Context-Sensitive Resource Discovery

Guanling Chen and David Kotz
Department of Computer Science, Dartmouth College
{glchen, dfk}@cs.dartmouth.edu

Abstract

This paper presents the "Solar" system framework that allows resources to advertise context-sensitive names and for applications to make context-sensitive name queries. The heart of our framework is a small specification language that allows composition of "context-processing operators" to calculate the desired context. Resources use the framework to register names, and applications use the framework to look up context-sensitive name descriptions. The back-end system executes these operators and constantly updates the context values, adjusting advertised names and informing applications about changes. We report experimental results from a prototype, using a modified version of the Intentional Naming System (INS) as the core directory service.

1 Introduction

Much of the technology necessary to realize Mark Weiser's vision of ubiquitous computing [16] is now available. Small portable devices, and the wireless networks to support them, are pervasive. The resulting pervasive-computing environments are crowded, heterogeneous, and always changing. To succeed without distracting the user, pervasive-computing applications must be aware of the context in which they execute, and automatically adapt as that context changes. An important component is the ability to discover resources (services and information sources) relevant to the application's context. If each resource is given a name, resource discovery amounts to a query in the name space.

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

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

These scenarios place several requirements on the naming service. It must be flexible, so names can characterize the resource and so queries can express the desired characteristics; it must be scalable, to handle many names; it must be fast, to support frequent name updates; and it must be responsive, to quickly notify applications about changes to the set of matches for their persistent query. Existing work has partially solved this problem. For example, the Intentional Naming System (INS) is designed to handle frequent name updates in a mobile environment [1].

These scenarios also place several requirements on the resources and applications. Resources must actively track their context so that they may update their name. 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).

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.

In this paper we present a name service, part of the Solar framework for context-aware applications, that helps resources to advertise context-sensitive names and allows applications to make persistent, context-sensitive queries. We show how our approach allows applications to flexibly define the desired context aggregation, off-loads the task of updating and monitoring the name space, and permits a decentralized and scalable implementation.

This work makes two contributions: a) an extension to the attribute-based name concept that supports context-sensitive names and context-sensitive name queries, and b) a distributed infrastructure that efficiently supports immediate and persistent queries in such a name space. Our infrastructure is designed to leverage existing distributed directory services; for our prototype we chose the Intentional Naming System (INS) but any other service providing a registration and look-up interface would suffice.

This work has two primary limitations. First, since one of our goals is to reduce load on thin client devices and on the low-bandwidth networks that serve them, we embed the service in the infrastructure; thus our techniques are not readily applicable to an infrastructure-free (ad-hoc) environment. Second, security and privacy issues are beyond the scope of this paper; interested readers may find more information in another paper [14].

2 The specification language

Solar's language serves three purposes: 1) resources may advertise their existence by registering a name, 2) applications may discover resources by querying a name, and 3) context-aware applications may indicate how to derive their desired context information from that provided by existing resources. A *resource* is an entity that wishes to be discovered by applications. It may be a *service*, willing to respond to requests from applications; for example, a service might compute the estimated travel time when provided two street addresses. Or, it may be a *publisher* of context information. For simplicity, we focus on publishers, and define them more thoroughly below. First, consider the structure of a name.

A *name specification* is a set of *attributes*, each of which has a tag string and a value string. Resources use name specifications to define their name; applications use name specifications to query for matching names. A query matches a name if all attributes of the query are present in the name, and the values of corresponding attributes are equal. (We plan to consider more complex matches, such as inequalities, in future versions of the system.) For example, the name

```
[sensor="camera", class="color",
 room="215", building="Sudikoff"]
```

has four attributes; for the first attribute, the tag is "sensor" and the value is "camera". The query

```
[sensor="camera", building="Sudikoff"]
```

matches the above name, and the name of other cameras that may be in Sudikoff. A *context-sensitive* name specification may be used either as a name or as a name query. In the specification

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

the value of the "room" attribute is defined by context information derived from an *event stream* called $alice-locator. Every context-sensitive name specification must be accompanied by a *graph specification* that defines the desired event streams. We next describe the event-stream abstraction, then the language of graph specifications.

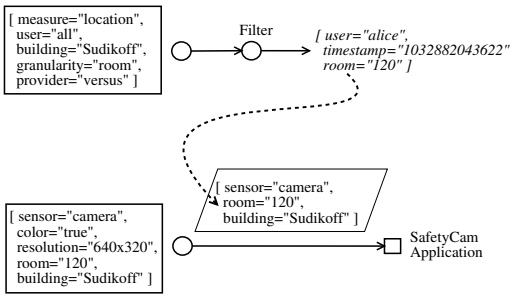
Event streams and operator graphs. Context information is often derived by post-processing and aggregating the output of several sources. *Sources* are sensors that produce raw context information. Each unit of context information published by a source is an *event*, structured as a set of attributes (tag/value pairs). Over time, the output of a source is an *event stream*. *Operators* aggregate context information by subscribing to one or more event streams and publishing another event stream. Thus, all sources and operators are event *publishers*. Since all operators conform to the same publish and subscribe interfaces, they can be stacked recursively to form a directed *graph* through which raw data flows and is processed into context as output.

Figure 1(a) demonstrates these concepts using the nursing-home example. At top left, a location source advertises a static (not context-sensitive) name and publishes location events about all residents. Its subscriber, a filter operator, discards events not pertaining to Alice. The event (in italics) indicates that Alice is in room 120. At lower right, the SafetyCam application uses the context-sensitive name specification

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

to identify and subscribe to a camera source near Alice. If $alice-locator refers to the event stream produced by the filter, then the arrival of the event resolves $alice-locator:room to "120", as shown, which matches the name advertised by the camera source in the lower left.

Graph specifications. To arrange the context-sensitive subscription depicted in Figure 1(a), SafetyCam uses the graph specification shown in Figure 1(b). The graph specification contains two parts, `define` and `load`. The `load` section provides the URL and



(a) A nursing-home example shows the sources (locator and camera), operator (filter), names (in rectangles), event (in italics), and the use of the output from an operator graph to select a name for the SafetyCam's subscription.

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

(b) Example graph specification to calculate Alice's current location, and how it is used to define a context-sensitive subscription.

Figure 1. A camera example with corresponding graph and name specifications.

class name of the Java classes for any non-standard operators used in the definition section, here @userFilter. The `define` section contains a sequence of statements, each of which defines an operator. Each statement defines the subscriptions for its operator, and together the statements determine the graph structure.

The first statement defines an operator that subscribes to any one of the publishers whose name matches the specification in brackets. The statement assigns the variable name $locator to that operator's event stream. The syntax "@func [name spec]" identifies the desired subscriptions, using the function @func to select among the set of names matching the name specification. Selection functions @any and @all are built-in; the user may also define custom selection functions.

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

```
// application subscribes to the root operator  of
// a specified operator graph.
subscribe(String graph-spec);

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

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

// application launches an operator graph (defined in graph-spec) and
// assigns a static name to the root operator.
launch(String graph-spec, String name-spec);
```

Figure 2. The set of Solar API methods.

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

The SafetyCam application may receive the stream of photo events by simply providing this graph specification (see the *subscribe* interface in Figure 2); since $cameras is the root of the resulting operator graph, the SafetyCam receives its events. The Solar system deploys the operators, arranges the subscriptions, and actively monitors the context-sensitive name specifications to adjust subscriptions as necessary. Notice that Figure 1(a) is a conceptual diagram and does not contain the relay and merge operators defined in Figure 1(b); the purpose for these operators will become clear below where we discuss operator-graph deployment.

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

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

as a query. The application could ask once to receive the current list of matching names, or it could register a *persistent query*, and be notified any time the set of matching names changes (see the *query* interface in Figure 2).

In another situation, suppose Alice carries a camera that provides a photo-capture source. Her camera source provides the same graph specification and the

above name specification as an advertisement; the result is that the camera has a context-sensitive name advertisement (see the *advertise* interface in Figure 2).

3 System architecture

Solar is a framework for collecting, processing, and disseminating context information to context-aware applications. A Solar system consists of a collection of peers, called Planets. Each Planet typically resides on a fixed host with reasonable computational resources and a consistent network connection. The Planets are collectively responsible for the execution of operators, the deployment of operator graphs, the dissemination of events, and the implementation of the name service. Resources and applications obtain Solar's services by connecting to any Planet. Further details about Solar are available in our earlier papers [5, 6]; in this paper, we focus on the naming system. [1]

A resource, such as a context source, may contact any Planet with a request to advertise a name, providing a graph specification if the name is context-sensitive (see the *advertise* interface in Figure 2). The Planet creates an internal object as a proxy for the source, and the proxy internally registers that name. Subscribers naming that source actually subscribe to the proxy, which forwards events from the source to all subscribers.

An application may contact any Planet with a name query, providing a graph specification if the name is context-sensitive (see the *query* interface in Figure 2). The Planet creates a proxy object to service the query. An application may also contact any Planet with a graph specification if it wishes to subscribe to a context event stream (see *subscribe* interface in Figure 2). The Planet deploys the desired operator graph and creates a proxy object to serve the application, which subscribes to the operator graph and forwards any received events.

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

Internally, we build Solar's name service on top of a generic directory service, using Solar operators to obtain, process, and monitor the necessary context information. This layered approach allows us to leverage existing research on (and implementations of) scalable, distributed, and flexible directory services.

In particular, we used the Intentional Naming System (INS) to implement the directory service [1]. To Solar we add INS resolvers, as shown in Figure 3. After summarizing INS, and our extensions to INS, we describe how Planets use INS for name queries and name advertisements.

[1] An early Solar paper [4] describes a different name system; this paper represents an entirely new approach to naming.

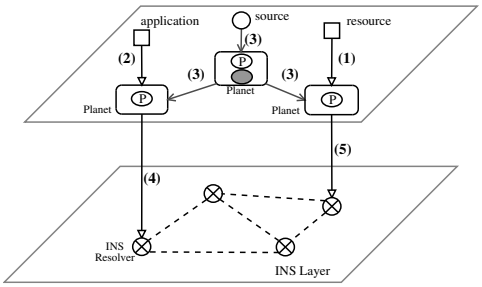


Figure 3. System architecture with both the Solar and INS layers. The shaded circle is an operator that may process events from the source to provide context for an application's name query or a resource's name advertisement. The ovals marked with P are proxies representing connected Planet clients. Arrows represent (1) context-sensitive advertisement; (2) context-sensitive query; (3) contextual events; (4) actual query based on current context; (5) actual name advertisement based on current context.

3.1 Extending INS

INS is a resource-discovery and communication system [1]. In conventional networks, a name service resolves names to addresses, then routers route messages to destination addresses. In INS, a distributed collection of *resolvers* form an overlay network that routes messages to destination names. Thus, INS combines name resolution and message routing into a single abstraction.

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

An application may send an intentional *anycast* message to a given name *pattern*; the resolvers route the message to any destination with a matching name. [Names and patterns are sets of hierarchical attribute-value pairs; a pattern matches a particular name if all attributes of the pattern are present in the name, and the values of corresponding attributes are equal.] An application may also send an intentional *multicast* message to a given pattern; the resolvers route the message to *all*

destinations with matching names.

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

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

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

3.2 Using INS

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

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

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

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

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

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

3.3 Example

In Figure 4 we show the deployment of the two variants of the camera example from Figure 1(b). Case (a) depicts a camera source with a location-dependent name, and case (b) depicts applications with location-dependent subscriptions to camera sources.

In both cases, the $locator operator's monitor finds all the locators in Sudikoff and picks any one of them (using selection function @any), so that the $locator operator subscribes to that locator source. (The locator sources are not shown in the figure). If for some reason the locator source changes its name and no longer matches the pattern, the monitor of its proxy sends a meta-event to its subscribers including $locator. The $locator monitor receives the meta-event and selects another locator among matching names. The events produced by $locator are filtered to produce $alice-locator, then used by the proxy's monitor to adjust the name advertisement

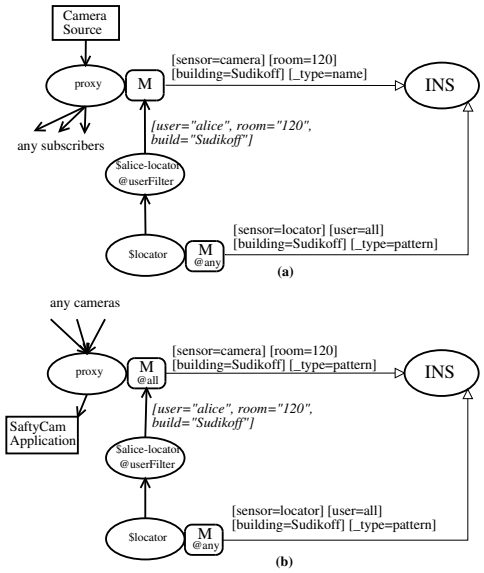


Figure 4. (a) A deployment of the context-sensitive name advertisement of the mobile camera carried by Alice. (b) A deployment of the context-sensitive subscription to all cameras co-located with Alice.

(case a) or subscription (case b).

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

3.4 The Planets

A Solar system consists of a collection of peer Planets, typically on separate but well-connected hosts in a distributed network. The Planets, written entirely in Java, form the execution environment for proxies, operators, and their attached monitors.

When a client connects to a Planet of its choice and submits its request, the Planet installs a proxy object for the client. The proxy relays messages and events between the client socket connection and the operators in the Solar system. The Planet parses the name and graph specifications in the client's request, creating operator and monitor instances according to the graph specification. The Planet arranges any static subscriptions between the operators, and monitors arrange an operator's context-sensitive subscriptions.

For each operator, the Planet maintains a list of subscribers and a queue of input events. A Planet thread passes input events one at a time, in the order received, to the operator's `handleEvent` method. That method may `publish` events, which are delivered to the input queue of each local subscriber, and to a network queue for subscribers on other Planets.

The Planets of a Solar system connect directly to each other, as needed, forming an overlay network for event dissemination. The Planet uses a queue and a thread to control the flow of events into a socket, and a thread to receive and distribute events at the other end of a socket. If more than one subscriber to a publisher reside on the same Planet, only one copy of published event is delivered to the subscribers' Planet. Further event-dissemination optimization, including batching and compression, remains future work.

3.5 Generalizing the framework

Solar is the backend system that implements the context-sensitive name service, and it also uses this name service to allow Solar applications to find and subscribe to Solar publishers. Thus the above discussion mainly focuses on named publishers (whether sources or operators) and applications as subscribers. Our framework, however, can also support context-sensitive names for other types of resources and applications. Services, for example, may ask Solar to advertise a context-sensitive name, using the same interface as for named publishers. Clients may ask Solar to provide a list of services, once or persistently, matching a static or context-sensitive name query. After a client locates a service, client-server communication may take any path, such as a socket or remote-method invocation (RMI), outside the Solar system.

4 Experiments and evaluation

We set out to measure the name-update performance of Solar's naming service. To obtain realistic events for our experiments, for 12 days we recorded the sensor data from a location system installed in our building.[2] For our experiments we created a "replay source" that read this trace and published each sensor reading as an event, and used it to drive our test clients.

For our experiments we used a set of fixed hosts (FH)[3] with 100 Mbps duplex connections. The average network latency among them was 0.37 ms. We also used two mobile hosts (MH): a Linux laptop[4] and a pen-based

[2] An IR-based badge-tracking system provided by Versus Technologies, Inc., http://www.versustech.com/.

[3] Dell GX260, 2.0 GHz Pentium 4, 256 MB RAM, running RedHat Linux 2.4.18-openmosix.

[4] Gateway Solo 3400, 450 MHz Intel Pentium 2, 128 MB RAM, running RedHat Linux 2.4.18-14.

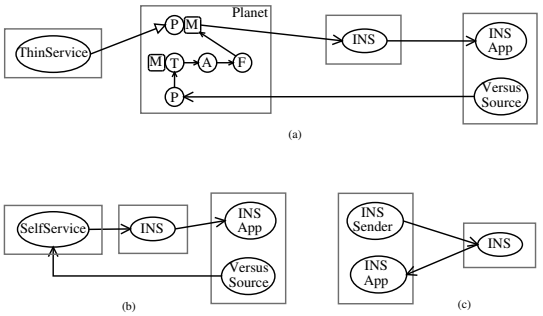


Figure 5. The setup for the latency tests (rectangle represents a FH or MH, P as proxy, and M as monitor): (a) ThinService sends a context-sensitive name-advertisement request to the Planet; (b) SelfService manages context-sensitivity by itself; and (c) a base case for INS-only latency, no event processing is involved.

Windows tablet.[5] The mobile hosts were connected via 802.11b through a dedicated access point with a clear 11 Mbps channel (distinct from channels used by any other nearby access point). The average network latency from a FH to the tablet was 2.9 ms and to the laptop was 2.7 ms. We used Sun Microsystem's Java runtime (v1.4.1-hotspot) on all platforms.

4.1 Latency measurements

Consider the task of advertising a location-sensitive name, based on the Versus location data. We compare an approach with Solar and an approach without Solar (see Figure 5).

In the Solar approach (Figure 5a), we set up an operator graph to provide the desired event stream, using our Versus replay source, a transformation operator (T) that converted badge and sensor numbers to symbolic names, an aggregation operator (A) that remembered the current location of each badge and produced an event whenever a badge changed location, and a filter operator (F) that removed all events except those about a particular person. We created a dummy "ThinService" that submitted this operator graph with a simple context-sensitive name specification.

In the other approach (Figure 5b), a "SelfService" application received all the events directly from the Versus source and did all the transformation and filtering internally. Whenever it detected a location change, it announced a new name to INS and sent a name-update multicast message to all matching patterns. We used a standalone INS client application, which registered a simple pattern to receive these name-update messages.

For comparison, we also measured the INS-only configuration in Figure 5c. An INS client "sender" periodically changed its announced name by announcing it to an INS resolver. An INS client application received all the name-update messages and calculated the difference. This configuration allowed us to isolate the overhead of event handling of the other two cases.

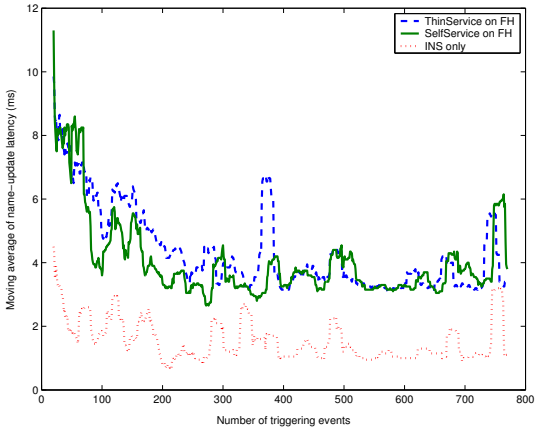
The replay source published many events, but only a few ultimately caused any change in the advertised name. We call such events "triggering events." For our experiments, we used a reduced trace that contained only the triggering events, since our goal was to measure the latency between the moment our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

Figure 6(a) shows the latencies for the three configurations of Figure 5, using only fixed hosts. The purpose of this test is to demonstrate the overhead of our approach, and to isolate the effects of the INS core. The x axis represents the progress of time, marked by the count of triggering events. Each data point is the moving average of the preceding 20 triggering events. ThinService and SelfService had similar performance because the bulk of their work, aggregating context information and updating INS, was essentially the same and executed in a similar environment (by the SelfService on an FH or by operators in a Planet on an FH). In general, ThinService had more overhead because it partitioned the work into three operators and connected them with queues, and there was some overhead for queue management. A substantial fraction of the latency, and its variation, was in INS. The bottom curve of Figure 6(a) shows the latency for the INS-only approach of Figure 5c. Clearly INS was a significant source of the variation seen in the other curves.

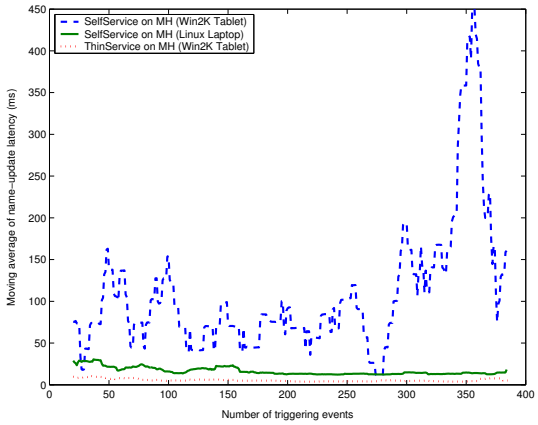
Note that latencies became smaller as the experiment progressed, due to the incremental compilation by the "Hotspot" JVM, but reached steady state after a few hundred events. There were, however, noticeable variations, caused by thread scheduling and temporary network congestion. The latency numbers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

Our second test demonstrates the advantage of our infrastructure approach. Figure 6(b) shows that the Thin-

[5] Fujitsu C-500, 500 MHz Celeron, 128 MB RAM, running Windows 2000.



(a) Latency comparison with services on fixed hosts.



(b) Latency comparison with services on mobile hosts.

Figure 6. Results of latency tests.

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

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

4.2 Scalability analysis

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

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

The above measurements did not consider, however, the scalability of the directory service, in this case INS. In the next experiment we distributed 40 client applications over 20 FHs and varied each Versus source's publishing rate. The INS application co-located with each source calculated the name-update throughput. We show the results in Figure 7. Instead of reaching 870 name updates per second, the throughput peaked near 500. With 40 clients each pushing over 10 triggering events per second, the INS resolver was overloaded and some packets were lost (INS routes name-update messages using UDP packets). The green curve (with * marks) shows the loss rate. The red curve (with + marks), which is the sum of name-update throughput and loss rate, flattens near 910 indicating the maximum event-processing throughput of the Planet. The number is similar to (though a bit higher than) the conservative throughput

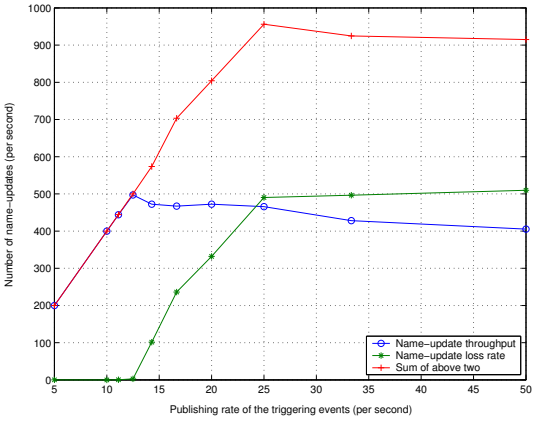


Figure 7. Results of scalability tests, measuring throughput of the Planets and the whole system under the load caused by an array of sources publishing triggering events at a fixed interval.

we measured above (about 870 events per second).

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

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

5 Related work

INS [1] unifies resource discovery (naming) and communication (message routing). Applications desiring context-sensitive names, however, must monitor the context themselves and re-advertise their name as needed. Furthermore, INS does not support persistent name queries. Solar uses an extended version of INS as the directory service at the core of its own context-sensitive name service. By offloading context processing and monitoring to Planets in the Solar infrastructure, our framework improves responsiveness and scalability.

INS/Twine achieves more scalability by partitioning the name space across resolvers by mapping names into numeric keys [2]. Solar could use INS/Twine as its core directory service, but would need a different mechanism to implement persistent queries.

A Location Information Server (LIS) [12] integrates location information and a resource directory, based on the X.500 directory service and the Light Weight Access Protocol (LDAP). The LIS shields the application from the methods for obtaining the location information and provides a set of APIs for query and event notification. While LIS provides some Solar features, including limited context translation and support for location-sensitive names, the capability of LIS is limited by the predefined configurations and rules. Solar provides a programmable interface to allow arbitrary definition of context-sensitive names. Solar also uses peer Planets to cooperatively service clients' requests and to disseminate contextual events, for better responsiveness and scalability.

A non-procedural language, iQL, can also specify the logic for composing pervasive data into context [7]. Their model supports both requested and triggered evaluation. For one composer, iQL allows the inputs to be continually rebound to appropriate data sources as the environment changes. The iQL language is powerful and expressive, with many language-level facilities for context aggregation. The language iQL complements Solar in two ways: iQL could be the programming language for individual operators, or iQL could be the high-level specification language the compiler could decompose into a graph specification used by Solar.

The Context Toolkit provides several abstractions to construct a context service [9]. It is a distributed architecture supporting context fusion and delivery. It uses a *widget* to wrap a sensor, through which the sensor can be queried about its state or activated. Applications can subscribe to pre-defined aggregators that compute commonly used context. Solar allows applications to dynamically insert operators into the system and to compose refined context that can be shared by other applications. The Context Toolkit provides a static attribute-based directory to allow its components to register a name so applications can find them. It does not support context-sensitive names or subscriptions, or persistent name queries.

There are numerous directory services and resource- and service-discovery systems. Jini [13] allows a client to locate a service and download its proxy interface, matching on the class of the proxy. The Service Location Protocol (SLP) focuses on the protocol for automatic discovery [10], Czerwinski et al. focus on expressiveness and security [8], and Castro et al. deals with

inter-domain service discovery [3]. DeapSpace [15] and Heidemann et al. [11] propose approaches for discovery in ad-hoc sensor networks. None of these systems, however, has explicit support for context-sensitive names or subscriptions.

6 Summary

In this paper we describe Solar's approach to support context-sensitive resource discovery. Solar encodes contextual information as attribute-based events, published by sources. Solar provides a specification language for clients to select the desired sources and to construct an operator graph to aggregate context, which may be used directly by the application, to define a name, or to make a persistent name query. Although our examples use only location context, Solar is capable of using any form of context information in context-sensitive names.

Solar uses a collection of "Planets" to cooperatively execute the operator graph. Planets monitor the context produced by that graph to update the name space and to notify interested applications about name changes. Context monitoring is recursive, since the operator graph itself can refer to context-sensitive source names.

In particular, Solar uses an extended version of the Intentional Naming System (INS) to implement the core directory service and persistent name queries. We measured the name-update latency and analyzed the name-update scalability. We conclude that Solar's approach is responsive because it off-loads context aggregation, dissemination, and monitoring into a scalable infrastructure of cooperating Planets, and is likely to be scalable.

References

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

[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. Springer-Verlag.

[3] B. P. Castro, P. C. Bisdikian, and M. Papadopouli. Locating application data across service discovery domains. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, pages 28–42, Rome, Italy, July 2001. ACM Press.

[4] G. Chen and D. Kotz. Solar: Towards a flexible and scalable data-fusion infrastructure for ubiquitous computing. In *UbiTools workshop at UbiComp 2001*, October 2001.

[5] 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. IEEE Computer Society Press, June 2002.

[6] G. Chen and D. Kotz. Solar: An open platform for context-aware mobile applications. In *Proceedings of the First International Conference on Pervasive Computing (Short paper)*, pages 41–47, June 2002. In an informal companion volume of short papers.

[7] 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. IEEE Computer Society Press.

[8] S. E. Czerwinski, B. Y. Zhao, T. D. Hodes, A. D. Joseph, and R. H. Katz. An architecture for a secure service discovery service. In *Proceedings of the 5th Annual International Conference on Mobile Computing and Networking*, pages 24–35, Seattle, WA, August 1999. ACM Press.

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

[10] E. Guttman. Service location protocol: Automatic discovery of IP network services. *Internet Computing*, July/August 1999.

[11] 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 Systems Principles*, pages 146–159, Chateau Lake Louise, Canada, October 2001. ACM Press.

[12] H. Maaß. Location-aware mobile applications based on directory services. In *Proceedings of the 3rd Annual International Conference on Mobile Computing and Networking*, pages 23–33, Budapest, Hungary, September 1997. ACM Press.

[13] S. Microsystems. Jini architecture specification, December 2001.

[14] K. Minami and D. Kotz. Controlling access to pervasive information in the "Solar"system. Technical Report TR2002-422, Dept. of Computer Science, Dartmouth College, February 2002.

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

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