *events*, to the sink. With a pull channel, the sink sends an explicit request, called a *query*, to the source and the result is then passed through the channel. It is illegal, however, for a channel to connect two ports with mismatched push/pull types. Note that an outport may have multiple channels connected. For a push channel, we call the downstream sink as a *subscriber* who subscribes to the upstream source, or a *publisher*.

In Solar's model, events are encoded in a data structure called a *record*. A record is a set of attributes, each of which contains pair of a *tag* string and a *value* object. The value of an attribute could be another record to form an attribute hierarchy.

Figure 2 shows a sample operator graph. The "locator" operator receives location updates of every badge from a Versus sensor[1] and publishes location-change events through push channels, b and c. An ActiveMap application simply displays every badge's current location. A reminder application receives filtered location-change events about a particular user, and it also queries the user's calendar about location of the next appointment. Based on this context information, the reminder then decides the appropriate time to alert the user.

Note the operator "locator" keeps internal state (the current location of all badges) and only publishes an event when some badge moves. Whenever a copy of the ActiveMap application starts up, it bootstraps by pulling the channel f to get every badge's current location. Otherwise, it may not be able to show the location of slowly-moving badges, such as those attached to printers.

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

2.3 Functional separation

Solar's model separates two roles: operator developer and graph composer (typically the application). The developer is responsible for defining a *port specification* for each operator, including the identifier and push/pull type of all inports and outports. While currently not adopted in Solar, the specification may further specify the event structure (attribute tags and type of the values) intended for each port. Based on the port specifications, the composer can build an operator graph with channels connecting appropriate ports.

The port specifications should be documented by an operator developer and ideally be stored in a *code repository* with APIs allowing programmable inspection. A composer who builds an operator graph, whether a human or a program, may use the code repository as a library to import the operator modules into a composed graph. The repository thus enables code-based operator reuse. On the other hand, any instantiated operators may also register a name advertisement in a *name directory* and act as a virtual sensor. In a composed operator graph, the sensors are specified as name queries that are resolved by the directory to select from existing named sensors or operators. The name directory thus enables instance-based operator reuse.

3 Service platform

Given logical operator-graph specifications, a CFN provides a platform to connect the distributed sensors and applications and to execute the operators. In this section, we describe Solar's service platform that is based on a self-managed overlay network. The software package of Solar is written in Java.

3.1 Planetary overlay

Solar consists of a set of functionally equivalent hosts, named Planets, which peer together to form a service overlay using a distributed hash table (DHT) based peer-to-peer protocol (specifically Pastry [27]). As shown in Figure 3, a sensor S may connect to any Planet to advertise its availability and an application A may connect to any Planet to select sensors and aggregate their data streams with customized operators. The Planets cooperatively provide several common services that are used to manage the operators and the data flows. We discuss the relevant services in Section 4.

Each Planet is a peer node in the overlay network and has a unique key randomly chosen from a large numeric space. The Pastry's peer-to-peer routing substrate provides a transport method to send a message to a destination identified by key, instead of an IP address. The message will be routed to the Planet whose key is currently closest to the destination key in $O(log(n))$ hops.

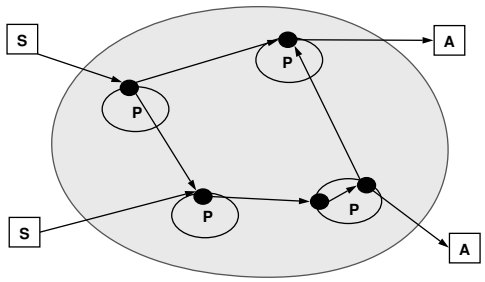

Figure 3. Solar consists of a set of functionally equivalent nodes, named Planets (denoted P), which peer together to form a service overlay using a P2P routing protocol. Sources S and applications A may connect to any Planet. The filled circles are operators and the arrows represent data flow.

Many Solar services take advantage of this hashtable-like interface. The overlay network is self-organizing and self-repairing and simulations on realistic network topologies show that: 1) the delay stretch compared with direct IP transport is usually below two, and 2) the paths for messages sent to the same key from nearby nodes in the underlying network converge quickly after a small number of hops [6].

3.2 Planet architecture

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

We consider each functionality mentioned above a *service*, which runs on every Planet. The core of the Planet is a service manager, which contains a set of services that interact with each other to manage operators and route events. We show the architectural diagram of a Planet in Figure 4.

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

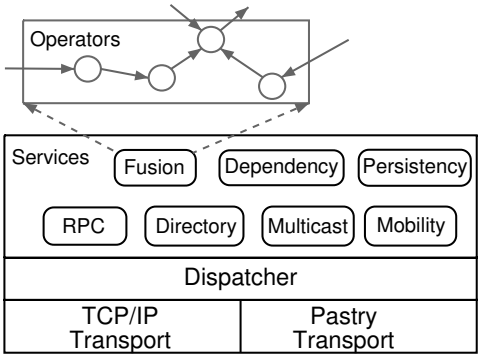

Figure 4. The diagram of Planet architecture, which exposes two kinds of transport interface: normal socket communication and peer-to-peer routing. The dispatcher multiplexes received messages to other Solar services.

Planet. A service may also get a handle of another service on the same Planet and directly invoke its local interface methods.

An application, which is a Solar client, sends a request to the *fusion* service on the Planet the application connects to. The fusion service may ask the local directory service to discover 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 what name advertisements, while the directory users do not need to know the internals.

We believe that our architecture based on service-level modules is a simple but powerful abstraction to build distributed systems, particularly overlay-based systems. The idea builds on object-oriented modules and allows easy upgrading and swapping of service implementations as long as the service interface does not change. For instance, we could easily add a caching capability to the directory service to improve query performance. The hidden intra-service communication, either through TCP/IP or DHT transport, is important to ensure low service coupling. The local access to remote service through a downloaded proxy in Jini shares a similar idea [32].

Our architectural approach also allows us to extract some common functionalities from several services and consider them primitive services. For instance, both the RPC service, simulating blocking remote calls, and the Heartbeat service, sending certain messages to a remote peer periodically (not shown in Figure 4), are used by several other services. Thus we can improve the Planet efficiency by reducing redundant resource usage.

The set of services on a Planet is configurable and it is easy to add new functionalities to a Planetary network by simply adding another service to each Planet. For instance, we are building a Web proxy service on the Planet. Besides serving HTTP pages for clients, the proxies are all federated with the Planetary network so that they could work together on content caching, prefetching, and re-directing the requests [34]. The new service may simply reuse the existing services, such as directory, multicast, and fusion.

The fusion service on a Planet manages local operators and schedules their execution. It uses the directory service to select sensors or other named operators desired by applications. Events sent to push channels are disseminated through an application-level multicast facility for improved scalability. The dispatch service registers callback with both transports to receive messages, which contains multiplex header indicating the destination service. The mobility service tracks clients (sensors or applications), which may detach from a Planet and reattach to another one. The dependency service is used to monitor and recover operators in case of Planet failures. We present these services in next section.

4 Solar services

In this section we discuss several Solar services that manage the operator graphs in Planetary network. Although the capacity of a given set of Planets is limited, it is easy to increase the capacity by adding more Planets while Solar's services are designed to harness the new Planets automatically.

4.1 Client mobility

Solar employs a client-service model in which a client may connect to any Planet, as the client's proxy, for service. A client hosts one or more *endpoints*, namely the sensors or applications. A mobile client may experience temporary disconnections due to weak links or mobility hand-offs. During disconnection, a client may roam and change its network address (network mobility), and it may or may not choose the original Planet when it reconnects (host mobility). A client may also voluntarily decide to switch Planets if it finds a "better" one, such as one that is closer or has a lighter load. The proxy may also make its own decision to disconnect a client, if the proxy is about to shut down or is too crowded, and force the client to select a different Planet.

Thus the client and its proxy Planet engage in a protocol maintaining state about each other. A client (thus the endpoints it hosts) may appear in three states to the proxy: *attached*, *detached*, or *departed*. State transitions from *attached* to *detached* are triggered either by explicit requests or by missing several heartbeat signals. If the client has been detached longer than a config-

urable threshold, the proxy assumes the client has departed (and will not re-attach to this Planet). The proxy appears to the client in two states: *attached* or *detached*; transitions are managed in a way similar to the client state. The proxy is responsible for relaying events and queries for attached clients, and buffering the events for detached clients as discussed in next subsection.

4.2 Data dissemination

A sensor or operator may publish an event stream through a specific outport, which may have multiple push channels connected. In other words, an event stream could have many subscribers. Solar uses Pastry's application-level multicast facility to disseminate the events for scalability reasons. Each push-based outport has a unique key and the events coming out of that port are routed to the Planet that is responsible for that key. That Planet is the root of the multicast tree, which is built incrementally as subscribers arrive on any Planet [7].

Conceptually there is a FIFO buffer on every node of the data path from sender to receiver. The data in a buffer may accumulate due to network congestion, temporarily client disconnection, or slow receiver consumption. The stream aggregation involved in data fusion and the constraints of devices hosting mobile applications all make buffers more vulnerable to overflow. Unlike traditional approaches that arbitrarily drop data or pause the sender, Solar allows the push channel to be associated with a data-reduction policy. The policy states how to drop or summarize portion of the data streams. The flexibility of pushing application-specific policies into the CFN allows loss-tolerant applications to trade in-order reliable delivery with timely transmission that respects application semantics.

Solar's multicast service registers with the mobility service, to be notified when clients detach or attach. As a receiver moves from one Planet to another, it receives old events buffered at its old proxy before it joins the new proxy. We present the details of Solar's data dissemination service, including the data-reduction approach, in another paper [10].

4.3 Naming and discovery

Every sensor registers a *name advertisement* in Solar's directory service, and optionally the deployed operators may also register an advertisement. Applications then use *name queries* to select desired sources, composing them with an operator graph that produces the context needed by the application. A name advertisement or a name query contains a set of attributes, each of which is a tag-value pair. The value of an attribute could be another set of attributes, thus forming an attribute hierarchy. A path from the root to a leaf is called a *name strand*. A name matches a query if the query's strands are a subset of the name's strands. Solar stores the name on these Planets that are responsible for the keys hashed from the name strands, in a way similar to INS/Twine [2]. The expressiveness of name queries may be enhanced by a customized function that is evaluated over matched names to select one of the satisfying sources.

To cope with the dynamics of a ubicomp environment, Solar exposes two interfaces in addition to those in traditional directory services. First, it supports continuous queries so the directory will actively notify the querying entity about changes in name space regarding to the query. Second, Solar supports context-sensitive names and queries, in which the attribute values may be defined by context computed from another operator graph. By combining these two features, an operator graph can be automatically reconfigured according to context changes. For instance, an instant messaging client on a mobile device may automatically highlight the names of buddies nearby. More information about Solar's directory service is in an earlier paper [9].

4.4 Dependency management

If an operator fails, its hosting Planet can simply restart it. It is also necessary for Solar to detect and recover failed clients or Planets. Solar provides a general dependency management service for distributed components. It contains two functionalities: monitoring and recovery, and tracking the state of components on behalf of their dependents. Each component of the operator graph registers a *configuration*, containing information on how to restart it in case it fails. Solar employs a self-monitoring protocol in which every Planet chooses a randomly selected peer to monitor its liveness. If a Planet fails, all its hosted operators are restarted at another Planet. If a client fails, Solar may try to restart it according to its configuration. Note that the monitoring protocol is bi-directional, namely, the Planet will re-select its monitoring peer if the peer itself fails.

Each component may also specify a policy for each dependency relation, stating how to handle the case when the component it depends on has failed or moved. The dependency could be instance-based, so the dependent waits until the upstream operators restart. The dependency could also be name-based, so the dependent may switch to a different source if the evaluation of the name query changes. The dependency policies are managed at the Planet responsible for the dependent's key. When an operator has no dependents any more, perhaps because its application has departed, it is garbage collected to reclaim the resources. The dependency service and the self-managed routing subsystem allow Solar to tolerate many common failure modes. We present the

details of Solar's dependency management in another paper [11].

4.5 Load balancing

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

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

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

5 Programming support

Solar provides an XML-based composition language for composers (typically applications) to specify operator graphs, identifying context-fusion operators and name queries that select sources. The language allows components to be connected using pull or push channels. In addition, the composer may associate a caching policy with a pull channel, or a data-reduction policy with a push channel. The caching policy allows quick return of queries while the data-reduction policy states how to drop or summarize an event queue.

The operator developers, on the other hand, construct operators by extending a suite of base classes. Our current operator library contains a variety of filters, transformers, and aggregators. We are also interested in providing operators that capture advanced machine-learning algorithms.

Solar does not restrict the syntax of its events, as long as it follows a hierarchical attribute structure. Thus it is easy to incorporate XML events into the Solar system. An advantage of using XML events is that we can use XSLT or XQuery to program the operators. Although they cannot fulfill arbitrary data-fusion functions, they are easy to write and quick to deploy for simple data processing [35]. It is also possible to provide a domain-specific language that specify the context-fusion logic, which Solar then parses into several operators for deployment [8]. IBM's iQL is an example of such a language [12].

6 Applications

In this section we overview some applications we built with Solar for two types of environments, smart spaces and emergency response. We then discuss the lessons we learned from our experience.

6.1 Smart space

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 has been used by ubicomp students to develop several applications, such as a wall-mounted Web portal that shows content personalized for the approaching user, and a "smart reminder" that alerts its user about events in advance at a time that depends on her current location and the location of next appointment [23]. Recently we instrumented an office by tying motion and pressure sensors to the chairs, then aggregating their outputs to detect whether a meeting is in progress, based on which a controller may automatically route incoming calls to voice mail without interrupting the meetings [33].

Our campus is covered by a wireless network with more than 550 WiFi access points, each of which is configured to forward their syslog messages to a workstation in our lab. The messages indicate events such as a client associating, roaming, or leaving the network, and are published by a Solar sensor. By subscribing to this sensor, applications may know the approximate location of the wireless clients. We developed a location-dependent "graffiti" application that allows its user to leave text or graphical notes at the current location, or to see others' notes left nearby. We also implemented a Web proxy that automatically pushes location-dependent information to a wireless client browser [10].

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

6.2 Emergency response

Solar is currently being deployed for a new project, based on an earlier system called the Automated Remote Triage and Emergency Management Information System.[3] These emergency-response applications use a large number of data streams from environmental sensors, physiological sensors (attached to victims and responders), and human observer inputs. Command and control applications use Solar to compute high-level and customized contextual information for the decision makers from different participating organizations. The goal of Solar in this project is to provide situational awareness at all levels of the incident command hierarchy.

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

6.3 Discussion

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

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

The first prototype of Solar was a pure event-driven system, in which each operator is a passive event handler. While this approach is natural for many physical sensors, we found it difficult to incorporate other types of online sources, such as calendar information. The query interface over pull channels in the current version meets this need. Another drawback was a lack of built-in timeout facility, so an operator could not gain control unless it receives an event: some fusion logic needs to react to a lack of events as much as it reacts to new events.

While Solar in a smart space is deployed on a LAN, the emergency response applications require Solar to be deployed in a WAN environment. It is known that the P2P routing protocols, such as the one used by Solar, typically suffer from a large number of simultaneous node arrival and departure in a WAN environment. In our case, however, the Planets are running on controlled infrastructure nodes instead of on random hosts provided by arbitrary users. Our dependency management service makes Solar resilient to common failures in WAN and our data-reduction technique handles the buffer overflow caused by network congestions.

7 Related work

Schilit proposed one of the first architectures to support context-aware applications [28], which uses an ActiveMap service to disseminate location-related information. Since then several researchers have designed systematic approaches for context-aware computing. The Context Toolkit facilitates programming by wrapping sensors with a well-defined widget interface and predefined aggregators may combine several widget outputs to produce context information [14]. The Context Fabric takes a query-based approach, which automatically constructs a data path to answer a context query [17]. The CIS, used by Aura, provides a database approach where contextual queries are encoded in a SQL-like language [19]. Nexus builds federated spatial world models based on which applications may reason about location context [16]. These systems have a focus on the interface between applications and system with the goal to reduce the programming complexity. They, however, lack studies on system issues such as scalability, mobility, and reliability, which are inherent challenges in ubicomp environments. Solar addresses these issues with its overlay-based service platform and a set of unique services, thus facilitating both the development and deployment of context-aware applications in large-scale scenarios.

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

The EventHeap used by iRoom project employs a tuplespace model, which aims to decouple the data producer and consumer [18]. This loosely-coupled coordination model significantly reduces the component interdependency and allows easy recovery from crashes. The simple interface of a tuplespace, tuple retrieval based on pattern matching, limits the expressiveness of data processing. There is no direct support for context fusion. It is also not clear how a tuplespace may scale.

In industry, the ContextSphere project from IBM research shares many similarities with Solar. Its precedent, iQueue, is also a distributed system that allows application to specify data composers like Solar operators [13]. The group also developed an expressive domain-specific language, iQL, for the composer specification [12]. We believe many of Solar's services could be easily used by ContextSphere to manage the composers, such as our context-sensitive directory, policy-driven flow control, and dependency management. On the other hand, Solar may take advantage of iQL as an operator language, which is more intuitive than Java while more expressive than XSLT or XQuery.

Data aggregation is a useful technique inside sensor networks to reduce unnecessary transmission [3, 15, 22]. 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. These systems, however, are complementary to Solar since the aggregated results coming out of a sensor network could supply one data stream as a Solar source.

The problem of achieving new composite service by composing existing autonomous Web services is generating considerable interest in recent years. Researchers have been working in composition languages [21], toolkits [24], supporting systems [5, 4, 26], and load balancing and stability algorithms [25]. If we consider sensors in Solar as output-only services, we could also enhance Solar's composition model with previous results, such as the more powerful composition language, the rule-based composition engine, and algorithms for preserved quality of composed service.

8 Summary

In this paper we motivate and define the concept of a Context Fusion Network (CFN), to support context-aware mobile applications that need to aggregate data from distributed sensors. Our operator-graph model defines both push- and pull-based data communication and exposes a simple interface to facilitate operator composition. One advantage of a CFN is the promotion of reusability, both code and instance re-usability. Our prototype infrastructure, Solar, provides an overlay-based platform and various management services for a CFN. We design and implement the services to make Solar a flexible, scalable, mobility-aware, and self-managed system for a heterogeneous and volatile ubicomp environment. We demonstrate the effectiveness of Solar with some applications we built for smart spaces and emergency response; we evaluate the performance or effectiveness of many of these services in other papers [9, 11, 10]. We also discuss some lessons we learned from our experience.

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

References

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

[2] M. Balazinska, H. Balakrishnan, and D. 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.

[3] P. Bonnet, J. Gehrke, and P. Seshadri. Towards Sensor Database Systems. In *Proceedings of the Second International Conference on Mobile Data Management*, pages 3–14, Hong Kong, China, January 2001.

[4] F. Casati, S. Ilnicki, L.-J. Jin, V. Krishnamoorthy, and M.-C. Shan. Adaptive and Dynamic Service Composition in eFlow. Technical Report HPL-2000-39, HP Labs, 2000.

[5] F. Casati, S. Ilnicki, L.-J. Jin, V. Krishnamoorthy, and M.-C. Shan. eFlow: a Platform for Developing and Managing Composite e-Services. Technical Report HPL-2000-36, HP Labs, 2000.

[6] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Topology-aware routing in structured peer-to-peer overlay networks. Technical Report MSR-TR-2002-82, Microsoft Research, 2002.

[7] M. Castro, M. 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.

[8] G. Chen and D. 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.

[9] G. Chen and D. Kotz. Context-aware resource discovery. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, March 2003.

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

[11] G. Chen and D. Kotz. Dependency management in distributed settings (poster abstract). In *Proceedings of the First International Conference on Autonomic Computing*, New York City, NY, May 2004.

[12] N. H. Cohen, H. Lei, P. Castro, J. S. Davis II, and A. 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.

[13] N. H. Cohen, A. Purakayastha, L. Wong, and D. 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.

[14] A. K. Dey, G. D. Abowd, and D. 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.

[15] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan. Building efficient wireless sensor networks with low-level naming. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, pages 146–159, Banff, Alberta, Canada, 2001.

[16] F. Hohl, U. Kubach, A. Leonhardi, K. Rothermel, and M. Schwehm. Next century challenges: Nexus – an open global infrastructure for spatial-aware applications. In *Proceedings of the Fifth Annual International Conference on Mobile Computing and Networking*, pages 249–255, Seattle, Washington, United States, 1999.

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

[18] B. Johanson and A. 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.

[19] G. Judd and P. Steenkiste. Providing Contextual Information to Pervasive Computing Applications. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, pages 133–142, Fort Worth,Texas, March 2003.

[20] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[21] F. Leymann. Web Services Flow Language (WSFL 1.0).

[22] S. Madden, M. J. Franklin, and J. 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.

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

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

[25] B. Raman and R. 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.

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

[27] A. Rowstron and P. 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.

[28] W. N. Schilit. *A System Architecture for Context-Aware Mobile Computing*. PhD thesis, Columbia University, May 1995.

[29] A. Schmidt, K. A. Aidoo, A. Takaluoma, U. Tuomela, K. V. Laerhoven, and W. V. de Velde. Advanced Interaction in Context. In *Proceedings of the First International Symposium on Handheld and Ubiquitous Computing*, pages 89–101, Karlsruhe, Germany, September 1999.

[30] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[31] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the 2002 International Conference on Compiler Construction*, pages 179–196, Grenoble, France, Apr. 2002.

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

[33] J. Wang, G. Chen, and D. 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*, Boston, MA, June 2004.

[34] L. Wang, V. Pai, and L. Peterson. The Effectiveness of Request Redirection on CDN Robustness. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, December 2002.

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