Controlling access to pervasive information

Kazuhiro Minami and David Kotz

Dept. of Computer Science, Dartmouth College
Hanover, NH, USA 03755
{minami, dfk}@cs.dartmouth.edu

October 2002

Abstract

Pervasive-computing infrastructures necessarily collect a lot of context information to disseminate to their context-aware applications. Due to the personal or proprietary nature of much of this context information, however, the infrastructure must limit access to context information to authorized persons. In this paper we propose a new access-control mechanism for event-based context-distribution infrastructures. The core of our approach is based on a conservative information-flow model of access control, but users may express discretionary relaxation of the resulting access-control list (ACL) by specifying *relaxation functions*. This combination of automatic ACL derivation and user-specified ACL relaxation allows access control to be determined and enforced in a decentralized, distributed system with no central administrator or central policy maker. It also allows users to express their personal balance between functionality and privacy. Finally, our infrastructure allows access-control policies to depend on context-sensitive groups, allowing great flexibility. This paper describes our approach, our implementation, and initial performance measurements.

1 Introduction

Many mobile applications automatically adapt to the changing conditions in which they execute. These *context-aware* applications take account of information about the context, such as the location of the user and relevant devices, the presence of other people, light or sound conditions, or available network bandwidth. While the necessary sensors are increasingly available, particularly for location [HB01], we note that most *sensor data* must be processed into higher-level *context information* before use by applications. It is unreasonable to expect every application to work with the raw sensor data, and it is unscalable to expect a single "context server" to support the diversity of transformations and the scale of numerous applications and users.

We therefore need an infrastructure to aggregate and transform low-level sensor data for context-aware applications, while remaining flexible and scalable [CPT+01]. Many such systems use an event-flow model, in which sensor data are represented as events, and a graph of operators transform these event streams into the event streams desired by the applications. iQueue [CPWY02] and Gryphon [BKS+99] are two examples.

Any such infrastructure must allow many applications and many users to share sensor data and context information. On the other hand, the context often includes information considered "private" by many users, such as their location, or "proprietary" by organizations, such as the marks written on a shared whiteboard or the calendar of meetings for product teams. A context-information infrastructure must, therefore, control access to the information it disseminates. Although researchers often note the importance of privacy and security in context-aware computing [EHL01, Sat01, ST93, Lan01], there is surprisingly little literature about access control in context-information systems.

In this paper we propose an access-control mechanism for protecting context information in an infrastructure for collecting, processing, and disseminating context information. In such an infrastructure, context information is derived from numerous sources through a wide variety of operations. Applications and users can dynamically request new derivative information, and supply new operations. In such a dynamic environment it is unreasonable to expect a system administrator, or the users, to manually specify an access-control list for each event or event stream.

In our approach, each event is tagged with an access control list (ACL). Each operator in the event-flow graph automatically tags the events it produces based on the ACLs of its input events, on an optional restriction rule provided by the operator programmer, and on optional relaxation rules provided by users. In this way, operators that derive information from sensor data also derive the

necessary ACL.

Our approach is discretionary, in that users may use their discretion to explicitly relax ACLs to grant access to other users. Where there is no explicit relaxation, however, we derive a conservative default access-control policy using principles borrowed from information-flow theory [Den84]. By convention, sources publish information with narrow ACLs, so by default personal information remains private until users explicitly release it in an explicit and structured fashion.

Our system also allows ACLs to include *context-sensitive groups*; for example, a thermostat application may allow access to anyone currently in the room, by specifying a group whose membership changes as people come and go. Thus, the access-control policies are themselves context-aware.

In the next section, we briefly describe the event-distribution infrastructure we use as the base for our work. Section 3 describes our design objectives, and Section 4 explains the semantics of our access-control mechanism. In Section 5, we outline our implementation and performance results. We discuss related work in Section 6 and then summarize in Section 7.

2 Background

Our access-control mechanism supports event-based context-dissemination systems. We describe the Solar system [CK02] as one example, but our approach should apply to many similar architectures.

Solar is middleware that supports the collection, processing, and dissemination of context information for context-aware applications. In Solar, sensor data and context information are represented as *events* flowing from *sources* through *operators* (which filter, transform, or aggregate events into new events) to applications. Solar uses a publish/subscribe model of event flow. Sources and operators each publish a single stream of events; operators and applications may subscribe to many streams of events. Unlike some systems, each Solar publisher produces one event stream and each event stream has only one publisher. An operator subscribing to multiple event streams receives one event at a time, as if the streams were (arbitrarily) interleaved: events from a given publisher are delivered in the order they were published, but no ordering is defined between events from different publishers.

Operator graph. An operator is an object that subscribes to and processes one or more input event streams, and publishes another event stream. Since the inputs and output of an operator are all event streams, the operators can be connected recursively. When an application wants a flow of context information, it asks Solar to instantiate a *subscription tree* that describes the flow of events from a set of sources (leaves of the tree) through a set of operators to the application (the root of the tree). Solar makes it easy to re-use existing sub-trees, so that applications and users may share operators and their event streams. The overlapping subscription trees form a directed acyclic graph called the *operator graph*.

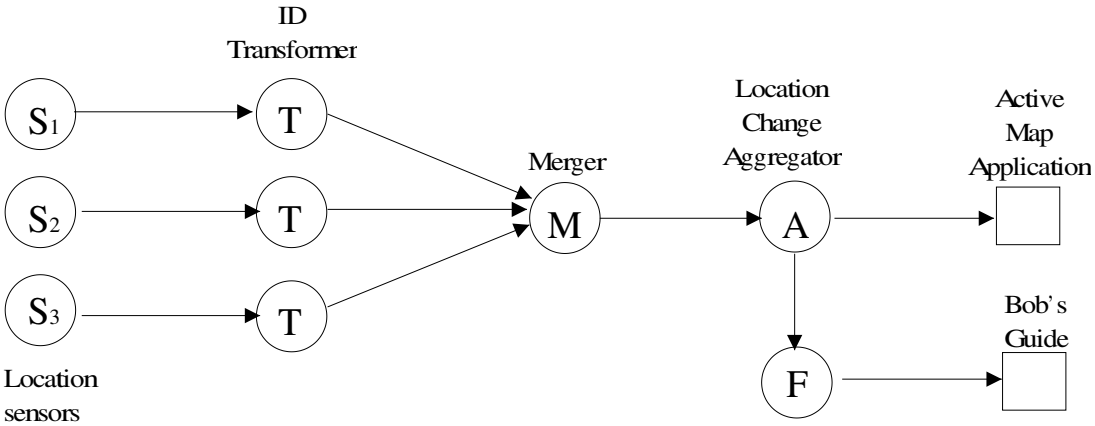
Example. Consider the example in Figure 1, which might be used in an office building with a location-tracking system. Each room has a location sensor that periodically reports the identification number for the badge(s) it detects in that room. *Transform* operators map the badge ID to the name of the person or device attached to that badge, and the *merge* operator combines these streams into one event stream. An *aggregator* operator uses internal state to detect location changes, emitting an event only when a person or device changes location. The Active Map application uses the resulting stream to display the current location of tracked objects. Another application is a "tour guide" for Bob, which uses a *filter* to obtain only events about Bob's movements. Note that both applications can share many of the operators and streams.

This example demonstrates a variety of Solar operators, some stateless, and some with internal state. Ultimately, though, operators are simply objects implementing an Operator interface. Solar allows applications to use operators instantiated from classes found in a library of common operators or from classes provided by the application programmer. As a result, the functionality of an operator is opaque to Solar. Solar asks the operator to handle a new event, and the result is that the operator publishes zero or more events.

Architecture. The Solar system is distributed across many hosts, although the only abstraction presented to operators and applications is the operator graph. Applications run outside the Solar system, on any platform, using a small Solar library that allows them request subscriptions and to receive events over standard network protocols.

Summary. Although we describe our access-control mechanism in terms of the Solar infrastructure, most of its features are not dependent on the specifics of Solar. Ultimately, our mechanism suits any event-based context-dissemination infrastructure in which events are produced by sources, are processed by a shared collection of operators that subscribe, process, and publish derived events, and are consumed by applications. In particular our approach does not depend on Solar's graph structure or model of event streams.

Figure 1: A sample operator graph. T are transformers, M are mergers, A are aggregators, and F are filters.



3 Design goals

We have four objectives in our system design.

Our primary objective is to limit application access to events, because events often contain information considered private by certain individuals or proprietary to certain organizations. We assume that applications run on behalf of a specific principal,[1] so this objective means that applications must only receive events to which their principal is allowed access.

In accomplishing this first objective, consider several properties of the operator graph. First, each node in the operator graph may be shared by many applications and users. Second, there are a variety of information sources and of applications, so there many operators are needed to derive the desired event streams from the sources. It is infeasible to manually specify an access-control list for each event stream. Third, the operator graph changes as applications and sensors come and go; we cannot know the topology of the operator graph in advance. Fourth, for scalable performance the operator graph is likely to be deployed across many servers.

The second objective is avoid the use of any central authority that defines the access-control policies. By generating access-control lists automatically using principles from information flow, then giving users discretion to relax ACLs where they deem appropriate, we empower users and reduce management overhead. We accommodate personal differences in privacy policies, and encourage users to contribute new sources, operators, and applications, and to define their access-control policies, without requiring the blessing of any central authority.

The third objective is to allow users to specify some *context-sensitive* access-control policies. For example, Bob may not release his current location to anyone, but lets others receive events about his location if they are currently in the same room. Since presumably they can see Bob, physically, these events do not significantly reduce Bob's privacy, but may allow the other user to implement many context-aware applications mentioned in the literature (e.g., an application that reminds me to "mention the party the next time I see Bob.").

The fourth objective is to maintain the transparency of operator sharing. Solar, for example, may overlap subscription trees to produce a graph with shared operators.

Finally, there are certain features that are explicitly *not* an objective of our current research.

First, we do not want to enforce a strict information-flow policy, because it is insufficiently flexible. It is also impractical: we cannot assume that all computations will occur inside a trusted environment that maintains information-flow principles. Once an application (outside the event-flow system) has an event, we cannot stop the application from giving away the information. We prefer to directly support limited forms of ACL relaxation, to allow most reasonable information sharing to occur explicitly within the framework.

Second, we do not address covert channels.

Third, we do not protect against any data-mining effort to infer sensitive information from information legitimately obtained.

Finally, this paper is about access control, not about authentication or trust. That said, we do assume that the event system authenticates application users sufficiently to attach a principal to each running application. We also assume that communications use encrypted channels. Thus our focus is on limiting the set of events receivable by legitimate principals, rather than on protection against eavesdroppers. Finally, we assume that applications, users, and operators trust the infrastructure, although the infrastructure does not trust applications or op-

[1] We imagine a special principal *anonymous* that could represent applications running on a public display, kiosk, or other unauthenticated device.

erators.

4 Access-control semantics

In this section, we describe our approach to access control, which is based on principals and access-control lists. Each user is represented by a named *principal*. A principal may be in one or more named *groups*. A group is a list of principals and other groups. An *access-control list (ACL)* is a list of principal and group names. In our notation, p is a principal, g is a group, and U is the universal set of all principals and groups.

We must first decide whether to attach an ACL to each *event* or to each *event stream*. Note that some operators, such as most of those in Figure 1, may publish events that reasonably should have different ACLs. In that example, an event about Bob's location should have ACL $\{\text{Bob}\}$, while an event about Alice's location should have ACL $\{\text{Alice}\}$. We thus choose to attach a separate ACL to each event. Every event e_i has two parts:

$$e_i = (d_i, a_i), \quad \begin{array}{l}\text{where } d_i \text{ is the data field} \\ \text{and } a_i \text{ is the ACL field.}\end{array}$$

An application executes on behalf of one principal. An application may subscribe to any event stream, but may receive an event if, and only if, the application's principal is a member of the event's ACL or a holder of a group mentioned in the ACL. Specifically, for principal p and an event e_i with ACL a_i, we allow access iff $p \in^* a_i$, defined as follows:

$$p \in^* a_i \Leftrightarrow [(p \in a_i) \text{ or } (g \in a_i \text{ and } p \in^* g)].$$

Notice that this definition is recursive, when groups are defined hierarchically. (An implementation must take care, in recursion, to avoid any cycles in group definitions.)

4.1 ACL derivation

An operator may receive events from many different publishers, and produce many different events. It is often inconvenient or impossible for the operator programmer to determine the appropriate ACL for each output event (consider a generic operator that filters room-temperature events). And, since an operator may be shared by many applications and users, it is not possible for any one of them to define the ACL of each published event. So, we wish to derive the ACL for each output event as a function of the ACL of incoming events, the desires of the operator programmer, and the desires of any interested user. We call our approach ACL propagation, because access-control information propagates through the operator graph.

In the discussion that follows, we focus on a single operator. Over its lifetime it has received a series of events $e_1, e_2, ..., e_i$. Events from a given subscription arrive in the order they were published, but events from multiple subscriptions are arbitrarily interleaved into the sequence. The operator is allowed to execute its handler once for each event e_i. The handler's result is to publish a set of zero or more new events $\{e_{ij}\}$, although the handler only produces the data field:

$$\{d_{ij}\} = \text{handleEvent}(d_i).$$

We now show how to derive the ACLs $\{a_{ij}\}$ from a_i in three stages. First, we compute a default ACL using a conservative information-flow approach. Second, the operator programmer may further restrict the ACL. Third, any principal in that ACL may relax the ACL.

Default ACL. After receiving the set $\{d_{ij}\}$, we compute a default ACL DEF_{ij} for each event. The computation depends on whether the operator's handler read or wrote any internal state in computing the event.

For stateless operators, there is no concern about whether sensitive information in earlier events may "leak" into this event. The default output ACL is the same as the input ACL.

$$DEF_{ij} = a_i, \text{ if no state read}$$

For operators with internal state, we compute a default ACL using principles from information-flow control systems [Den84]. Since operators are opaque, we cannot apply information-flow principles inside the operator, and we conservatively derive each output event's ACL as the intersection of the ACLs of every event ever received by this operator. Thus, $DEF_{ij} = \cap a_k$ for $k = 0$ to i. This policy is too restrictive in some cases. Consider the Merge operator in Figure 1, which receives an event about Bob's location tagged $\{\text{Bob}\}$, then an event about Alice's location tagged $\{\text{Alice}\}$. Because of the intersection rule, every event published after the arrival of Alice's event has the empty ACL $\{\}$.

Thus, we force operators to represent any internal state as a set of state objects, each associated with a simple key k (e.g., a string). The key's meaning is determined by the operator programmer; for example, the key may be a principal's name. Think of the set of states as a hash table with methods `s=get(k)` and `put(k,s)` and the following rules apply within the handling of a given event: 1) the operator may call `get` and `put` on any state objects any number of times. 2) The state retrieved by `get` is unchanged unless followed by a `put`; that is, `get` retrieves a copy of the state. 3) A call to `put(k,s)` creates state s_k if it does not exist. These rules prevent information from "leaking" across states.

We compute an accumulated ACL denoted as ACC_i for each input event e_i based on the following operational semantics. We also maintain an accumulated ACL for each state, denoted as ACC_{is}. Informally, since the state may contain information gathered from all prior events, its accumulated ACL must be the least upper bound (intersection) of the ACLs on those prior events. When the handler is called to process a new incoming event, the accumulated ACL for that event and each state are set initially as follows:

$$ACC_i = a_i$$
$$ACC_{is} = ACC_{(i-1)s}, \quad \text{for all state } s$$

where $ACC_{0s} = U$ for all s.

Whenever the handler accesses a state k, the accumulated ACLs are updated according to these rules.

$$ACC_i = ACC_i \cap ACC_{ik}, \quad \text{if } \texttt{get(k)} \text{ is called}$$
$$ACC_{ik} = ACC_i \cap ACC_{ik}, \quad \text{if } \texttt{put(k,s)} \text{ is called}$$

When the handler publishes an event e_{ij}, DEF_{ij} is the ACC_i at that time.

$$DEF_{ij} = ACC_i$$

If there are no calls to `get`, $DEF_{ij} = a_i$; that is, the output event inherits its ACL from the input event. If the operator called `get`, its computation was "polluted" by state information that may contain information that should remain private; DEF_{ij} is then the intersection of the ACL on the event and the ACL on all those states.

Restriction rule. The operator programmer may wish to restrict the ACLs of output events, perhaps because the operator's parameters may contain sensitive information that affects the output results, or because the operator contains an algorithm that computes valuable information from less-valuable input data. Therefore, the developer has the option to include a function Restrict(e_i, d_{ij}) that computes the list of principals to be removed from DEF_{ij}. The default Restrict(e_i, d_{ij}) returns U. We use this function to compute the "output ACL suggested by the operator:"

$$OP_{ij} = DEF_{ij} \cap \text{Restrict}(e_i, d_{ij})$$

Although we anticipate few situations where it would be helpful, we allow the Restrict() function to `get` any state, because it cannot change the data d_{ij} and it can only narrow the eventual ACL a_{ij}.

Relaxation rule. We allow users to attach their own *relaxation function* to any operator, to relax the ACL for events output by that operator. Each principal p may specify their own access-control policy by defining and attaching their own relaxation function, Relax$^p(e_i, d_{ij})$. We expect that in many cases the user may supply a constant rather than a function, that is, Relax$^p(e_i, d_{ij}) = RLX^p$, but the function allows more flexibility. If neither is supplied, the default Relax$^p(e_i, d_{ij}) = \{\}$.

In many object-oriented implementations, operators are instances of a class. In that case, users may attach a relaxation function to the class, to individual instances, or both. Relax^p() is the function attached to the instance, if available; otherwise it is the function attached to the class, if available; otherwise it is $\{\}$.

We compute only those relaxation functions contributed by principals who already have access to the event, according to OP_{ij}. The rationale is that any of those principals may subscribe directly to this operator, receive the events, and pass them to their friends anyway. The relaxation function allows a more structured solution.

We compute the set of principals added by all eligible principals,

$$USR_{ij} = \bigcup \{\text{Relax}^p(e_i, d_{ij}) \mid p \in^* OP_{ij}\}.$$

Note that we use $\in^*$, but it is computationally expensive in the presence of context-sensitive groups (see Section 4.3), so we expect many implementations to use $\in$.

Finally, we can define the ACL of the output event:

$$a_{ij} = OP_{ij} \cup USR_{ij}$$

Sources. Although we discuss operators above, these rules apply to sources as well. The difference is that sources have no input events. All of the above equations hold, then, if we apply them for $i = 0$ and $a_0 = U$, as the source publishes a sequence of events e_{0j}. By definition, $ACC_{0s} = U$ for all s, so $DEF_{0j} = U$. So

$$OP_{0j} = \text{Restrict}(e_0, d_{0j})$$

(though e_0 is null), and a_{0j} is computed as above from OP and USR. Thus, the programmer of the source can use Restrict() to define any output ACL, and that ACL may be relaxed by users according to the same rules as for operators.

Using an ACL approach, a source listing principals p_1 and p_2 in the ACL grants both p_1 and p_2 the ability to receive the event, and thus to share the event with anyone they wish. It is not possible in Solar (or in general, except in a closed system), to enforce a policy that disallows a third party p_3 from receiving the event unless both p_1 and p_2 agree. Such a policy might be appropriate, for example, in an event containing a camera image of users p_1 and p_2.

4.2 Example

Consider the example in Figure 2, which might be used in an office building with a location-tracking system. Each room has a location source that periodically reports the identification number for the badge(s) it detects in that room. *ID Transform* stateless operators map the badge ID to the name of the person or device attached to that badge. A *Location change aggregator* operator, subscribing to all the location sources, uses internal state to detect location changes, emitting an event only when a person or device changes location. The Active Map application for Bob uses the resulting stream to display the current location of tracked objects. The meeting application for Dave monitors a list of people in room 215, by subscribing to the event stream published from the *215 Monitor* stateful operator, which detects people entering or exiting the room.

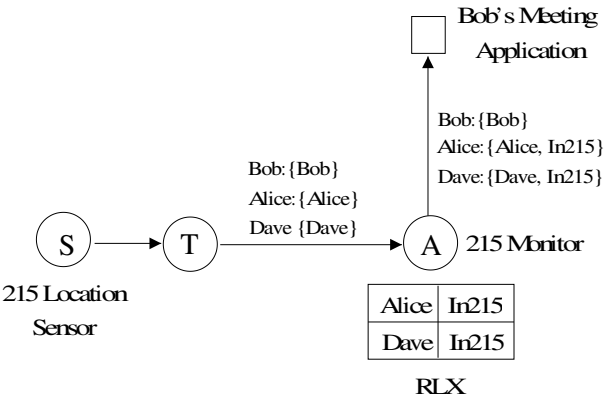
Suppose that Bob, (wearing badge 015) is in room 120, and Alice (wearing badge 232) is in room 215. The location source S_1 publishes badge numbers seen in room 120, and S_2 publishes the badge numbers seen in room 215. Each location source uses Restrict() to define the ACL on their raw sensor data to be $\{\text{locsensor}\}$, a principal representing the administrator of the location-sensing system. Although "locsensor" represents an administrative authority, notice that other kinds of sources could choose different principals for their ACLs. There is no need for "root" or other single central authority.

The ID Transformer operator translates badge IDs into person names; thus the badge IDs "015" and "232" are translated into "Bob" and "Alice" on T_1 and T_2 respectively. To these transformers the administrator (locsensor) added a relaxation function that adds a translated principal name to the output ACL. Thus a principal p can read named location events about person p (assuming a clear mapping between people and principals). This approach is conservative: the raw sensor data remains private to the administrator, and the named sensor data is private to the individual.

The Location-change aggregator takes care to use a separate state object s_p for each principal p, so that $ACC_{is_p} = \{p, locsensor\}$ and those location events can pass through with unchanged ACLs. Solar delivers location events about Bob, published from A_1 to the Active Map application, because the ACLs of those events contain the principal named "Bob". The events about Alice are not delivered to Bob's application (see Section 5.1).

It is left to individuals to decide when to relax the ACL on events about them. Assume that Alice does not want to be visible to most applications, but she is willing to allow Dave's Meeting application (run by Dave) to see that she is in the meeting room 215. She adds Dave to her relaxation list RLX on the aggregator "215 monitor,"

Figure 3: An example context-sensitive policy.



which outputs arrival and departure events about people in room 215. Alice could provide her location to the Active Map application by adding the same relaxation list on the Location-change aggregator.

4.3 Context-Sensitive Groups

There are many instances where the policy itself should be context-sensitive, that is, where the access list produced by a relaxation function depends on the current context beyond the information available in the event. Furthermore, it is difficult for most users to write relaxation functions. For both reasons, we allow ACLs to contain the name of *groups*. Groups usually have semantics that are meaningful to ordinary users, such as "people in room 215." If a group is defined elsewhere to be a context-sensitive set of principals, then any ACL including that group will itself be context-sensitive.

For example, in Figure 3, suppose Alice allows people in the meeting room (215) to receive her location information only when she is there, too. So, she adds a relaxation function (in this case, a constant) that outputs the group name "In215." The group "In215" is defined elsewhere to contain the set of principals corresponding to people currently in room 215; if Bob and Dave are in room 215, their applications can receive the event about Alice (see Section 5.1).

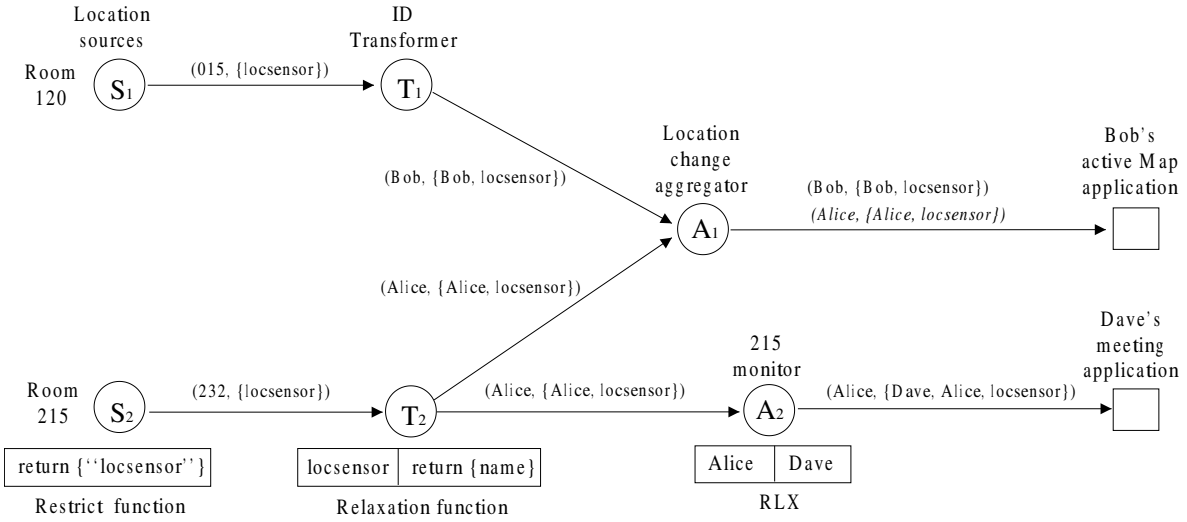
5 Implementation and Performance

In this section, we outline our current implementation and describe initial performance results.

5.1 Implementation

Our implementation is built into a simulator of the Solar context-dissemination system described in Section 2.

Figure 2: An example of ACL propagation. The nodes are operators: S for sources, T for transformations, and A for aggregators. The arrows are labeled with the event represented by a tuple in which the first item denotes the badge ID or the principal name in the data field, and the second item denotes the ACL field. The data field also contains the location, but for brevity we omit that information here.



This simulator represents the entire system in a single Java VM, and neither uses nor simulates a network. For our purpose, to experimentally measure the costs of ACL propagation, this simulation easily isolates the computational overhead from network costs that are not relevant to our study.

We first explain our representation of the ACL field, and then describe two modules in our access-control mechanism. One is the ACL propagation module that derives the ACLs on each source or operator. The other is the access-control enforcement module that limits the set of applications that may receive the event.

Representation of the ACL field. We assume that there exist unique mappings from principal or group names to integer values, which we call the "principal ID" and "group ID". Principal IDs and group IDs are drawn from distinct universes. We represent the ACL field of each event as an ACL object that contains a set of principals and a set of groups. We implement each set as a vector bit set containing the corresponding integer IDs, using Java's BitSet class.

There is an object for each group, easily located (by ID number) through a global hash table. The group object contains the set of principals that are members of the group. (Although our framework also allows groups to be members of groups, our implementation avoids this recursive complexity.) Most group objects contain a static set of principal IDs. Some group objects contain a dynamic set of IDs, defined in a context-sensitive way as in the example in Section 4.3.

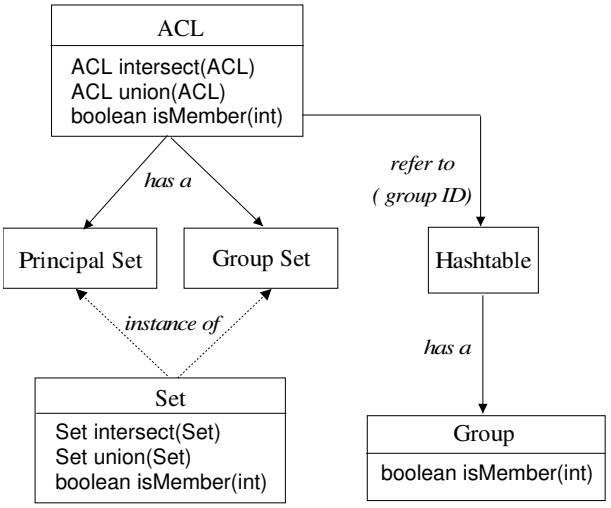
The structure of the ACL and its related objects are shown in Figure 4. The ACL class provides methods for intersect, union, and membership test, which perform the corresponding set operations on the principal and group sets inside. For example, the intersection of two ACLs is a new ACL whose principal set is the intersection of the two other ACLs' principal sets, and whose group set is the intersection of the two ACLs' group sets. The `isMember` method tests membership of a given principal ID in the ACL's principal set and in each group object whose group ID is in the group set.

To define a new context-sensitive group, any user may ask the event system to deploy (and name) a subscription tree whose root operator publishes events listing the current set of member principals.[2] Anyone may keep up-to-date on the group membership, as it changes, simply by subscribing to the group's root operator, by name. We describe the details of one such implementation in another paper [not cited here due to the blind-review policy].

ACL propagation module. Figure 5 depicts the set of components deployed at each node in the operator graph. Collectively, under the control of a thread subclass we call "OPRunner", three components process each incoming event and produce any output events. The OPRunner thread extracts an input event e_i from the input queue,

[2]For efficiency, most implementations would allow three types of events: `set` *`list`*, which announces the full membership, `add` *`list`*, which announces additions to the membership, and `del` *`list`*, which announces deletions from the membership. Any subscribers would maintain the current list in their state.

Figure 4: Structure of the ACL object. Each component is shown with its class or instance name. The methods of the class are listed below the class name.



sends a_i to the StateSet object to initialize the accumulated ACLs, sends the data field d_i to the Operator object to compute any new event data d_{ij}, and then both e_i and each d_{ij} to the ACLHandler object. We describe each component in turn.

We provide an abstract Operator class. The operator developer must implement the `handleEvent` method. The method receives the data field d_i of an input event, and may publish one or more output events d_{ij}. The developer may optionally override the default `Restrict` method so that it returns a list of principals and groups to be removed from the event's ACL.

To implement information-flow semantics, we must be aware of any operator state and the set of previous events that may have affected that state. Since operators are written in Java, this awareness is difficult. Our model and implementation provide a compromise. Operators must have no internal state other than that managed by our system. The `handleEvent` method stores any persistent state in the StateSet object. We currently assume that the `handleEvent` method cannot use instance variables, access files, or open network connections. In a complete implementation, we can enforce these properties through static byte-code analysis.

The StateSet object offers only `get` and `put` as public methods. A state can be created or updated only by calling `put`, because `get` returns a copy of the state, rather than a reference to the system's copy of the state. The StateSet object maintains the accumulated ACL ACC_i for the current input event e_i. Upon receiving e_i, the OPRunner calls the StateSet to initialize ACC_i to a_i. The StateSet also tracks the accumulated ACL for each state s. The accumulated ACLs are updated using the rules in Section 4.1.

The ACLHandler object maintains a set of relaxation objects defined by users. Each object, a subclass of our abstract RelaxFunc class, contains an implementation of the `Relax` method and the ID of the principal who attached that relaxation function to the operator. (In Solar, for example, these objects are attached by the user deploying the operator or using the operator as a source.)

Whenever the Operator's `handleEvent` method calls `publish`(d_{ij}) to publish a new event, `publish` calls upon the ACLHandler to construct the corresponding ACL a_{ij}. It computes DEF from ACC_i in the StateSet object, computes OP by calling the `Restrict` method of the operator, and computes USR by calling the `Relax` method of eligible RelaxFunc objects, and produces a_{ij}. The `publish` method combines the data field and the ACL field for the output event, and posts it into the output queue.

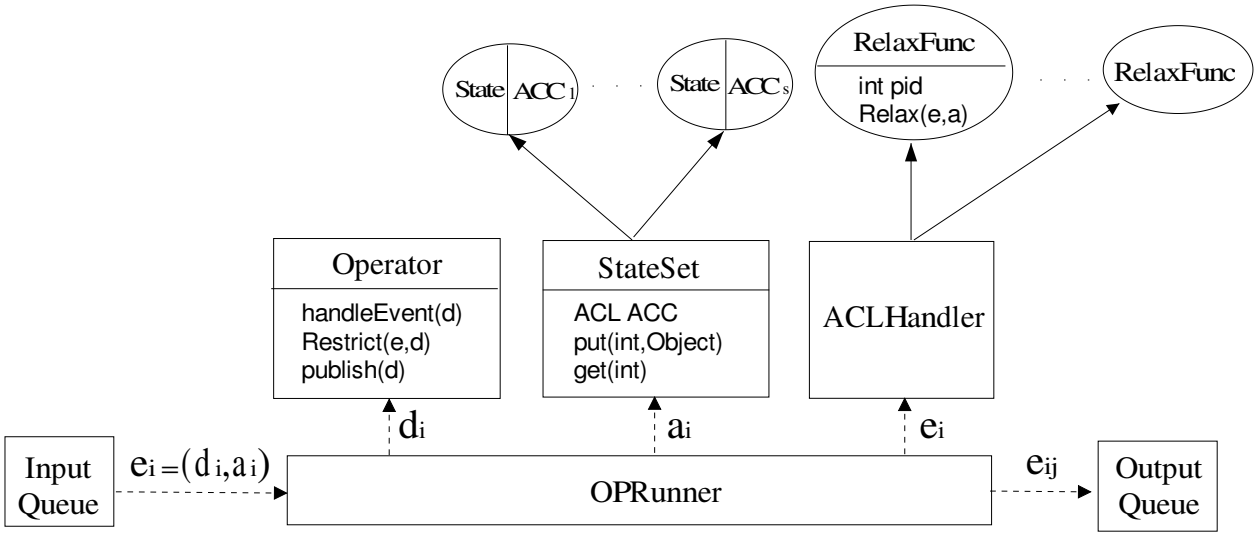
Access-control enforcement module. Ultimately, the point of computing an event's ACL is to limit the set of applications that may receive the event. For this purpose, Solar adds a special *ACL operator* at the root of every subscription tree, as shown in Figure 6. For an event $e_i = \{d_i, a_i\}$, this operator's special event-handling method `accessAllowed(`a_i`)` returns a boolean indicating whether to pass d_i on to the application. The ACL operator, constructed with parameter p where p is the principal running the application, returns true iff $p \in^* a_i$. To evaluate that expression, an ACL operator must check p's membership in any groups named in a_i.

Our current simple approach requires to call the `isMember` method of every Group object in the group set of an ACL to test the membership of a given principal p. This approach would be computationally expensive if the Group objects were located in remote hosts. We, therefore, prefer subscription-based approaches that allow the ACL operator to maintain a list of p's current groups. For example, for any principal p with active applications, our system deploys a trusted aggregator that subscribes to all groups, publishing an event whenever p joins or leaves a group. ACL operators for p subscribe to this operator. (This approach explains why we chose to implement access-control enforcement as an operator; its normal `handleEvent(`d`)` method processes the incoming group updates, while its special `accessAllowed(`a`)` method processes incoming data events.)

5.2 Performance measurements

Although performance is not our primary goal, clearly it is important that our flexible access-control scheme not add unreasonable overhead to the flow of context information. In this section we examine the latency and through-

Figure 5: Structure of the ACL propagation module. $e_i = (d_i, a_i)$ denotes the input event, and e_{ij} denotes the output event.



put of event propagation between sources to applications, through operator graphs of varying size and with ACL operations of varying complexity.

We measured event latency using the implementation described above. In our implementation (a mock Solar system), all sources, operators, and applications execute within a single Java virtual machine. Clearly this design dramatically reduces the cost of event distribution, compared with a real Solar system distributed across multiple VMs and hosts. By eliminating all network costs, and much of the event-distribution overhead, our experiments here focus on the latency of ACL operations. As a result, our measurements represent inherent overhead of our approach and implementation, independent of the efficiency of Solar or any similar context framework.

For our experiments, the testing framework ran on JDK 1.4.1 on Linux RedHat 7.3, on a host with a 2 GHz Pentium 4 CPU and 256 MB memory.

In all of the experiments that follow we constructed a linear operator graph, with a single source, zero or more operators, an ACL operator, and a single application. We implemented simple, concrete versions of the necessary classes: a Source class that periodically publishes an event containing a small payload and a timestamp, an Operator class that accessed a given number of state objects during each call of `handleEvent` (each access involved `get` followed by `put`) and re-published every event received, an ACLoperator that tests membership but ignores the result and forwards all events, an Application class that recorded the latency of each event (its arrival time minus its timestamp), and a RelaxFunc class described below.

Our testing framework allowed us to specify the following parameters. Here "range" refers to the number of items in the universe of a set, and "length" refers to the number of items actually in the set.

OP Number of operators in the graph, in addition to the ACL operator.

PR Range of the principal set.

PL Length of the principal set.

GR Range of the group set.

GL Length of the group set.

ST Number of states accessed by the `handleEvent` method on each operator.

FN Number of relaxation functions attached on each operator.

We implemented a simple RelaxFunc class. Its `Relax` function returned an ACL with a principal set and a group set. Half of the principal set's PL members were fixed and the other half were chosen randomly from the universe of PR principals. The group set had GL members randomly chosen from the universe of GR groups. The fixed subset of principals was necessary to avoid driving the states' accumulated ACLs to the empty set.

We used the same implementation and parameters for the `Restrict` method of the Source and the Operator objects.

We did not use context-sensitive groups (again, we examine context-sensitive groups in detail in another paper, not cited due to the blind-review policy).

Figure 6: The system inserts a special ACL operator at the root of every subscription tree.

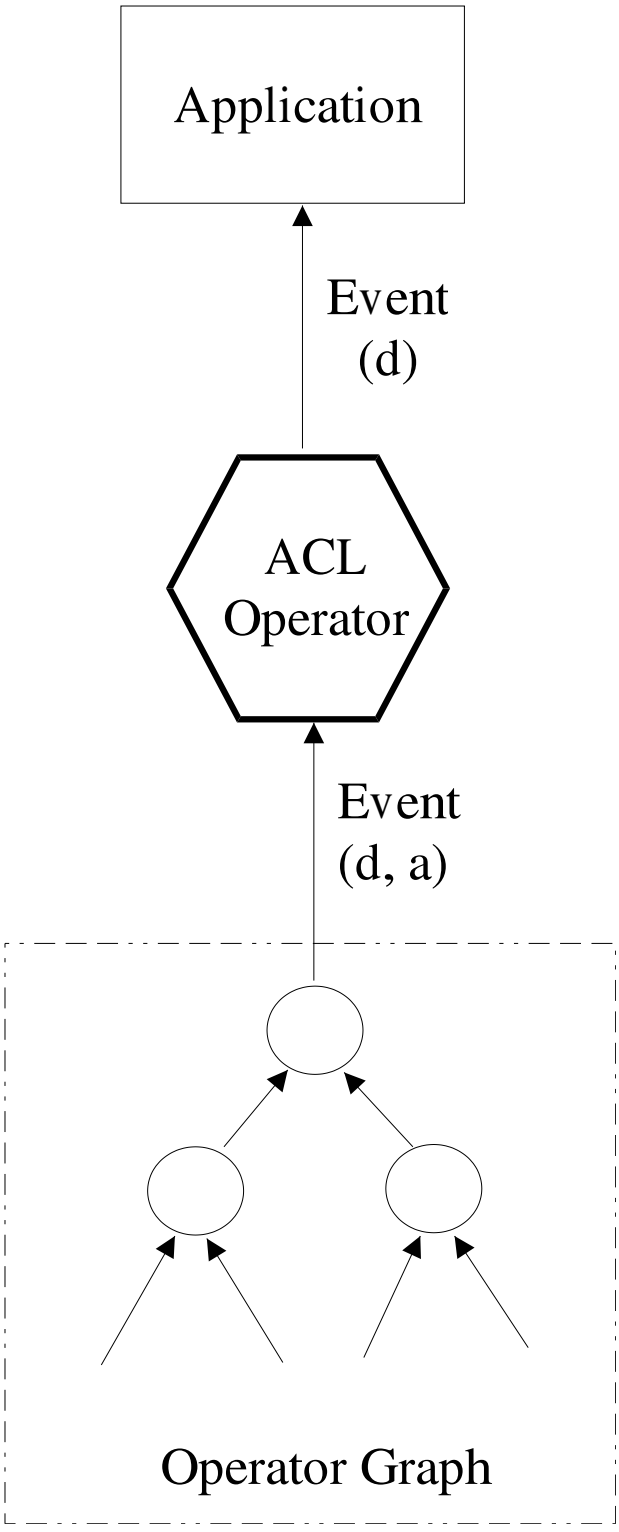
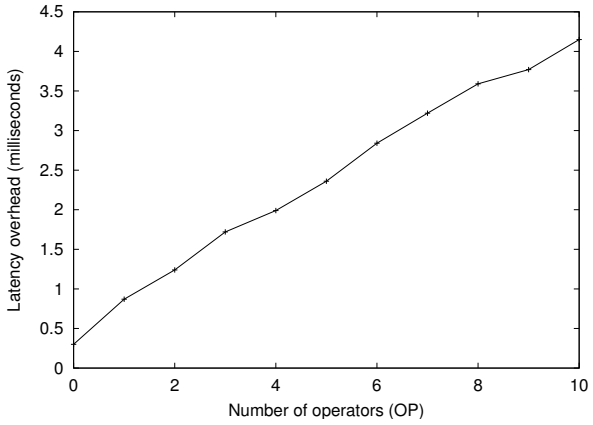


Figure 7: Result of latency overhead measurements with varying number of intermediary operators. The parameters for the source and all operators were $PR = 500, PL = 250, GR = 50, GL = 25, ST = 3$ and $FN = 3$.



Our source published one event every 500 msec. Our application measured each latency and reported the average across 100 such events. We ran all experiments with and without ACL computation; we can subtract the latency *without* the access-control mechanism from the latency *with* the access-control mechanism to compute the ACL-computation overhead.

Figure 7 shows the result of latency overhead measurements with a varying number of operators. The aggressive parameter values assume that half of all principals and half of all groups were involved in every set, that every event handler read and wrote three states, and three relaxation functions applied to each operator. Most realistic deployments, we expect, would use much smaller parameters. Each group object holds a set of principals specified by the same PR and PL parameters.

The figure shows that the latency was roughly proportional to the number of operators, because the number of set operations performed on the Set objects in the ACL objects was proportional to the number of operators, and the computation cost of each operation was fixed here by the range (PR or GR) of the set.

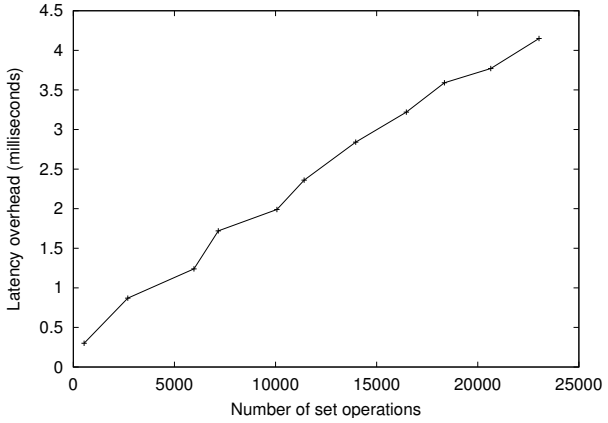
The number of intersect operations performed at each operator was determined by the ST parameter. The number of union operations at each operator was determined by the number of relaxation functions executed; this number varied since not all were eligible for each event. Table 1 presents the number of each set operation, and Figure 8 shows the relationship between the latencies and the sum of intersect, union and membership operation counts. The graph shows that the latency was proportional to the number of set operations.

We also examined the throughput of our system. The

Table 1: The number of each set operation.

#operators	0	1	2	3	4	5	6	7	8	9	10
#intersect operations	200	1788	3376	4964	6552	8140	9728	11316	12904	14492	16080
#union operations	0	202	412	614	836	1034	1264	1466	1678	1862	2142
#membership test operations	334	705	1177	1580	2181	2241	2970	3688	3771	4287	4806
latency overhead(ms)	0.3	0.87	1.24	1.72	1.99	2.36	2.84	3.22	3.59	3.77	4.15

Figure 8: Relationship between the latency overheads and the sum of intersect and union operation counts. OP changes from 1 to 10. The other parameters are $PR = 500, PL = 250, GR = 50, GL = 25, ST = 3$ and $FN = 3$.



conditions were similar to those used for the latency measurements, except the source published events as fast as possible, rather than once every 500 milliseconds. Figure 9 compares the result of throughput measurements with and without ACL computation. The figure shows that the throughput in both cases was roughly in inverse proportion to the OP parameters. Although the throughput with ACL computation was smaller by an order of magnitude than the one without ACL computation, notice that in our experiment there was no computation within each operator. A real operator graph would have substantial operator computation and the overhead of ACL computations would represent a smaller proportion of the overall time. In our experiment, the throughput with the access-control mechanism was still more than 200 events per second in the most challenging case ($OP = 10$).

Our next experiment measured the scaling behavior with respect to the range of principal and group sets. Figure 10 shows the latency overhead resulting from various principal (PR) and group (GR) set ranges. [Here we assumed that the range of the group set was equal to or smaller than the range of the principal set, since it seems likely that the number of groups is less than the number of principals in most situations.] We set the length of each set to be half of its range, so both scale linearly.

Figure 9: Result of the throughput measurements with varying number of intermediary operators. The parameters for the source and all operators were $PR = 500, PL = 250, GR = 50, GL = 25, ST = 3$ and $FN = 3$.

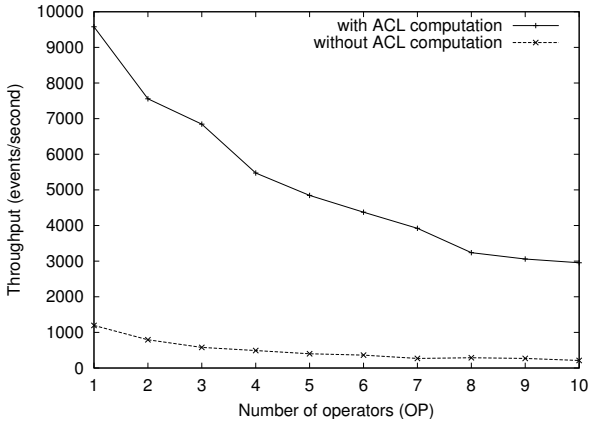


Figure 10: Result of latency overhead measurements varying the range of principal and group sets. The parameters are $OP = 10, ST = 3, FN = 3, PL = PR/2$ and $GL = GR/2$.

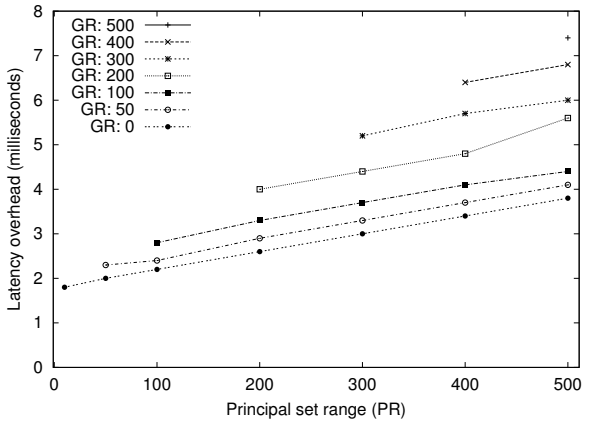


Figure 11: Result of the throughput measurements varying the range of principal and group sets. The parameters are $OP = 10, ST = 3, FN = 3, PL = PR/2$ and $GL = GR/2$.

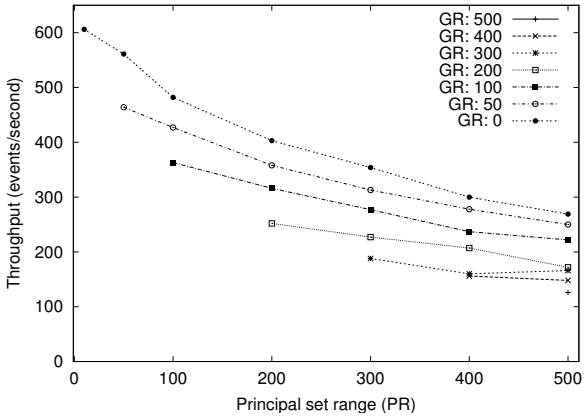
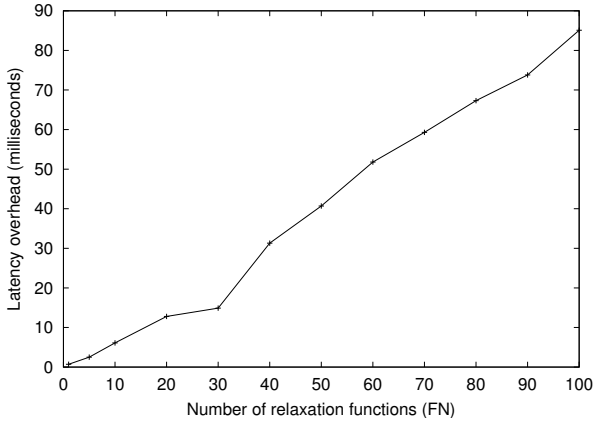


Figure 12: Result of the latency overhead measurements with varying number of relaxation functions to be attached to each operator. The parameters are $OP = 10, PR = 500, PL = 250, GR = 100, GL = 50$, and $ST = 3$.



For a given group set range, it is clear that the latency overhead was proportional to the principal set range. In our implementation, the cost to compute the intersect or union operation on the principal set is proportional to its range, and the number of set operations does not depend on the range or length of the principal set.

For a given principal set range, the overhead grows linearly with respect to the group set range, too. The number of membership operations grows linearly with the length of the group set, and cost for intersect and union operations on the group set also grows also linearly.

The preceding graphs demonstrate that the latency was, not surprisingly, linear in the number of set operations, and in the size of the set operations. The absolute numbers are of interest as well. In the worst case, given our parameters, the overhead per event was less than 8 msec. For most context-aware applications, this overhead is negligible.

Figure 11 shows the throughput measurements with the same set of parameters as in Figure 10. The throughput with the same GR parameter was in inverse proportion to the the parameter PR. The throughput in the worst case was 126 events per second.

Figure 12 shows how the latency depended on the number of relaxation functions attached to each operator, FN. The same number of relaxation functions were attached to each operator. The latency was linear in the number of relaxation functions, since the number of union operations is proportional to the number of relaxation functions. The latency reached nearly 90 msec here, which is somewhat alarming, but note that occurred in a 10-operator chain with 100 relaxation functions computed on every operator!

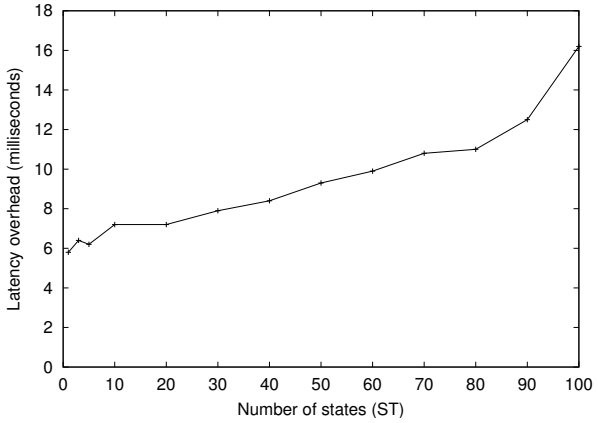
Figure 13 shows how the latency overhead depended on the number of states accessed for each operator, ST. The latency was roughly linear in the number of states, since the number of intersect operations performed in the StateSet objects was proportional to the number of states to be accessed. Note, however, that we expect most operators will access zero or one states per event.

Summary. Our performance measurements show that the ACL computation added less than 10 msec to the latency of event delivery, even with aggressive parameter values. The latency overhead was roughly linear with respect to most of the parameters. We also conducted some of the same test cases in terms of the throughput. The throughput is in inverse proportion to the the parameters such as OP, PR, and GR. The throughput in the most aggressive test case is still more than 100 events per second. We believe that these overheads are more than acceptable for most context-aware applications.

6 Related work

There is little published about access control in event-distribution systems for pervasive computing. Dey [Dey00] built an experimental mechanism to control access to context in the Context Toolkit. The developer of a widget object can specify the "owner" of the information being sensed, using some rules in a way similar to our Restrict() function in Section 4. Their mechanism, however, does not support automatic derivation of ACLs like ours, so it is necessary to specify the functions for every widget object. In addition, it does not provide a mechanism for users to specify relaxation policies.

Figure 13: Result of the latency overhead measurements with a varying number of states to be accessed by each operator. The parameters are $OP = 10, PR = 500, PL = 250, GR = 100, GL = 50$, and $FN = 3$.



Our ideas are substantially influenced by recent work on information-flow models of access control. Since the traditional information-flow policy is too restrictive, several projects provide ways to "declassify" information. Ferrari [BDFS98] describes an object-oriented system that controls access to objects using information-flow principles, but provides trusted functions that can relax the strict information-flow policy. That is, a principal with no rights to access some object directly can access it through the trusted function. They, like us, use ACLs to define the security level in their flow policy. Although their trusted function does allow access to objects by principals not in the ACL, it is not clear how their mechanism could be used in our situation, in which we relax an event's ACL so it can be accessed, later, by applications downstream. Furthermore, in their paper it is not clear whether ordinary users would have the power to implement trusted functions or contribute to the relaxation lists $RW(m)$ and $IW(m)$ for the method m.

Myers et al. [ML00] extend the Java programming language to allow variables to be tagged with labels. Labels contain an access-control policy (essentially, an ACL) for each "owner" of that variable. Variables computed from others are given a label computed from the labels of operands in the computation, according to information-flow principles. An owner principal may declassify a variable by extending its own ACL in the label, but other owner's ACLs are unchanged. A principal may only obtain the value of the variable if it is in the *intersection* of the owners' ACLs. Our relaxation semantics are more liberal than their declassification semantics, as we allow anyone with current access to add any other principal to the ACL. Although their approach is intended for compile-time analysis of fine-grained information-flow in Java programs, it may be possible to apply the same concepts to events and operators, at run-time.

Covington et al. [CLS$^+$01, CAS01] introduces the environmental role (ERole) to achieve context-aware authorization. Their approach is based on the concept of Role-based access-control (RBAC); constraints on environmental (context) variables can be defined with a Prolog-like logic language. Authorization is based on an ordinary role and an ERole; in effect, the ERole is an additional condition to be satisfied for an authorization decision. ERoles, though context-sensitive, are different from the context-sensitive groups in our approach. An ERole is about the state of the environment, and not about the relationship of a principal to the environment, as in our "In215" group.

7 Summary

Security is a critical component in any realistic deployment of pervasive computing. A pervasive-computing infrastructure necessarily collects a lot of context information to disseminate to its context-aware applications, but due to the personal or proprietary nature of much of this context data, the infrastructure must limit access to context information to authorized persons. We propose a new access-control mechanism for event-based context-distribution infrastructures. Our approach is based on ACL propagation, in which each event e is tagged with an ACL derived automatically from the ACL on events that contributed to the production of e. Our approach is based on a conservative information-flow model of access control, but we allow users to express their access-control policies through relaxation functions. This combination of automatic ACL derivation and user-specified ACL relaxation allows access control to be determined and enforced in a decentralized, distributed system with no central administrator or central policy maker. It also allows users to express their personal balance between functionality and privacy. Finally, our infrastructure allows access-control policies to depend on context-sensitive groups, allowing great flexibility.

We met our four objectives: 1) applications may only receive events to which their principal has access, 2) access-control policies are determined by information-flow principles and relaxed where appropriate by request of users, 3) the access-control policies can themselves be context-aware, through the use of context-sensitive group definitions, and 4) operators and their event streams may be shared by multiple users and applications, while this sharing remains transparent to the users and applications. Our initial performance measurements shows that the overhead due to the ACL propagation is reasonably small, and scale to support large number of users and static groups.

In the future, we plan to study important qualitative metrics, the *flexibility* and *usability* of our approach. We are building several context-aware applications, and will use them to evaluate how flexibly users can define relaxation policies, whether those policies are meaningful for multiple applications. and how much administrative overhead is involved in management.

Finally, although this paper describes our work in the context of the Solar system, we believe that our ACL-propagation technique would also apply to many other event-based context-dissemination infrastructures.

References

[BDFS98] E. Bertino, S. De Capitani di Vimercati, E. Ferrari, and P. Samarati. Exception-based information flow control in object-oriented systems. *ACM Transactions on Information and System Security*, 1(1):26–65, November 1998.

[BKS+99] Guruduth Banavar, Marc Kaplan, Kelly Shaw, Robert E. Strom, Daniel C. Sturman, and Wei Tao. Information flow based event distribution middleware. In *Proceedings of the Middleware Workshop at the 19th IEEE International Conference on Distributed Computing Systems*, pages 114–121, Austin, Texas, May 1999. IEEE Computer Society Press.

[CAS01] Michael J. Covington, Mustaque Ahamad, and Srividhya Srinivasan. A security architecture for context-aware applications. Technical Report GIT-CC-01-12, Georgia Institute of Technology, Atlanta, GA, May 2001.

[CK02] Guanling Chen and David 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.

[CLS+01] Michael J. Covington, Wende Long, Srividhya Srinivasan, Anind K. Dey, Mustaque Ahamad, and Gregory D. Abowd. Securing context-aware applications using environment roles. In *Proceedings of the Sixth ACM Symposium on Access Control Models and Technologies*, pages 10–20. ACM Press, 2001.

[CPT+01] Norman H. Cohen, Apratim Purakayastha, John Turek, Luke Wong, and Danny Yeh. Challenges in flexible aggregation of pervasive data. Technical Report RC21942, IBM Research Division, Thomas J. Watson Research Center, P.O.Box 704, Yorktown Heights, NY 10598, January 2001.

[CPWY02] Norman Cohen, Apratim Purakayastha, Luke Wong, and Danny L. Yeh. iQueue: A pervasive data aggregation framework. In *Proceedings of the Conference on Mobile Data Management*, pages 146–? IEEE Computer Society Press, January 2002.

[Den84] Dorothy Denning. *Cryptography and Data Security*. Addison-Wesley, 1984.

[Dey00] Anind K. Dey. *Providing Architectural Support for Building Context-Aware Applications*. PhD thesis, College of Computing, Georgia Institute of Technology, December 2000.

[EHL01] Maria R. Ebling, Guerney D. H. Hunt, and Hui Lei. Issues for context services for pervasive computing. In *Proceedings of the Workshop on Middleware for Mobile Computing 2001*, Heidelberg, Germany, November 2001.

[HB01] Jeffrey Hightower and Gaetano Borriello. Location systems for ubiquitous computing. *IEEE Computer*, 34(8):57–66, August 2001.

[Lan01] Marc Langheinrich. Privacy by design—principles of privacy-aware ubiquitous systems. In *Proceedings of UbiComp 2001: International Conference on Ubiquitous Computing*, volume 2201 of *Lecture Notes in Computer Science*, pages 273–291. Springer-Verlag, 2001.

[ML00] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

[Sat01] M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, 8(4):10–17, August 2001.

[ST93] Mike Spreitzer and Marvin Theimer. Providing location information in a ubiquitous computing environment. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 270–283, Asheville, NC, 1993. ACM Press.