

Designing a Publish-Subscribe Substrate for Privacy/Security in Pervasive Environments*

Lukasz Opyrchal
Miami University
Oxford, OH
opyrchal@muohio.edu

Atul Prakash
University of Michigan
Ann Arbor, MI
aprakash@umich.edu

Amit Agrawal
Indian Institute of Technology
New Delhi, India
csu02103@cse.iitd.ernet.in

Abstract

To help explore issues in design of pervasive environments, a sensor network for tracking locations, and potentially other activities, is being deployed in parts of several buildings at the University of Michigan. Managing privacy is expected to be a significant concern for acceptance of such pervasive environments. This paper outlines an initial design of a publish-subscribe communication substrate for controlled distribution of sensor data. The focus of this paper is on policy management so as to provide means for allowing users to control distribution of data tagged with their ID to other users and services. The paper shows how a wide variety of policies can be specified in the system and points out directions for future work.

1. Introduction

Computational environments are becoming increasingly pervasive due to increased interest in use of technologies such as RFID tags for tracking objects, people, and for use of data mining methods for determining users' behavior to help deliver more targeted products and information, etc. Applications are also emerging where cell phone companies track users' location to help provide location-based services such as notifying a user when their friends are nearby, or of restaurants and gas stations in the vicinity.

However, such environments also raise significant privacy concerns in the minds of people, whether the concerns are grounded in reality or not. For example, findings from the Active Badge systems [13] suggest that individuals do not wish to have their movements available to everyone. Finding a solution to those concerns is of considerable importance if pervasive environments are to be widely used.

With the help of a National Science Foundation infrastructure grant, a location sensor network is being deployed in several parts of Department of EECS building, consisting initially of RFID and 802.11g location tracking sensors.

In addition, a location sensor network is also being deployed in one of the medical clinics at the University of Michigan to help track locations and activities of patients who suffer from memory problems and are recovering in the clinic so that they can be gently reminded if they miss a recommended appointment or activity. In [10], privacy is defined as simply the ability of an individual to control the terms for acquisition and usage of their personal information. The question that we consider in this paper is how can one build applications and services around the data that will be collected, while providing means to the users to have significant control over the conditions of distribution of their data?

We note that the focus of this paper is limited to those distribution constraints that can be managed and enforced by a computational infrastructure. At some level, legal mechanisms are likely to be required to deter people from leaking information that they have access to. For this paper, the assumption is that people are not generally looking to maliciously violate privacy rules. For localized environments such as the medical clinic or the EECS department, we believe such an assumption to be reasonable. At the same time, however, the goal of the computational infrastructure should be to prevent inadvertent violations of users' privacy policies to the extent possible, and thus finding formal ways of encoding policies is desirable so that they can be computationally enforced.

To illustrate the issues in policy specifications, an example of a simple policy used by some mobile phone services (e.g., i-mode services from DoCoMo and recently from AT&T) is one that allows users to determine their friends' nearest cell, provided their friends have given them permission to do so (similar to permissions to view status in instant messaging systems). In general, however, the policies can be much richer:

- Environment-dependent sharing. Users may want to share their location information only at certain times of the day, or when they are in certain locations. Users may want to specify that their location is available during specified events, so that they can be more easily tracked down, for example, if they are late for the event.

* This work is supported in part by grants from the National Science Foundation (grants CCR 0082851 and ATM 0325332), Intel, IBM, and Microsoft.

- Privacy-protected access to location-based notification and autominder services. It should be possible for a user to receive location-based notifications, such as the nearest gas station, without disclosing their location to the gas station. Anonymizing, trusted services will be needed as intermediaries.

In recent years, publish subscribe (pub-sub) middleware has become an emerging paradigm for distribution of data among users and services. In publish-subscribe systems, there are two types of users: publishers and subscribers. The infrastructure mediates delivery of events from publishers to subscribers. The current commercial publish subscribe middleware implement the *subject-based* paradigm, where every event is annotated with one of the pre-defined subjects (topics, channels, etc.) [5, 23, 16, 22]. Subscribers are allowed to subscribe to one of the pre-defined topics.

An emerging alternative to subject-based systems are *content-based messaging systems* [21, 3, 9, 12, 15]. These systems support an event schema defining the type of information contained in each event (message). For example, applications interested in location information of users may use the event schema:

```
LOC_INFO: [user: String, building: String, room: String,
           time: integer]
```

A content-based subscription is a predicate against the event schema, such as

```
(user = "aprakash" & building == "EECS Building")
```

Only events that satisfy the subscription predicate are delivered to the subscriber. Examples of content-based publish subscribe systems include PreCache [20, 11] and content-based prototypes from Microsoft [8] and IBM [3].

To help provide an infrastructure for distribution of sensor data to applications that handle these types of policies, we propose to use a content-based messaging substrate as the underlying mechanism. However, there is a significant difference from earlier work in content-based publish-subscribe systems. In existing systems, the focus is on subscribers being able to control what information they receive by specifying predicates, for example, to handle information overload problem. In contrast, our primary focus is on publishers being able to control who receives their data and under what terms. We thus need to augment the content-based publish-subscribe paradigm to allow publishers (users) to control dissemination of information they own.

A general description of security requirements in content-based systems is given in Wang *et al* [24]. The authors provide a high level description of potential issues and point in the direction of possible solutions. One of the first attempts at solving the access control problem in content-based systems is presented in [4]. The authors combine role-based access control (RBAC) with a distributed event notification service. Unfortunately, the authors do not describe the details of their policy language and the type of access control rules that can be supported by their system.

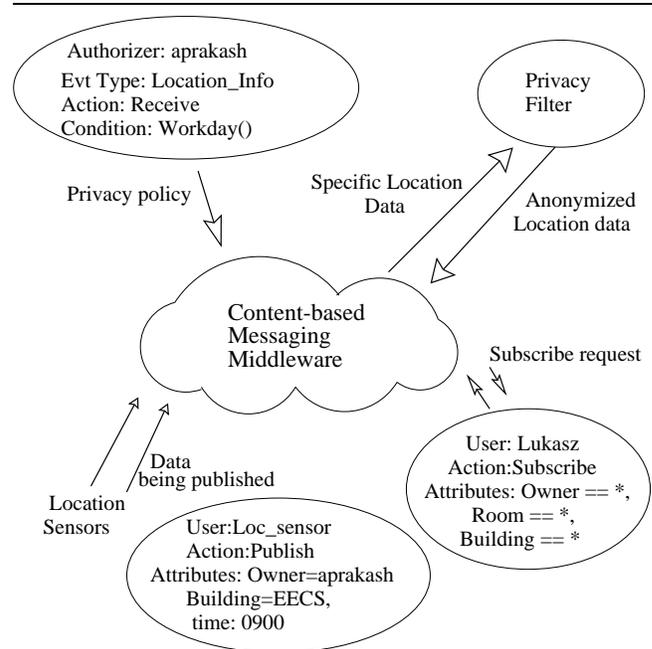


Figure 1. Incorporating policies in a content-based messaging system

The most advanced system that we are aware of that supports privacy in pervasive environments is the Confab system by Hong and Landay [14]. There are several differences in approach. While our system is based in a content-based messaging middleware, their system is based on hybrid blackboard and dataflow architecture, leading to potentially different programming models. In Confab, each data item is tagged with privacy preferences and at present only relatively simple preferences are supported. In contrast, our approach is to make a distinction between privacy policies and data. Privacy policies generally come from users and can apply to multiple data items, as opposed to being explicitly attached to each data. Both approaches have pros and cons, depending on the threat model and the targeted applications.

Figure 1 shows an example scenario in which our system is used. A user *aprakash* has expressed a privacy policy to allow receipt of his location information only during the Workday. Location sensors publish data for different users, including *aprakash* (which is time-stamped by either the sensors or the middleware). A user has expressed an interest in receiving location information for all users, but will only get location information for *aprakash* during the Workday period. Of course, as pointed out above, the policies can be richer. The user *aprakash* may wish to restrict availability of published data to only certain users or services. Furthermore, the user may wish to allow or restrict the ability of those users or services to delegate their right to other users. We consider these issues later in the paper.

In our model, subscribers to data can be either users, end-

user applications, or services. Services can include *privacy filters* that anonymize sensor data or aggregate data from multiple sensors or over time. Examples of data anonymizing include hiding the name of the user from location data, abstracting the location data so that a recipient only knows that the user is in a building, but not the specific room, etc.

We note that for some users, relying on a common publish-subscribe infrastructure completely to do the right thing with shared data may still be a significant trust issue. One idea we have considered is to support the use of multiple *publish-subscribe* brokers, in which a user has a designated trusted *privacy broker*, which could be running as a service on a machine that the user owns. The privacy broker could use our policy infrastructure, just like a centralized service would, except that it would only be handling events that pertain to the users that trust it.

The rest of the paper is organized as follows. Section 2 describes our security models for content-based publish subscribe systems, security policy dimensions relevant to these systems and our security policy language. Section 3 describes a prototype of a secure content-based publish subscribe system based on the security model and policy language described here. Finally, in Section 4, we present our conclusions and directions for future work.

2. Policy Model

Both the privacy as well as the security research communities have examined issues in representing policies, with P3P [1] being an example representation in the privacy community and techniques such as role-based access control models (RBAC), trust management systems, Chinese wall models, and Clark-Wilson models in the security community [6]. Our work largely builds on the work in the security community because security policies have been a subject of investigation for a long time and the maturity of the tools for enforcing those policies. Also, our long-term interest is in taking a unified approach to security and privacy because ultimately, security underpinnings are required to enforce privacy when unauthorized users attempt to tap available data.

For the purpose of this paper, we will assume that all clients connecting to a content-based publish subscribe system are authenticated. The data security problem in content-based publish-subscribe systems is further discussed in [17]. In this paper, we primarily focus on access control and delegation aspects in publish-subscribe systems and their extensions to managing privacy in pervasive environments.

2.1. Basic Definitions

Event Owner: Similarly to Belokosztolszki *et al* [4], we introduce the notion of an **event owner** in our model. Event owner is an entity who has the right to authorize other entities to perform certain actions. An event owner can authorize other users to subscribe to its events, receive events, or

```

Authorizer: POLICY
Licensee: admin
Conditions: (app_domain == "LOC_APP") -> "true";

Authorizer: admin
Licensee: joe
Conditions: (app_domain == "LOC_APP") &&
            (evtType == "LOC_INFO") &&
            (user == "joe") && (owner == "joe") -> "true";

```

Figure 2. Entity admin receives all rights for the LOC_APP application and grants ownership rights to user joe when the user attribute is “joe”

even delegate authority to modify the policy for the events it owns. In most applications, we will associate a different owner for individual events within an event type. For example, events of type “LOC_INFO” can be owned by different users - if an event is about *joe*, then joe is the owner of that event. Each owner will control access to the events it owns

Application: The pub-sub system can support multiple *applications*, where applications refer to broad categories such as a user-tracking system. In turn, each application consists of a number of event types. Each application must have at least one *administrator*. The administrator is a client of the pub-sub system who has the right to delegate authority to perform different actions within the application. For example, the administrator can delegate the *ownership* of different event types, add new event types, etc. Figure 2 shows the administrator of a LOC_APP application (location-tracking application) delegating rights to user *joe* when attribute *user* is equal to “joe”.

This paper does not present a new policy evaluation technique. Our prototype implementation, uses the KeyNote Trust Management System [7] for evaluating and checking our security policy. Therefore, all of our policy examples are based on the KeyNote language syntax. We briefly describe the different fields that are found in KeyNote policy specifications:

Authorizer - this is the entity granting the right. The authorizer can be any entity in the system (provided it is allowed to add policy rules). If an authorizer tries to grant rights which it doesn’t have itself, the rule is rejected. The user *admin* is implicitly granted all rights. This is achieved by writing a rule with authorizer being set to “POLICY” (figure 2). Such rule is always trusted. The authorizer field must not be empty.

Licensee - this is the recipient of the right. The licensee field can contain a single entity, a list of entities, or a wildcard¹. A wildcard indicates that any user who satisfies the condition specified in the *condition field* is given the right.

Condition - this is the condition that is checked when policy rules are evaluated. If the condition evaluates to “true”

¹ In the KeyNote language there are no wildcards. To delegate the right to any user who satisfies the condition, the licensee field must be left out of the rule.

```
Authorizer: admin
Licensee: joe
Conditions: (app_domain == "LOC_APP") && (evtType == "LOC_INFO") &&
((action == "SUBSCRIBE") || (action == "RECEIVE") ||
(action == "CHANGE_POLICY")) && (owner == "joe")
-> "true";
```

Figure 3. Assignment of ownership rights to the owner.

then the licensee is given the appropriate rights (in practice, conditions are evaluated bottom-up until a rule with authorizer value "POLICY" evaluates to "true").

Signature - cryptographic signature verifying that it was the authorizer who wrote the rule (not shown in most of our examples). The signature field is required in all rules except for the "POLICY" rules which are read from file and implicitly trusted (unsigned rules are not accepted over the network).

The *condition rules* are simple logical expressions and may use the **and** operators "&&" and the **or** operators "||". Condition rules may use a combination of event attributes and external attributes. The availability of external attributes depends on the implementation. They can include current time, number of received events (by each subscriber), etc.

2.2. Access Control

We identify a number of actions that can be performed in our system. Each action has certain security implications and should be controlled through an access control policy. The supported actions are:

- **authenticate** - authenticate to the system
- **advertise** - introduce a new event type into the system
- **publish** - publish an event of a particular type
- **subscribe** - subscribe to an event of a particular type
- **receive** - receive an event
- **change policy** - modify the security policy (add/remove/modify rules)

We consider authentication to be outside of our security model. It is used only to positively authenticate users trying to perform one of the other actions. Authentication must be performed in order to enforce the security policy.

Each *owner* usually receives the right to publish, subscribe, receive, and change policy for the events/event types he owns. A KeyNote rule that would permit this for user *joe* is shown in Figure 3. Using that rule, user *joe* is being given the right to subscribe and receive his own events as well to manage policy for those events.

2.3. Delegation

One of the problems with some trust management systems is *delegation of rights*. The KeyNote system, which we use in our prototype, allows an entity to delegate any rights, that it possesses, to other entities. The owner *joe* can in turn authorize other users to perform certain actions. In figure 4, user

```
Authorizer: joe
Licensee: alice
Conditions: (app_domain == "LOC_APP") && (evtType == "LOC_INFO") &&
((action == "SUBSCRIBE") || (action == "RECEIVE")) &&
(owner == "joe") -> "true";
```

Figure 4. User *joe* grants the right to subscribe and receive his events of type *LOC_INFO* to user *alice*.

alice is authorized to subscribe and receive all events with attribute *user* equal to "joe".

From a privacy perspective, the ability of a user to grant all or subset of rights to another user, irrespective of the ownership of data, is not always desirable. In a location tracking application, for example, Joe might give Alice the rights to subscribe and receive his location events. But Joe may not want Alice to, in turn, pass these rights to another user.

Alice could simply send each event about Joe to another entity outside of the pub-sub system. It is beyond the scope of our system (and perhaps any software system) to control Alice's ability to leak the data received outside the system, or even to republish Joe's data with a false claim of ownership within the system. Alice would be a malicious user in that case and this is the classical problem of digital rights management, which we know is hard to do if users are malicious. However, we do want to provide a basic solution for Joe to make sure Alice cannot inadvertently allow others to receive his events within the system by simply adding a rule that grants such rights to others, as allowed by KeyNote.

In order to restrict Alice's ability to grant those rights, we provide the *change_policy* action. In order to allow a user to receive some events, Alice would have to enter a new policy rule. To do this, she would need the right to perform the *change_policy* action for events owned by Joe. If Joe does not grant the right to modify policy rules for his events to Alice, Alice will be unable to delegate the rights to receive Joe's events without Joe's permission (figure 4 shows Joe delegating the rights to subscribe and receive events but not to change policy).

2.4. Data Security Policy and Advertisements

An **advertisement** is a way for an authorized entity to introduce a new event type into the system. The access control policy is checked to make sure that the advertiser is authorized to perform the action. An advertisement describes the new event type and indicates the type of access control and data security required. An example of an advertisement is shown in figure 5. For brevity, we omit discussion of data security and granularity of security guarantees, which can be found in [18].

Types of access control

The following are the possible types of access control:

No-control - no access control is performed for events of this type. All users of the system are allowed to subscribe and to

```

Application: LOC_APP
Event type: LOC_INFO
Attributes: user:string
           building:string
           room:integer
Access control: receive-subscribe
Security: confidentiality, integrity
Granularity: matching_set

```

Figure 5. An advertisement for event type “LOC_INFO”

receive events of this type.

Subscribe-time - access control policy is checked whenever a new subscription for events of the particular type is entered. If allowed, the new subscription should be inserted into the subscription set, otherwise it is rejected. Since subscription requests are controlled through the access control policy, individual events are delivered to the interested (matching) subscribers without further access control checks.

Receive-time - access control policy is checked before events are delivered to interested subscribers. All users are allowed to enter subscription requests for events of this type. When an event is published and a matching process determines a set of interested subscribers, the access control policy is checked whether each subscriber in the matching set is allowed to receive the particular event.

The receive-time policy is useful when access control rules depend on the environment or other dynamic values external to the event itself. For example, an application may allow users to enter any subscription but may limit the number of events received by each subscriber to a particular number. Subscribers may also be limited to receive events during a particular time during the day or only during weekdays (and not on weekends). It is impossible to check these types of rules at subscribe time.

Receive-Subscribe-time - both subscription attempts and event receive attempts are controlled. This policy combines the subscribe-time and receive-time policies.

3. Prototype and Application

We have built a prototype content-based publish subscribe system to demonstrate the viability of our model and policy language as described in section 2. While not 100% complete, the current version implements most of the discussed features. This section describes our implementation as well as a sample application built on top of the pub-sub system. The application is a secure, privacy-aware, location tracking system where location sensors (RFID sensors) publish events whenever a *tag-wearing* person enter the sensor’s detection radius (the infrastructure for such system will be built into the new CSE building at the University of Michigan).

The current implementation only supports equality tests in the subscription language. It is straightforward to add additional operators to the subscription language and we are cur-

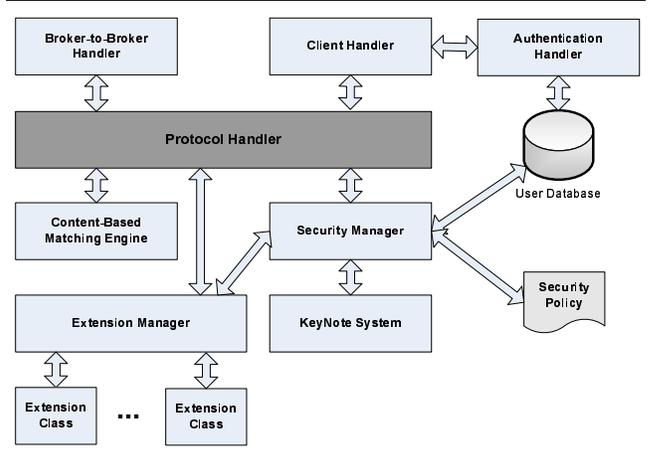


Figure 6. Broker Architecture.

rently implementing inequality operators. The matching algorithm is based on tree matching algorithms presented in [2, 3, 19].

The subscription language supports *wildcards*. For example, the subscription

```

(user=="joe" && building=="EECS" &&
room=="*")

```

specifies interest in “LOC_INFO” events where the *user* attribute is equal to “joe” and *building* attribute is equal to “EECS”. In other words, the subscriber is interested in tracking user joe anywhere in the EECS building.

3.1. System Architecture

Our pub-sub system consists of clients (publishers, subscribers, owners, administrators) and the event delivery system. The event delivery system is designed to consist of a network of event brokers. The events will be routed between brokers using a combination of algorithms described in [3, 19]. The current implementation, designed as a proof of concept for our security infrastructure, supports only one central broker.

A broker accepts client connections and performs requested actions. The event broker maintains a database of users who have signed-up to use the system. We assume that users sign-up off-line and that during that process they generate a public/private key pair and submit their public key to the pub-sub system. When connecting to the broker, clients must authenticate first. The broker can be extended to support any type of authentication protocol. A simple solution can be an *ssh-like* protocol based on the public/private key pairs generated when signing up for the service. Currently, we implemented a simple password-based authentication protocol.

Once authenticated, clients can request to add subscriptions, publish events, or add new policy rules (presumably to authorize other users to perform some actions). We assume that the communication between a client and a broker is en-

encrypted. This can be achieved by establishing a *session key* during authentication and using that key to encrypt all messages. An alternative solution is to use one of the caching algorithms described in [17] to improve performance.

The architecture of an event broker is shown in figure 6. The *client handler* is responsible for all communication with pub-sub clients. The client handler parses the message, determines the protocol type (**authenticate, publish, subscribe, change policy, etc.**), and calls an appropriate *protocol handler* method (in case of an authentication message, the client handler uses *authenticator* methods instead).

The *protocol handler* is the main part of the broker. It validates the message passed to it from client handler and decides what to do with it. In case of a valid message, the protocol handler checks with the *security manager* whether the requested action is allowed. If the security manager allows the action, protocol handler calls appropriate methods in the *matching engine* module. The security manager uses the *KeyNote System* [7] to determine whether the given action is allowed under the current security policy. This is done by constructing an *action query* (based on the parameters passed in from the protocol handler) and calls the appropriate KeyNote method. KeyNote, in turn, evaluates the action request in the context of current security policy and returns *true* if action is allowed or *false* if it is rejected.

The matching engine handles subscription requests by simply adding new subscriptions to the matching tree. In case of a publish request, the matching engine searches the matching tree to determine the set of all subscriptions matching the given event. Since subscriptions are annotated with subscriber id's, the search algorithm returns a list of *matching user id's* to the protocol handler. If the event requires access control check before sending it to the matched subscribers, the protocol handler must check, for every matching user, whether she is authorized to receive the event. This is done by querying the KeyNote engine separately for each matching user. We are forced to use this, rather inefficient, algorithm because of the KeyNote API which only allows querying action permission for one user at a time. We are currently working to extend the KeyNote system to return a list of authorized users (instead of true/false values for an individual user). This extension will allow us simply intersect the list of authorized users with the set of users matching the current event to obtain a set of *interested and authorized* users.

Sometimes, an event owner may want to authorize other users to perform an action based on attributes which are not part of the event schema. In the location tracking example above, assume that the user Joe wants to grant access to all of his location events but only during regular work hours. Since the event schema for LOC_INFO event type does not include time, it would be impossible to write such rule if we were only allowed to use event attributes. Another such example is if an Joe wanted to allow user Alice to receive his events but only once an hour

Our pub-sub system supports *external attributes* to enable users to write rules such as the ones described above. External attributes are attributes which are not part of the event schema but are added to the event before security policy is evaluated. This allows us to write policy rules which depend on attributes which are not included in the event at publish time. The *extension manager* is the module which determines whether external attributes should be added to a particular event. By convention, external attribute names have *ext* pre-pended to them.

For extension manager to work, we must implement a special *extension class* for each application domain. This extension class adds appropriate attributes to events of different types within the application domain. The extension manager has two important API calls: **processEvent()** which is called whenever a new event is published. This method allows the extension manager to keep track of different pieces of information, such as the number of events received by each user (as in the example above). This collected information is then used to fill in external attributes by the **addAttribs()** method. The addAttribs() API call is used before evaluating whether the current action is allowed under current policy.

3.2. Location Tracking Application

We have implemented a content-based messaging substrate with privacy policy support for the location tracking system as presented in figure 1. The application uses Radio Frequency Identification for Business (RFID) sensors deployed throughout the location tracking area (building, campus, city, etc.). Users of the system carry small RFID badges which are detected by the sensors. The sensors transmit their data to *location publishers* (one or more) which are clients of the pub-sub system. The location publishers convert the sensor data into pub-sub events of type *LOC_INFO*. The event schema for this event type is defined as follows:

```
[LOC_INFO: (user: String, building: String, room: String)]
```

The location publishers publish the events to the publish subscribe system. Users of the system can then subscribe to location information about other participants.

A privacy-aware location tracking application has to allow users to control the flow of information about them. It has to provide flexible policy language which allows users to express complex rules such as the ones mentioned above. The location tracking application implemented on top of our pub-sub system does just that.

To allow users to control availability of the sensor data about them, every user is granted ownership of events (Rule 4 in figure 7). To get around the problem of granting ownership individually to each user (as in figure 2), the value of *owner* in the Licensee field is derived from the event itself and passed to KeyNote during policy evaluation.

```

1 Authorizer: POLICY
  Licensee: location_admin
  Conditions: (app_domain == "LOC_APP") -> "true";

2 Authorizer: location_admin
  Conditions: (app_domain == "LOC_APP") && (evtType == "LOC_INFO")
  (action == "SUBSCRIBE") -> "true";

3 Authorizer: location_admin
  Licensee: location_publisher
  Conditions: (app_domain == "LOC_APP") && (evtType == "LOC_INFO")
  (action == "PUBLISH") -> "true";

4 Authorizer: location_admin
  Licensee: owner
  Conditions: (app_domain == "LOC_APP") && (evtType == "LOC_INFO") &&
  ((action == "RECEIVE") || (action == "ADD_ASSERT"))
  -> "true";

5 Authorizer: Bob
  Licensee: Alice || Eve || Nick
  Conditions: (app_domain == "LOC_APP") && (evtType == "LOC_INFO") &&
  (owner == "Bob") && (action == "RECEIVE") &&
  ((extTime == "WORK_DAY") || (extTime == "WORK_NIGHT")) -> "true";

6 Authorizer: Nick
  Conditions: (app_domain == "LOC_APP") && (evtType == "LOC_INFO") &&
  (owner == "Nick") && (action == "RECEIVE") && (extCollaborator == "true")
  -> "true";

7 Authorizer: Eve
  Conditions: (app_domain == "LOC_APP") && (evtType == "LOC_INFO") &&
  (owner == "Eve") && (action == "RECEIVE") &&
  (building == extBuilding) && (room == extRoom) -> "true";

```

Figure 7. Sample policy for the location tracking application “LOC_APP”.

Users can then delegate a subset or all of their rights (for example rights to receive events) to other users. By default, a user’s events are private and nobody but the user can receive them.

The right to **subscribe** to events is given universally to all users of the system. This is done to allow subscriptions such as:

```
[LOC_INFO: (user="*", building="EECS", room="2246") ]
```

where we want to know if anybody entered room 2246 in the EECS building. Since the event owner is not known at subscribe time, it would be impossible to decide which “subscribe” policy rules apply to this subscription request. Since everybody is allowed to subscribe to all events, users can write rules about who can actually receive the events. A small part of the policy is shown in figure 7 (note that we are omitting parts of the KeyNote language syntax for clarity).

Rule 1 allows *location_admin* to administer the “LOC_APP” application. Rule 2 allows all users to subscribe to events of type “LOC_INFO”. Rule 3 allows a special client, *location_publisher*, to publish events of type “LOC_INFO”. Rule 4 gives all users the ownership rights to events about themselves. Bob gives permission to Alice, Eve, and Nick to receive his events in rule 5. The receive right is only valid during *work days* and *work nights*. Nick authorizes all users who are his collaborators to receive his events in rule 6. The attribute *extCollaborator* is evaluated externally. Finally, Eve allows all users to receive her events but only if the subscriber and Eve are in the

```

Alice: [user = "Eve" && building = "EECS" && room = "**"]
       [user = "Bob" && building = "EECS" && room = "**"]
       [user = "Bob" && building = "GGBR" && room = "1005"]
       [user = "Sam" && building = "EECS" && room = "3115"]
       [user = "Tom" && building = "*" && room = "**"]
       [user = "*" && building = "EECS" && room = "2246"]

Eve:   [user = "*" && building = "*" && room = "**"]

Bob:   [user = "Alice" && building = "ATL" && room = "**"]

```

Figure 8. Few example subscriptions from the location tracking application.

```

user = Eve,      building = EECS,  room = 1005
user = Eve,      building = EECS,  room = 1003
user = Tom,      building = GGBR,   room = 1020
user = Alice,    building = ATL,    room = 133
user = Sam,      building = EECS,   room = 3227

```

Figure 9. Few example events from the location tracking application.

same room in the same building (rule 7).

We notice the use of external attributes **extTime**, **extBuilding**, **extRoom**, and **extCollaborator** in rules 5 - 7. The extension class *ExternalLocation* was implemented to add the current time and the location of the subscriber to the event attributes.

Figures 8 and 9 show a number of sample subscriptions and events from our location tracking system.

4. Conclusion and Future Challenges

We showed that the following types of access control and privacy policies can be formalized in our system:

- Where users wish to make their data available to only selected users.
- Where users wish to place computable conditions before making data available. Those conditions can be enforced at subscription time (to prevent users from even subscribing to data) or at receive time (to allow users to subscribe to data but potentially not receive it if it violates the predicate).
- Where users wish to control the ability of users to delegate the rights granted to other users.

Services such as privacy filters can be accommodated by treating them as both a publisher and a subscriber. As a subscriber, they have to be authorized by users to receive their events. As a publisher, however, the situation gets more complicated. We envisage two scenarios, one in which the service acts as the owner of filtered data and thus controls further dissemination. This would typically be true of services that aggregate data and can be dealt with in the framework to a degree by treating the service as an original source of the data.

The second scenario is that the service republishes the filtered event, but continues to associate the user with the event as far as the policy enforcement is concerned.

We presented a solution to controlling delegation of rights by placing restrictions on the ability of users to change policies. However, that solution has its limitations. It handles the situation well when policies are generally static. However, consider a scenario where a user Alice grants all rights to user Bob, including the ability to delegate rights. Bob now subscribes to Alice's events and also delegates those rights to Charlie. Unfortunately, Alice cannot simply modify her rule to take away `change_policy()` right of Bob to revoke Charlie's rights; such a change will only affect future policy update operations of Bob, not the past. From a privacy management perspective, a better solution would be to determine delegation rights dynamically. However, that appears to require a more general framework than KeyNote and we plan to explore this further in the future. With the current system, Alice could eliminate her rule that granted Bob the rights to receive her events. Since KeyNote evaluates rules by examining the entire delegation chain to the root, Charlie also would be unable to use his delegated rights to receive her events. But such a solution may be an overkill in some environments.

We would also like to provide support in our system to allow users to be prompted to "sign" a contract if they wish to subscribe to a user's data. Our plan is to make the contract be a part of the *Condition* in the policy rule (e.g., as a functional predicate `ContractSigned(contractDocumentURL)`), and motivated by the realization that not all aspects of privacy terms can be captured as a computable predicate and are better expressed in a legal framework or as understandings between the publisher and subscriber. It is also possible that a subscriber may wish a publisher to sign a contract before it will accept the publisher's data or provide a service to the publisher. Designing a solution for such scenarios in our framework requires further investigation.

Finally, we would like to apply our policies to not only real-time events but also to events that may be first archived in a database. A good and efficient solution for applying the policies to queries on archived events is not obvious. It is likely to require better integration of the policy framework with the query system on the database.

References

- [1] Mark S. Ackerman. General Overview of the P3P Architecture. MIT World Wide Web Consortium, 1997. <http://www.w3.org/TR/WD-P3P-arch>.
- [2] Marcos K. Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar D. Chandra. Matching Events in a Content-Based Subscription System. In *Proceedings of Principles of Distributed Computing (PODC '99)*, Atlanta, GA, May 1999.
- [3] Guruduth Banavar, Tushar Chandra, Bodhi Mukherjee, Jay Nagarajarao, Robert E. Strom, and Daniel C. Sturman. An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems. In *International Conference on Distributed Computing Systems*, June 1999.
- [4] A. Belokosztolszki, D. M. Eyers, P. R. Pietzuch, J. Bacon, and K. Moody. Role-based access control for publish/subscribe middleware architectures. In *Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*. ACM Press, June 2003.
- [5] Ken P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):36–53, December 1993.
- [6] Matt Bishop. *Computer Security: Art and Science*. Addison-Wesley, 2003.
- [7] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. The KeyNote Trust-Management System, Version 2, September 1999. Request For Comments (RFC) 2704.
- [8] Luis F. Cabrera, Michael B. Jones, and Marvin Theimer. Herald: Achieving a Global Event Notification Service. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Elmau, Germany, May 2001. IEEE Computer Society.
- [9] Antonio Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-area Networks*. PhD thesis, Politecnico di Milano, December 1998. Available from <http://www.cs.colorado.edu/~carzanig/papers/>.
- [10] Mary J. Culnan. Protecting Privacy Online: Is Self-Regulation Working. *Journal of Public Policy and Marketing*, 19(1):20 – 26, 2000.
- [11] Renee B. Ferguson. PreCache Unveils Net-Injector Platform. *eWeek*, January 2003. <http://www.eweek.com/article2/0,3959,808317,00.asp>.
- [12] R. Gruber, B. Krishnamurthy, and E. Panagos. An Architecture of the READY Event Notification System. In *Proceedings of the Middleware Workshop at the International Conference on Distributed Computing Systems*, Austin, TX, June 1999.
- [13] R. J. Harper. Why do and don't People wear Active Badges: A Case Study. *Computer-Supported Cooperative Work*, 4(4):297 – 318, 1995.
- [14] Jason I. Hong and James A. Landay. An architecture for privacy-sensitive ubiquitous computing. In *Proceedings of MobiSys 2004*, June 2004.
- [15] B. Krishnamurthy and D. Rosenblum. Yeast: A general purpose event-action system. *IEEE Transactions on Software Engineering*, 21(10), October 1995.
- [16] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The Information Bus - An Architecture for Extensible Distributed Systems. *Operating Systems Review*, 27(5):58 – 68, December 1993.
- [17] L. Opyrchal and A. Prakash. Secure distribution of events in content-based publish subscribe systems. In *Proceedings of the 10th USENIX Security Symposium*, pages 281–295, August 2001.
- [18] Lukasz Opyrchal. *Content-Based Publish Subscribe Systems: Scalability and Security*. PhD thesis, University of Michigan, Ann Arbor, 2004. Under preparation.
- [19] Lukasz Opyrchal, Mark Astley, Joshua Auerbach, Guruduth Banavar, Robert Strom, and Daniel Sturman. Exploiting ip multicast in content-based publish-subscribe systems. In *Proc. of Middleware 2000*, April 2000.
- [20] PreCache. <http://www.precache.com>.
- [21] Bill Segall and David Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of AUUG97*, Brisbane, Australia, September 1997.
- [22] Dale Skeen. Vitria's Publish-Subscribe Architecture: Publish-Subscribe Overview. Technical report, Vitria Technology Inc., 1996. <http://www.vitria.com>.
- [23] *TIBCO Messaging Solutions*. http://www.tibco.com/software/enterprise_backbone/messaging.jsp.
- [24] C. Wang, A. Carzaniga, D. Evans, and A. L. Wolf. Security Issues and Requirements for Internet-Scale Publish Subscribe Systems. In *Proceedings of the HICSS-35*, Big Island, Hawaii, January 2002.