CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2004; **16**:1099–1120 (DOI: 10.1002/cpe.806)

Access-controlled resource discovery in pervasive networks

Sanjay Raman[*,†], Dwaine Clarke, Matt Burnside,
Srinivas Devadas and Ronald Rivest

*Laboratory for Computer Science, Massachusetts Institute of Technology,
Cambridge, MA 02139, U.S.A.*

SUMMARY

Networks of the future will be characterized by a variety of computational devices that display a level of dynamism not seen in traditional wired networks. Because of the dynamic nature of these networks, resource discovery is one of the fundamental problems that must be solved. While resource discovery systems are not a novel concept, securing these systems in an efficient and scalable way is challenging. This paper describes the design and implementation of an architecture for access-controlled resource discovery. This system achieves this goal by integrating access control with the Intentional Naming System (INS), a resource discovery and service location system. The integration is scalable, efficient, and fits well within a proxy-based security framework designed for dynamic networks. We provide performance experiments that show how our solution outperforms existing schemes. The result is a system that provides secure, access-controlled resource discovery that can scale to large numbers of resources and users. Copyright © 2004 John Wiley & Sons, Ltd.

KEY WORDS: access control; access control list; pervasive computing; intentional naming network; public key infrastructure

1. INTRODUCTION

Resource discovery is one of the fundamental challenges that must be faced in the context of pervasive computing. While resource discovery is vital to enabling operation in pervasive networks, their unpredictability and dynamism give rise to problems of security. As a resource provider, we want to guarantee that foreign users that enter our environment will not be able to act maliciously. Similarly, as a user in a foreign environment, we want to know what resources we are able to use and which ones we can trust. Such access restrictions are easily handled in fixed networks as foreign users can simply be denied admission to the network. But the fundamental notion behind pervasive computing gives rise to the idea of resources and users of varying privileges interacting in the same environment [1].

[*]Correspondence to: Sanjay Raman, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, U.S.A.
[†]E-mail: sraman@abp.lcs.mit.edu

Received March 2003
Revised June 2003
Accepted June 2003

While several systems [2–4] propose resource discovery solutions for dynamic environments, they do not consider how the integration of security protocols influences scalability and performance.

Resource discovery systems are typically implemented in the network layer, below security, allowing networks to overlay any desired security protocol. An access control framework can be layered over a resource discovery protocol, but these two protocols seem to have different goals. The problem is that the best criteria-matching resource (e.g. 'the nearest, least-loaded printer') may not necessarily be a resource to which a user has access.

The primary focus of this paper is to address the issue of resource discovery in a pervasive computing environment. More specifically, this paper presents a system that integrates access control with resource discovery in order to enable scalable and efficient operation. This paper describes a resource discovery system that is scalable and efficient and is designed to elegantly integrate with a larger proxy-based security system [5].

The proxy-based security system uses a distributed SPKI/SDSI protocol [6], which allows for private, encrypted communication between heterogeneous lightweight devices in a pervasive computing environment. In this architecture, each resource has an associated trusted software proxy whose primary function is to execute commands on behalf of the resource it represents. Proxies store certificates and other secure information for the resource they represent. Two security protocols are utilized: a computationally inexpensive protocol for resource–proxy communication and a sophisticated SPKI/SDSI access control framework layered over a key-exchange protocol for resource authorization between proxies.

The architecture presented in this paper makes four key contributions.

- A scalable model for resource discovery based on the Intentional Naming System (INS) [7] that integrates access-control information with service information.
- Integration of access-controlled resource discovery with a proxy-based security infrastructure to provide secure and authentic communication in a pervasive computing environment.
- Implementation of resource lookup that makes access control decisions *while* finding the best resource.
- Design of lightweight, efficient access control lists.

Section 2 gives the background for this research. We summarize the resource discovery problem in terms of a simple scenario in Section 3. Section 4 details our system architecture and describes how we have developed an access-controlled resource discovery system. We present the advantages and performance evaluation of our system in Section 5. We conclude the paper in Section 6.

2. BACKGROUND

The resource discovery system presented here is designed to be an integral part of a larger proxy-based security architecture [5]. Resources are defined to be any piece of hardware or software that is providing a service to members of the network. A resource may be location-aware. In this architecture, each resource has an associated trusted software proxy. A proxy is software that runs on a network-visible computer and its primary function is to execute commands on behalf of the resource it represents. Proxies store certificates and other information for the resource they represent and are trusted implicitly. Proxies communicate with each other using a protocol based on SPKI/SDSI (proxy–proxy protocol).

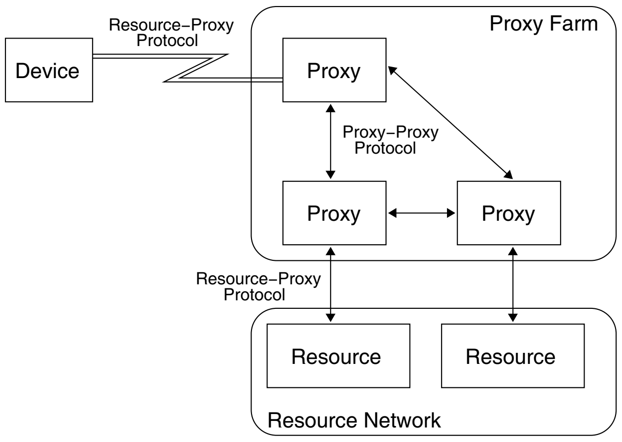

Figure 1. An overview of the basic components in the proxy-based security infrastructure. Proxies communicate via the proxy–proxy protocol. Devices and resources communicate with proxies via the resource–proxy protocol.

A separate resource–proxy protocol is used for secure communication between resources and proxies. Having two different protocols allows us to run a computationally-inexpensive security protocol on impoverished resources and a sophisticated protocol for resource authorization on the more powerful resources and proxies. Figure 1 shows an overview of this system and the protocols it uses. This system is used as the foundation for the secure resource discovery system that we develop here.

3. THE PROBLEM RESTATED

The problem that the system in this paper solves is that of how to scale a system of resources that are protected by access control. It is tremendously inefficient if users repeatedly attempt to contact a resource that they are prohibited from using. One only has to consider an environment with a large number of protected resources. If users have no knowledge of which resources they can access, it could take an exhaustive computational effort to find an accessible resource. In order to gain scalability and efficiency, the resource discovery system needs to know about access control privileges so that it can return the best resource to which *a user has access*. By knowing the user's authorizations (i.e. the groups to which the user has membership) and the access control lists of the supported resources, a resource discovery system can effectively meet this goal.

3.1. A simple scenario

This issue is especially pertinent when dealing with networks where the state of the network is highly dynamic. It is very plausible that users will not know exactly what resources are available, nor will they know which resources they are authorized to use. As a simple example, consider an environment

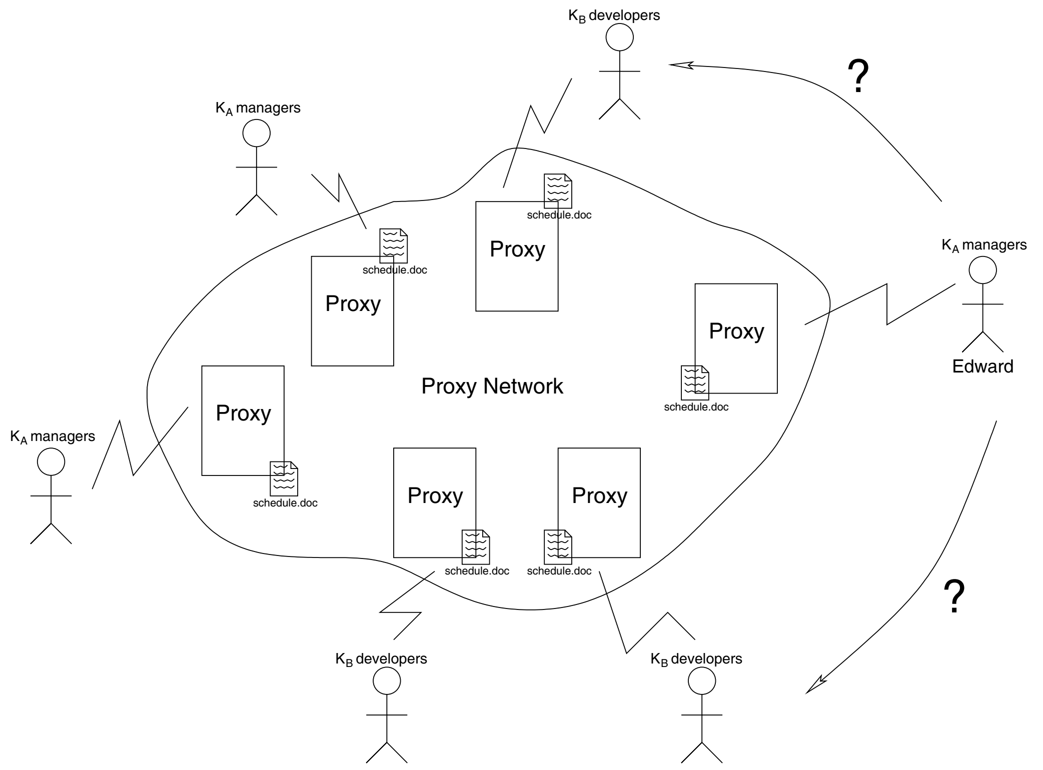

Figure 2. This figure illustrates the conflict experienced by a resource discovery system in an access-controlled environment. How does Edward find the closest, accessible copy of `schedule.doc` without needing to perform an exhaustive search?

which treats all devices in the network as resources in a peer-to-peer application. Figure 2 illustrates the following scenario:

> Edward, a manager at a large software firm, arrives in the morning at a conference with his location-aware device. Upon arriving and coming online, Edward wants to download his personalized conference schedule for the given day. At this conference, there are two tracks: one for managers and one for software developers. Thus, the users in the system are divided into two groups, K_A `managers` and K_B `developers`. All the users already at the conference have a document, `schedule.doc`, in their repository, but the document is track-specific. That is, the copy of `schedule.doc` that members of K_A `managers` have is different than the copy that members of K_B `developers` have. When Edward comes online, he wants to synchronize his copy of `schedule.doc` by getting the latest version. The conference is spread out over several buildings and the users are spatially far apart. Because the physical area of the conference is large, there is no central repository

for the schedules. Instead, schedule distribution and synchronization happen peer-to-peer. As a member of K_A `managers`, Edward must get the document from another member of *his group*. Members of K_A `managers` do not have access to the schedules of members of K_B `developers`, and vice versa. Edward would also like to get the schedule from the geographically closest user, in order to minimize his delay and make the synchronization process as fast as possible.

3.2. Problems

This scenario creates a conflict of interests. Not only must Edward find the closest user, but he must also find a user that is in his group (a resource to which he has access). A simple resource discovery system could easily tell Edward the location and identity of the closest user. This problem has been solved many times over [7–9]. But how does Edward know if the physically closest user is a member of his group? And, if this user is not a member of his group, where exactly is the closest member of Edward's group? Mobility of the users only further complicates the issue. It would be time consuming and inefficient for Edward to blindly search for the closest member of his group. The only way in which a resource discovery system can identify the closest, accessible resource is to know ahead of time Edward's identity and authorizations.

3.3. A naïve solution

Before presenting our solution, it is instructive to outline a naïve solution. This solution will be used as a baseline of comparison in terms of performance and will be important for the analysis of Section 5. Resource discovery systems that do not incorporate the ideas presented in this paper will typically operate by returning the address(es) of the best criteria-matching resource, without accounting for a user's access privileges. It will become clear that issues of scalability and efficiency are major obstacles with such a system.

In attempting to discover the geographically closest user, Edward will query the resource discovery system through his personal proxy. The proxy will tell the resource discovery system to 'find me the closest user'. Ideally, Edward would like to contact the closest, *accessible* user, but this resource discovery system does not know anything about Edward's identity or authorizations. In response to the query, the resource discovery system will return a list of the geographically closest users to Edward's proxy. At this point, Edward's proxy does not know which of the resources in the list are accessible to him. The only reasonable way for the proxy to proceed is to sequentially iterate through the resources in the list in the hope that they are accessible. The proxy must engage in some sort of authorization check in order to determine if the user has access to the resource. As long as a contacted resource fails, the proxy will have to repeat the process.

This approach can be inefficient and surely is not scalable. If a given user has access to every resource in the network, then the efficiency of access control is not an issue. But, in most heterogeneous environments, users are assumed to be diverse and access privileges will exhibit the same differences. In Edward's scenario, if he is not close to any users of his group, he would have to iterate through many inaccessible resources before finally finding a match. Edward is faced with executing a process on the order of $O(n)$ if there are n other resources in the network. The results of Section 5 will illustrate this point.

4. SYSTEM ARCHITECTURE

A better approach would be to give the resource discovery system knowledge about the access control lists that protect the resources. We require that the designed system be secure, efficient, scalable, and robust. In order to meet our goals, the INS [7] was selected. The solution presented here uses several modifications to intentional naming that enables access control decisions to be made while finding the best resource. Before detailing our solution, we summarize INS as a standalone resource discovery system.

4.1. Intentional naming overview

INS is a resource discovery and service location system intended for dynamic networks. INS is ideal for dynamic networks because an application only needs to tell the service the resource characteristics it is seeking. Since the availability of resources may be dynamic, these systems require a naming service that is just as flexible. The Domain Naming Service (DNS) works well for static networks since an application can be fairly confident in the names of resources. INS provides users with a layer of abstraction so that applications do not need to know the availability or exact name of the resource for which they are looking. A simple example of a user's request in INS is to find the nearest, least-loaded printer. DNS would require the user to know the exact name of the resource, such as `pulp.lcs.mit.edu`.

INS uses a simple language based on expressions called *name specifiers*, which are composed of an attribute and value. An attribute is simply a category by which a resource can be classified. For example, a camera in the system can be described by its `resolution`, `battery-life`, and/or `available-memory`. An INS name, or *intentional name*, is a hierarchy of these atomic name specifiers. An example of an INS name is `[service=camera [resolution=640x480] [battery-life=87%] [available-memory=56mb]]` to describe a camera with the specified properties.

INS is composed of a network of *Intentional Name Resolvers (INRs)* that serve client requests for resources and maintain information about the searchable metadata of each resource. Data are represented in the form of a dynamic *name-tree*, which is a data structure used to store the correspondence between name specifiers and the destination resource. The structure of a name-tree strongly resembles the hierarchy of a name specifier. Name-trees consist of alternating levels of attributes and values, with multiple values possible at each attribute. A particular name specifier is resolved by traversing the tree, making sure to visit all the corresponding attribute–value pairs of the target resource. Each leaf value in the name-tree has a pointer to a *name-record*, which holds the physical location of the resource. Figure 3 illustrates an example name-tree.

4.2. Security integration with INS

The solution presented here uses several modifications to intentional naming that enable access control decisions to be made while finding the best resource. While INS does allow for a security framework to be layered over it, we have already seen how a system can benefit from integrating access control decisions with resource discovery. INS is extended in the following three ways to provide access-controlled resource discovery:

1. implementation of a real-time maintenance of the access control lists in the INS name resolvers;
2. introduction of a certificate-based authorization step during resolution of an INS request; and

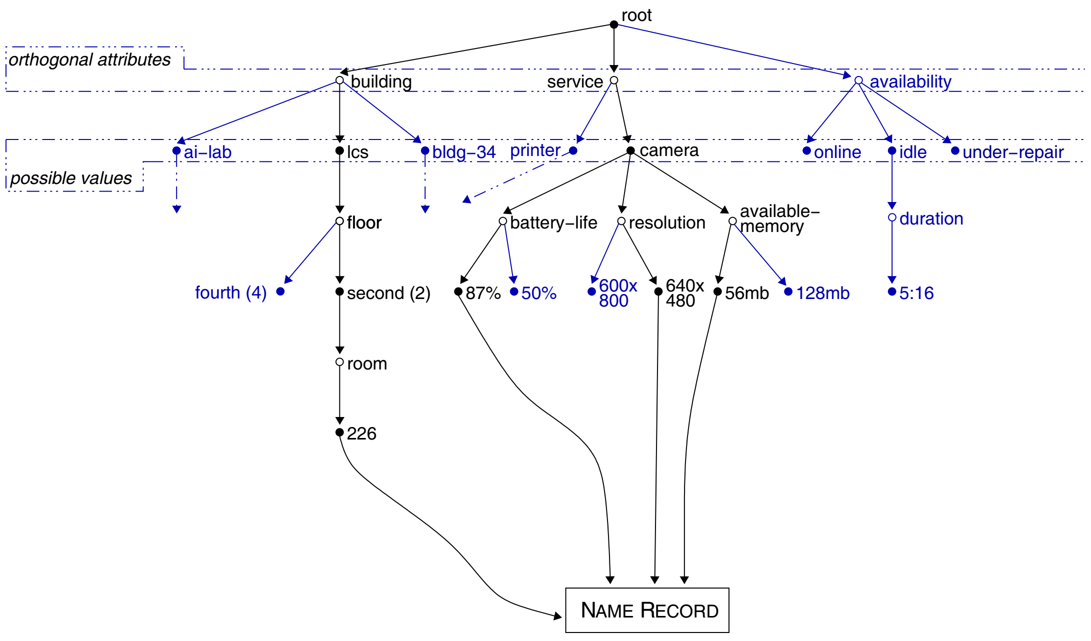

Figure 3. A graphical view of an example INS name-tree. The name-tree consists of alternating layers of attribute-nodes, which contain attributes and value-nodes (possible values). Value-nodes contain pointers to all the name records to which they correspond.

3. design of a lookup algorithm that prunes the possible name records by eliminating resources based on a user's identity and authorizations.

In the following sections, the key extensions of INS are presented. Finally, we will return to the scenario that is discussed above to see how this new system integrates access-control information and INS knowledge to efficiently return the best, accessible resource. Another key factor influencing this design was its inclusion as a small piece of a larger security infrastructure. Therefore, some of the components of this design were chosen to leverage the existing functionality of the proxy-based security system [5].

4.2.1. Storage of ACLs in INS

Assuming that resources have the ability to inform INS of the access control lists that protect them, how can these lists be properly stored in the INS knowledge base so that they can be referenced when making resource decisions? INS uses a name-tree to store its knowledge about resources in the system. Name-trees are dynamic, changing their structure based on how resources advertise and re-advertise themselves to INS.

An access control list is treated as an additional attribute that defines a resource. One can specify a `camera` based on its `resolution`, say; similarly, an access control list is just another way to classify the camera. In order to store ACLs as attribute–value pairs, a new type of attribute was introduced. Previously, all attributes were treated as *searchable*, in that they were used as a dimension along which a resource can be explicitly queried. But, when a user makes a request for a resource, the user cannot specify the ACL attribute–value pair in the query. Nor do we want the ACLs being represented as additional branches in the name-tree. So, in order to store ACLs, the concept of a *hidden* attribute was defined. INS attributes are now defined as searchable or hidden, with the only hidden attribute being that of the ACL. When advertising its service profile, a resource will advertise its ACL like any other searchable attribute, but the name resolvers are responsible for denoting the ACL as a hidden attribute and storing it on the name-record for the particular resource.

Storing ACLs as attribute–value pairs is advantageous because we do not change the manner in which data are stored and we do not have to radically alter the way in which queries are handled (Section 4.2.2). The structure of the name-tree remains the same, while the hidden attributes are stored directly on the name-records for each resource.

4.2.2. *Redesign of lookup algorithm and ACL propagation*

Adjie-Winlop *et al.* [7] describe the LOOKUP-NAME algorithm that INS uses to retrieve name-records for a given name-specifier. This algorithm operates by pruning attribute branches of the name-tree that fail to match the given search criteria, ultimately arriving at a subset of all the name-records that contains the possible matching resources. This algorithm works well with the way name-trees are organized in INS. Since the name-trees consist of alternating levels of attributes and values, it is very easy to prune branches of the tree while progressing through the target name-specifier. But, left alone, this algorithm fails to work with hidden attributes such as ACLs.

Due to the transparency that is required, users will not explicitly construct queries with ACL name-specifiers. One option for determining a user's accessible resources would be as follows. First, the LOOKUP-NAME algorithm would be run to completion, arriving at a list of criteria-matching resources. At this point, INS would have a handle to the name-records for each of the matching resources. We could proceed by iterating through these possible name-records and checking whether the user making the request is on the ACL. While this approach will save us considerably over the approach of contacting each of the resources for access decisions, it still is inefficient. A closer inspection of the LOOKUP-NAME algorithm reveals additional ways in which this process can be optimized.

We designed a modified algorithm, LOOKUP-NAME-AC (illustrated in Figure 4), that eliminates potential name-records *while* pruning S, the set of all possible name-records. The LOOKUP-NAME-AC algorithm operates under some assumptions on the state of the INS name-tree. In order for the algorithm to terminate successfully, the algorithm assumes that each value node in the name-tree contains an intermediate ACL. This intermediate ACL is computed to be the logical OR ($\vee$) of the intermediate ACLs stored at all of the value nodes that are its children in the INS name-tree. Beginning at the value-nodes that contain pointers to name-records, intermediate ACLs are computed. For these leaf nodes, the intermediate ACL is simply the ACL of the name-record to which it points. After computing the ACLs at these leaf nodes, the intermediate ACLs for the parent nodes are computed all the way up the name-tree. OR'ing ($\vee$) multiple access control lists happens at the 'entry' level. That is, the result of the logical OR ($\vee$) of two ACLs is a new ACL with every entry that exists in either of the two ACLs.

```
LOOKUP-NAME-AC($T$, $n$, $k$) {
    $S \leftarrow$ the set of all possible name-records
    if !($T$.acl contains $k$)                              ▷ acl check
        return
    end
    for each av-pair $p$ :=($n_a$,$n_v$) in $n$
        $T_a \leftarrow$ the attribute-child of $T$ such that $T_a = n_a$
        if $T_a$ == null
            continue
        end
        if $n_v == *$                                       ▷ wild card matching
            $S' \leftarrow \emptyset$
            for each $T_v$ := value-child of $T_a$
                if $T_v$.acl contains $k$                   ▷ acl check
                    $S' \leftarrow S' \cup$ NAME-RECORDS($T_v$)
                end
            end
            $S \leftarrow S \cap S'$
        else                                                ▷ normal matching
            $T_v \leftarrow$ value-child of $T_a$ such that $T_v = n_v$
            if IS-LEAF($T_v$) || IS-LEAF($p$)
                if $T_v$.acl contains $k$                   ▷ acl check
                    $S \leftarrow S \cap$ NAME-RECORDS($T_v$)
                end
            else
                $S \leftarrow S \cap$ LOOKUP-NAME-AC($T_v$, $p$, $k$)
            end
        end
    return $S \cup'$ NAME-RECORDS($T$)
}
```

Figure 4. This algorithm looks up the name-specifier n in the name-tree T for the user k and returns all accessible matching resources.

For example, if $acl_a = [e_1, e_2, e_3]$ and $acl_b = [e_1, e_2, e_4, e_5]$, then

$$acl_a \vee acl_b = [e_1, e_2, e_3, e_4, e_5] \tag{1}$$

where the notation $acl = [e_1, \dots, e_n]$ indicates that $e_1, \dots, e_n$ are entries of the ACL. Figure 5 illustrates how ACLs are propagated up the INS name-tree from the leaf nodes.

The modified algorithm is similar to its predecessor, except now it eliminates candidate records based on whether the user is included in intermediate ACLs. This new algorithm takes the user's identity and

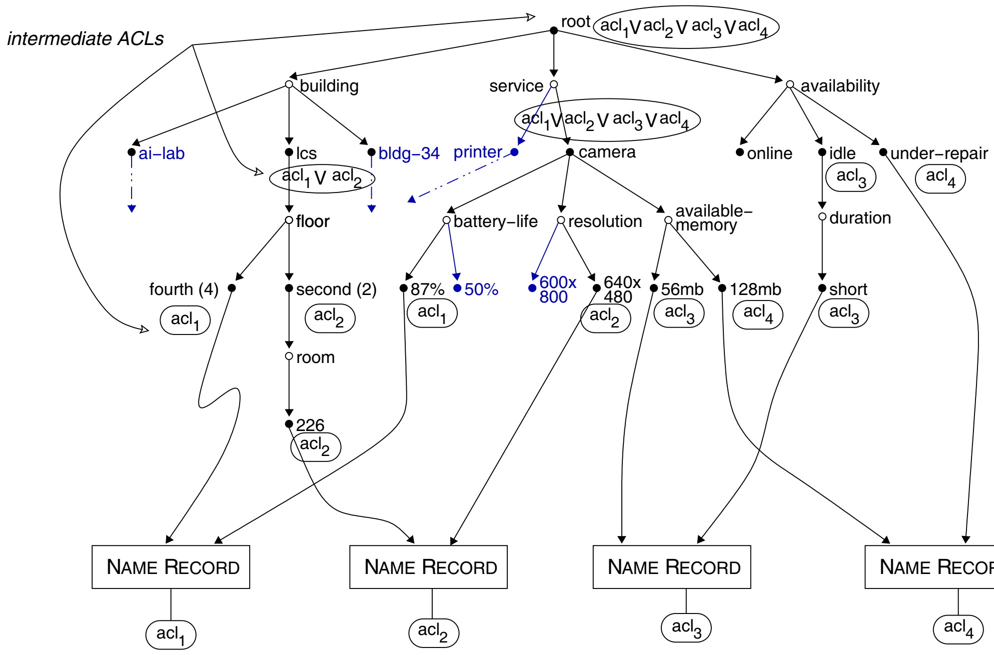

Figure 5. This figure shows how ACLs are propagated from the leaf nodes up the INS tree to the root of the data structure. At each intermediate value-node in the tree, an ACL is stored and is computed by taking the logical OR ($\vee$) of the ACLs at all of the child nodes.

authorization rules as arguments. For each name-specifier in the INS query, INS will prune branches that do not match the search criteria and that do not contain the user in their intermediate ACLs through a series of recursive calls. When the algorithm terminates, it returns only the relevant, accessible name-records. By taking the OR of the ACLs, we enable access control decisions to be made while INS is locating the proper name-record, eliminating the need to iterate through inaccessible resources and branches of the tree. This simplifies the task of the lookup algorithm as well as potentially reducing the amount of the name-tree that needs to be traversed. This algorithm terminates without the need to backtrack and does not ever check a given ACL more than once. The additional cost of this algorithm, though, is clearly in these checks that the algorithm must make for each name-specifier, but we argue in Section 5 that this tradeoff is still advantageous.

4.2.3. Dynamic maintenance of name-trees

ACLs are resource properties that may change. Groups or keys may be added or removed, or the operations allowed by a particular group/key may be changed. In dealing with name-tree maintenance, there are three qualities that any design should achieve.

- *Freshness.* Our primary goal is to keep the state information in the INS name-tree *fresh*. At any point in time, we want to know with high probability that the access control and resource information is up-to-date.
- *Responsiveness.* The maintenance procedure should be *responsive* to changes made to the access control information. It is important that changes to ACLs are rapidly reflected in the INS knowledge base.
- *Authentication and privacy.* Finally, maintenance updates should be *authentic* and *private*. There are a whole suite of attacks that can be centered around unauthentic updates (replay, DoS). Also, entities should not be able to maliciously learn sensitive information about resources. For these reasons, the security of maintenance updates is very important.

Many of these issues have been considered when designing INS for service updates, so our focus is specifically on how access control updates are handled. Responsiveness is achieved by using triggered updates which are fired when an ACL changes state. Periodic updates are also used to enforce freshness. The utility of these updates comes from the fact that ACLs typically have expiration times. Clearly, the update period should be chosen such that it is less than the ACL expiration time ($T_{\mathrm{update}} < t_{\mathrm{expire}}$) but not so small that it unnecessarily floods the network with update packets. Upon receiving an update request, INS actively modifies its name-tree to reflect the current state of access rights and intermediate ACLs are recomputed. Handling the privacy and authenticity of these messages, as well as the authenticity of messages in which a resource updates INS with its other service attributes, is a subject of ongoing research.

4.2.4. User authorization rules

In order for this system to function, INS needs access to the user's set of current authorizations. The modified lookup algorithm depends on knowing the user's identity and the groups of which he is a member. Each proxy in the system stores a user's signed SPKI/SDSI certificates. Clarke *et al.* [10] describe an efficient algorithm for determining, from a set of SPKI/SDSI certificates, the access control groups of which a particular user is a member and the operations that he is allowed to perform. Complete and detailed descriptions of the procedures are found in [10], but this is well beyond the scope of this paper. In essence, a (finite) transitive closure is taken over the certificates, and rules representing the user's authorizations are extracted. The rules are simple and not signed. However, each rule has a representation as a signed user certificate, or a chain of signed user certificates. The closure algorithm is run when there is a change in the user's certificates, such as when he acquires a new certificate, or when one of his certificates expires.

The proxy presents the user's authorization rules to INS with the user's query. INS uses the rules to check if the user is on an (intermediate or leaf) ACL contained at a node in the INS tree (using the LOOKUP-NAME-AC algorithm). An important point is that these ACL checks performed by INS can be made fast and efficient. The ACL check is used to determine *if* a user is on an ACL, and it is not necessary for INS to know the proof that the user would generate to show that he is on the ACL.

When INS has completed its searching and returned an address, the proxy will then use a secure authentication and authorization protocol to contact the resource [5]. The modified INS system we present now returns only resources to which the user has access, so the proxy should only have to execute this security protocol once.

4.3. The scenario revisited

After presenting the components of the access-controlled resource discovery system, it is helpful to revisit the scenario presented in Section 3.1 to see how this new system handles the same problem.

Again, Edward is looking to obtain a copy of his schedule, `schedule.doc`, from the closest user in his group. Edward places a request for the document via his proxy. Edward does not explicitly have to indicate to his proxy his group membership or the fact that he wants to retrieve the document from another group member; this is handled automatically by his proxy. Edward's proxy contacts an INR with which it has previously registered. It then queries the INR for the best accessible resource, translating the request specified by Edward to an INS-specific name-specifier. Edward's proxy also computes his authorization rules (they may be computed on the fly or pulled from the proxy's cache) and sends them along with the request to the INR. The INR, which has received access control advertisements from all the registered resources in the network, takes the request and the user's authorizations and executes the LOOKUP-NAME-AC algorithm. After a single execution of this algorithm, the INR returns the closest, accessible resource to Edward's proxy. Edward's proxy then uses a secure protocol to contact the resource and uses a standard secure copy protocol to retrieve the file from the resource. Because INS knew about Edward's group membership, it returned a resource that is accessible, meaning the time-consuming security protocol *would only have to be executed once*.

5. EVALUATION

A prototype system was implemented in Java using INS 2.0, a pure Java implementation of INS. In this section, a formal evaluation of this system is presented. The overall goal is to quantify how our design outperforms a resource discovery system that does not integrate access control with resource decisions. These experiments were all conducted using off-the-shelf Intel Pentium II 266 MHz computers with a 512 KB cache and 128 MB RAM, running Windows NT Server 4.0. The software was built and run using Sun's Java Virtual Machine version 1.3.

5.1. Comparison of resource retrieval time

A measure of the time savings of our solution is necessary to evaluate its effectiveness. As a baseline, this system will be compared with a basic scheme, where INS is used as the resource discovery system, but *does not* have access to ACLs or the authorizations of the requester. This basic scheme was described in detail in Section 3.3. For convention, we will assume that the user, U, is operating in a network with n total resources.

To understand the performance gains of this new solution, we must analyze the time it takes U to successfully access the most optimal resource and compare this time in both the *basic* and *access-controlled* systems. This time is denoted as t_{BASIC} for the basic scheme and as t_{AC} for the access-controlled system. Each of these time values can be generally expressed by the following equation:

$$t_x = t_{\mathrm{query}} + \left(\sum_{k=1}^{n} b_k \cdot (t_{\mathrm{latency}} + t_{\mathrm{acl\text{-}check}}) \right) + t_{\mathrm{crypto}} \qquad (2)$$

t_{query} is the *query time*, the time it takes the resource discovery system to respond to U's request. t_{query} also includes any time U's proxy uses to prepare the request. b_k is a Boolean value, which is `1` if U contacts resource k and `0` if U does not. t_{latency} is the round-trip network latency between two proxies. This is essentially the time it takes U to retrieve a resource's ACL over the network. $t_{\text{acl-check}}$ is the *ACL-check time*, the time it takes for a simple ACL check to be performed. ACL checks were made very fast with our adopted implementation (as will be shown later in this section). Finally, t_{crypto} is the time it takes U to derive the full authorization proof and for this proof to be verified by a particular resource's proxy.

5.1.1. t_{BASIC}

In the basic scheme, the time for U to successfully access the most optimal resource is given by the following equation:

$$t_{\text{BASIC}} = t_{\text{query}_{\text{BASIC}}} + \frac{1}{p} \cdot (t_{\text{latency}} + t_{\text{acl-check}}) + t_{\text{crypto}} \tag{3}$$

This derivation of t_{BASIC} can be found in [11]. $t_{\text{query}_{\text{BASIC}}}$ is the time it takes the LOOKUP-NAME algorithm to execute and p is the probability U has access to a given resource.

5.1.2. t_{AC}

Similarly, the time to retrieve a resource using our access-controlled solution is given by:

$$\begin{aligned} t_{\text{AC}} &= t_{\text{query}_{\text{AC}}} + t_{\text{latency}} + t_{\text{crypto}} \\ &= t_{\text{query}_{\text{BASIC}}} + D_n \cdot t_{\text{acl-check}} + t_{\text{latency}} + t_{\text{crypto}} \end{aligned} \tag{4}$$

The key difference is that t_{AC} is not dependent on the likelihood that U has access to a given resource. Instead, the query time, $t_{\text{query}_{\text{AC}}}$, depends on D_n, which represents the number of ACL checks that will have to be made while traversing the INS name-tree. It is a function of the number of resources in the network (n), but is also affected by the complexity of the name-tree and name-specifiers. For more details, see [11].

5.1.3. Name lookup performance, $t_{\text{query}_{\text{BASIC}}}$ *and* $t_{\text{query}_{\text{AC}}}$

To quantify the difference between $t_{\text{query}_{\text{BASIC}}}$ and $t_{\text{query}_{\text{AC}}}$, we constructed a large, random name-tree and timed how long it took the tree to perform 1000 random lookups using each algorithm. The name-tree and name-specifiers were chosen uniformly according to the parameters defined in [7] ($r_a = 3, r_v = 3, n_a = 2$, and $d = 3$). n, the number of distinct, unique names in the tree, was varied from 1 to 13 000 in increments of 100 to see how $t_{\text{query}_{\text{BASIC}}}$ and $t_{\text{query}_{\text{AC}}}$ vary with increasingly large name-trees. The maximum heap size of the JVM was limited to 64 MB, thus limiting the range of the experimentation.

Figure 6 shows the results of this experiment. Using the basic LOOKUP-NAME algorithm, the performance went from a maximum of around 700 name lookups/s to a minimum of 200 lookups/s. From Figure 6, it is evident that as the number of names in the name-tree increases, the lookup rate decreases. As a result, the amount of time required for a single lookup increases. But, the drop-off is not

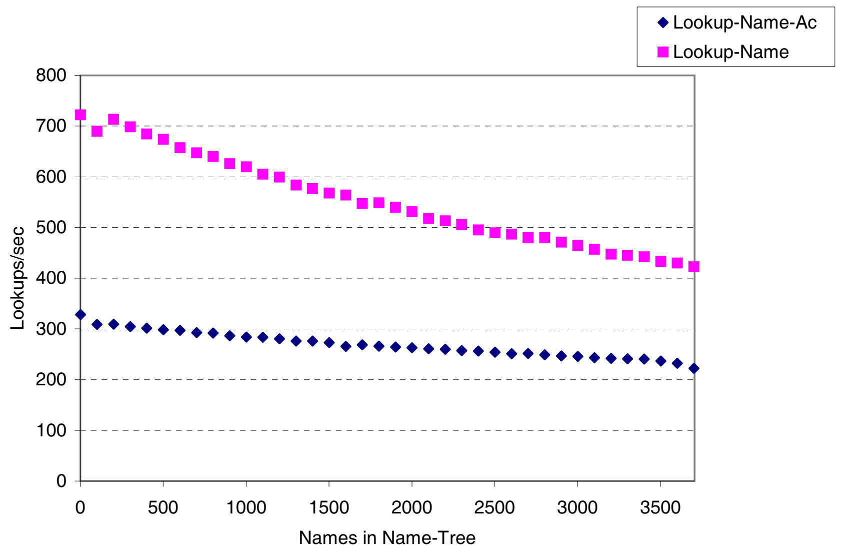

Figure 6. A comparison of the lookup rates for the two algorithms as the number of names in the name-tree varies. The lookup rate (lookups/s) is plotted against the number of names in the name-tree. For the standard case, the lookup rate progressively gets smaller, starting from a maximum of around 700 lookups/s to 200 lookups/s. In the access-controlled case, each name in the name-tree is protected by an ACL with 10 unique keys. As the number of names increases, the lookup rate progressively gets smaller, starting from a maximum of around 325 lookups/s to 240 lookups/s.

as drastic as one would think and clearly is not linear. For a moderately large system with approximately 2000 resources (or names), the average lookup time is around 1.8 ms. For small systems consisting of hundreds of resources, the lookup time is around 1.4 ms. These times are small and the difference in lookup times between the small and large systems is minimal.

The experiment was repeated in the access-controlled case. Each resource was initialized with ACLs containing 10 unique entries and the intermediate ACLs were computed. Figure 6 presents the performance results of the LOOKUP-NAME-AC algorithm as the number of names in the tree varied from 1 to 3500. As is evident from this figure, the lookup rate is significantly reduced from the rate without the ACL checks. The experiment was terminated at a maximum of 3500 names due to memory constraints of the JVM. With approximately 100 name-records in the tree, a rate of 325 lookups/s was achieved. In the non-access-controlled case, this rate was much higher at around 700 lookups/s. At approximately 3500 name-records, the rate of the LOOKUP-NAME-AC algorithm was at 240 lookups/s, indicating only a drop of about 90 lookups/s. Conversely, the rate in the basic case dropped to 450 lookups/s with 3500 names, indicating a drop of 250 lookups/s.

Table I details the average lookup times for the two algorithms for varying sizes of the name-tree. The difference between the lookup times is on the order of a few milliseconds and can be attributed directly to the intermediate ACL checks that are made. In the following section, it will be shown

Table I. The average lookup time experienced by the two algorithms for varying sizes of the name-tree.

Names in name-tree	Average lookup time (ms)	
	LOOKUP-NAME-AC	LOOKUP-NAME
100	3.24	1.45
200	3.23	1.47
500	3.35	1.48
1000	3.52	1.61
1500	3.66	1.76
2000	3.80	1.88
2500	3.94	2.04
3500	4.23	2.31

that $t_{\text{acl-check}}$, the time for a simple ACL check, is approximately 0.07 ms. Based on the name-trees we used during the experimentation, we can calculate approximately 15 intermediate ACL checks. This roughly accounts for a $15 \times 0.07 \approx 1.05$ ms difference between the lookup times. The data in Table I seem to support this back-of-the-envelope calculation.

5.1.4. *ACL performance, $t_{\text{acl-check}}$*

One of the fundamental differences between a basic solution and ours is the use of ACL checks during the name-lookup process. In order to determine the cost of an ACL check, random large ACLs were constructed with the number of distinct entries in the ACL ranging from 1 to 14 000 and the number of ACL checks that could be executed in the span of a second was measured. Figure 7 illustrates the results of this experiment. As expected, as the number of entries in the ACL grows, the ACL check rate decreases logarithmically. ACLs in our system are represented by red-black trees (binary trees), keyed by the users' public keys, that guarantee a $\log(n)$ time cost for adding new indices and looking up values. As the number of entries in the ACL goes from 1 to 1000, the check rate decreases by 500 check/s. A similar rate decrease can be seen as the number of entries is varied from 1000 to 10 000.

Figure 7 shows five stratified regions of lookup rates that correspond to the number of decisions that must be made in order to find a key in the ACL. Depending on where a key is located in the range of possible keys, the number of decisions to find it in the tree can vary. For an ACL of 1000 entries, the time it takes to perform an ACL check can be one of the following values: 0.083 ms, 0.074 ms, 0.067 ms, or 0.061 ms (according to the four different regions in the graph). These values are an order of magnitude smaller than the time taken by the LOOKUP-NAME algorithm to find a name. Therefore, the idea of making several ACL checks during the name retrieval process adds a minimal time cost and seems very reasonable.

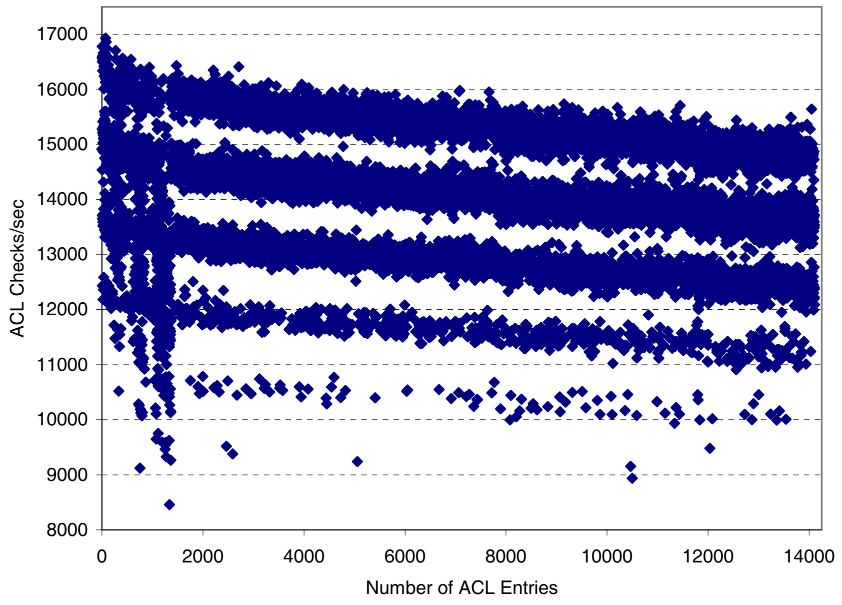

Figure 7. The ACL check rate (in ACL checks/s) is plotted against the number of entries in the ACL. It is evident that the rate decreases with an increasing number of entries, but only slightly. Also of note are the four regions of concentration of points.

5.1.5. *Round-trip network latency,* t_{latency}

t_{latency} is the round-trip network latency between proxies in the network. It is a fundamental component of the resource retrieval time in the basic solution (t_{BASIC}), which requires a client proxy to explicitly contact potential target proxies in order to determine access privileges. To estimate this parameter, simulations were run in ns [12].

A precise measure of t_{latency} is somewhat subjective, as the exact value of the round-trip time between two proxies depends on the network infrastructure, the number of hops between proxies, the current traffic conditions, the link bandwidth, and any additional network characteristics. For simulation purposes, we adopt a network structure where proxy–proxy communication will take at most two hops. Two routers are linked together, with each router containing seven end proxies, each sending packets according to the following traffic flows. A third router is connected to INS and the other two routers. Therefore, in order to communicate with another proxy, a proxy must only send packets through two hops. The links between proxies and routers each have a bandwidth of 133 Mbps and a propagation delay of 5 ms. The router–router links have a bandwidth of 100 Mbps. There are three main traffic flows in this network, namely Proxy–Proxy traffic, Proxy–INS service updates, and Proxy–INS requests. A single proxy–proxy flow was started between two proxies and the round-trip time (CRTT) for each packet was measured over a span of 30 s. Figure 8 shows the results of this experiment.

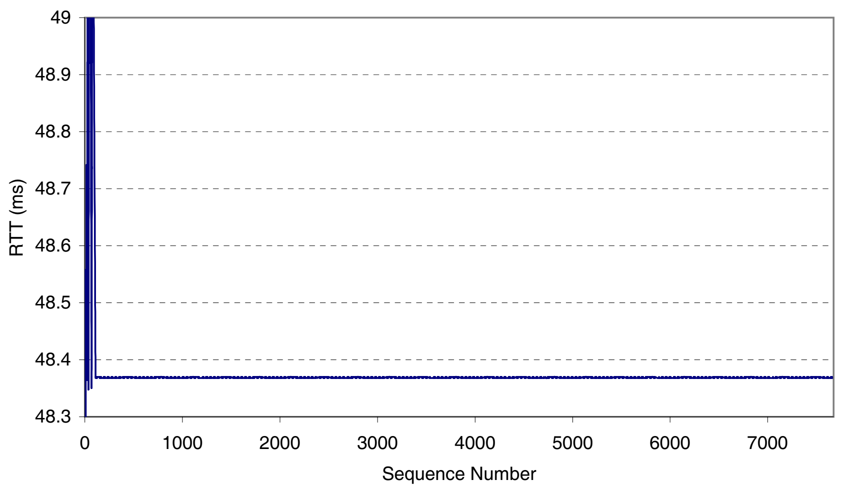

Figure 8. The results of the proxy–proxy latency simulation are shown here. The RTT of packets sent between proxies is constant throughout the packet flow. The mean RTT between two network proxies is 48.37 ms.

As is evident from this figure, the RTT stays almost constant throughout the duration of the traffic flow. Initially, there is some variance as TCP uses a slow-start mechanism to find the optimal window size. But, after equilibrium is reached, the mean RTT of proxy–proxy communication is 48.37 ms. It is worth noting that in a network with many resources, this number is a best-case scenario. The link bandwidths used were large, the propagation delays were small, and the two-hop assumption will break down as the number of resources increases. Despite using favorable conditions, we see that t_{latency} is three full orders of magnitude larger than $t_{\text{acl-check}}$. As will be shown in Section 5.1.6, this result plays a key role in determining the efficiency of our access-controlled resource discovery system.

5.1.6. t_{AC} *versus* t_{BASIC}

In this section, we analyze the difference in retrieval times between the two solutions. Subtracting Equation (4) from (3), we get

$$\begin{aligned}\Delta_t(n) &= t_{\text{BASIC}}(n) - t_{\text{AC}}(n)\\ &= \frac{1}{p}(t_{\text{latency}} + t_{\text{acl-check}}) - (D_n \cdot t_{\text{acl-check}} + t_{\text{latency}})\end{aligned} \tag{5}$$

From Equation (5), we can see that whether the access-controlled scheme outperforms the basic scheme depends on whether $\frac{1}{p}\cdot(t_{\text{latency}}+t_{\text{acl-check}})$ is greater than $(D_n\cdot t_{\text{acl-check}}+t_{\text{latency}})$. If it is, we can conclude it is more efficient for INS to perform the ACL checks as it descends down its name tree, rather than leaving this up to the user's proxy. In order to make this comparison, we consider our scenario (in Section 3.1) with 1000 total users divided equally among the two groups (K_A `managers` and K_B `developers`). Therefore, the probability that Edward has access to any given resource is $p = 0.5$.

If we also assume the structure of the name-tree is as described previously, $D_n = 15$. From our experiments in Section 5.1.4, we will assume an ACL check with 1000 entries per ACL takes 0.083 ms. Finally, the latency between proxies will be assumed to be 48.37 ms (as calculated in Section 5.1.5). Using these parameters, the difference in lookup time is

$$\begin{aligned}\Delta_t(n) &= \frac{1}{0.5}(48.37 + 0.083) - (15 \cdot 0.083 + 48.37)\\ &= 47.291 \text{ ms}\end{aligned} \tag{6}$$

Even with the parameters chosen to favor the basic solution, the access-controlled solution wins by a large margin. It is likely that this is a conservative estimate. With 1000 resources in the network, t_{latency} will likely be greater than 48.4 ms as the propagation delays of the links will increase and the number of hops between proxies will increase. Furthermore, if p becomes smaller, the basic solution is subject to more trips across the network, making our savings greater. The main difference in the resource retrieval times for each solution can be attributed directly to the fact that ACL checks are extremely fast. Our solution is not subject to the network latency and the three orders of magnitude saved in performing an ACL check give our solution a clear advantage. The query time saved in the basic solution is minimal compared with the time that the ACL checks save.

5.2. Performance of ACL propagation

The LOOKUP-NAME-AC algorithm requires that intermediate value nodes in the name-tree have computed the logical OR of all the ACLs in its subtree. In order to do this, the `propagateAcls` method is called periodically (for freshness) and any time a triggered update is initiated by a user's proxy.

The `propagateAcls` method is invoked every time an update to an ACL occurs. For analysis purposes, the time between ACL updates is denoted $\epsilon_{\text{triggered}}$. Immediately after an ACL update occurs, the `propagateAcls` method must be called. Since the method is synchronized, the name routers cannot service any incoming requests during this time, backlogging requests in a queue. This creates somewhat of a 'time-slotted' service model (as shown in Figure 9), where the requests can only be serviced between the end of the execution of `propagateAcls` and the time the next update arrives[‡]. Essentially, the INR can serve requests for some time, update itself, serve requests, and so on. Clearly, the goal here is to minimize the maintenance time with respect to the available service time so that the service slots are much bigger compared to the maintenance slots.

This 'slotted' model can potentially lead to problems, because users will not stop sending requests when the INR is under maintenance. A queue will build up as the name-tree is under maintenance and the requests in the queue along with all other requests must be processed before the next update arrives, or the system will experience congestion and collapse. If we model the queue as an M/M/1 queue [13] with Poisson arrivals and exponential service times, a formulation can be made as to when operation of the system will be successful (i.e. no collapse). The arrival rate of INS requests is λ. The service rate

[‡]In reality, there are other maintenance updates that the INRs handle, such as changes to service profiles (e.g. growth in the number of documents in a particular printer's queue, or a particular speaker going offline, etc.). But for the simplicity of this analysis, these updates are ignored here. The argument presented can be easily extended to account for these updates as well.

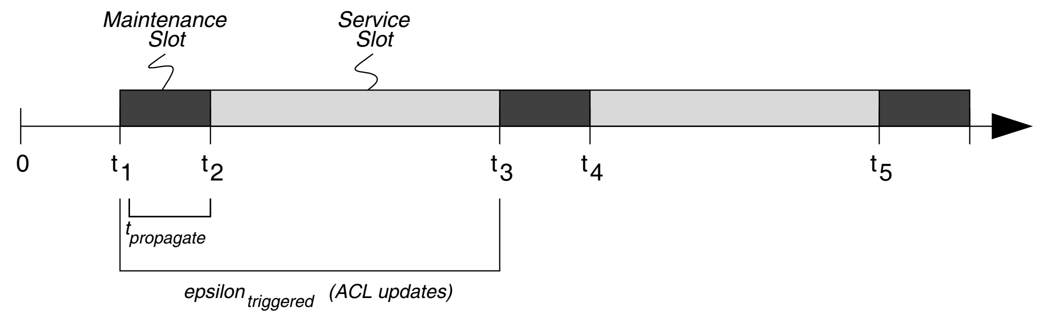

Figure 9. The `propagateAcls` method must be called after an ACL update has been sent to the INR. Since the method is synchronized, new requests cannot be serviced while the method is being executed. Servicing of requests can only occur between the execution of `propagateAcls` and the next update.

is μ (the average service time is $1/\mu$). In this system, $1/\mu$ equals $t_{\mathrm{query}_{\mathrm{AC}}}$. The time for the execution of `propagateAcls` is $t_{\mathrm{propogate}}$. The collapse condition will occur if all the requests are not serviced before the next ACL update arrives. While the INR is under maintenance, we expect N_{Q}, the queue size, to grow to $\lambda t_{\mathrm{propagate}}$ (by Little's Theorem [13]). Similarly, while the INR is in the service slot, the number of incoming requests will be $\lambda(\epsilon_{\mathrm{triggered}} - t_{\mathrm{propagate}})$. The time to service these requests must be *less than* the duration of the service slot in order for queue buildup to be avoided. That is

$$\epsilon_{\mathrm{triggered}} \gg t_{\mathrm{propogate}} \cdot \frac{\mu}{\mu - \lambda} \qquad (7)$$

Is it a reasonable assumption that this condition holds? We have seen that $t_{\mathrm{propogate}}$ is on the order of a few seconds and $1/\mu$ is on the order of a few milliseconds. According to Equation (7), it can be seen that as long as the INR is not receiving requests at the same frequency (every few milliseconds), then the system will be fine. Even if λ is on the order of a request/ms, the frequency of ACL changes will be in minutes, not milliseconds. The condition in Equation (7) will easily hold and the system operation will be smooth.

5.3. Tradeoffs

The premise our solution makes is that basic solutions scale poorly and are based on inefficiencies that limit the performance of the system. Specifically, finding a resource requires explicit contact to check access privileges. From the experiments, we have verified this by showing our solution significantly reduces the resource retrieval time. At the same time, it makes a large system with many resources manageable and efficient. A similarly sized system may be inoperable under the basic resource discovery approach.

While saving time, our solution does add greater requirements for storage to INS. This is primarily driven by the need to store ACLs (both resource-level and intermediate) in the name-tree, a constraint not made necessary by the basic solution. We analyze the additional space required by our solution and compare it with that of the basic solution.

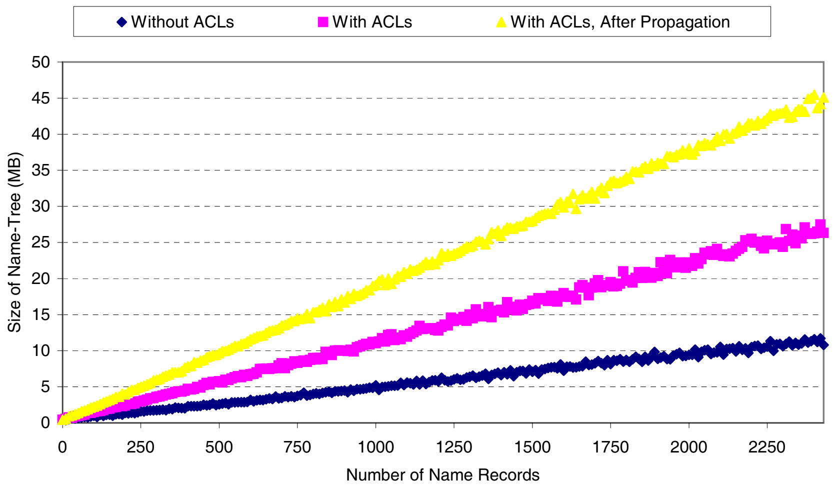

Figure 10. There are three plots shown here. The first plot shows the memory used by standard name-trees without ACLs. The second plot shows the same name-trees with ACLs on the records. The final plot shows the name trees with the ACLs after the `propagateAcls` method has been called (the name-trees contain intermediate ACLs). As the number of name-records increases, the memory used increases linearly.

Ultimately, the goal is to evaluate how much *more* memory the access-controlled solution requires than the basic solution. Again, using randomly constructed name trees (with the same dimensions as described in prior experiments), the number of name-records in the tree was varied from 0 to 2500 and the physical size of these trees was measured. As a base case, the name-trees were created without access control lists. These are the name-trees used by the basic solution. As a second experiment, the name-trees were then created with each name-record containing an ACL with 10 unique entries. The size of these trees was measured. The final experiment involved measuring the size of the name-trees after the `propagateAcls` method was called. The `propagateAcls` method forces the computation of the intermediate ACLs, so the memory used increases. Figure 10 shows the results of these experiments.

From this figure, we can clearly see that the name-trees required by the access-controlled solution use much more memory than the basic name-trees. A name-tree of 2000 name records that does not store ACLs uses approximately 9.4 MB of storage, whereas a name-tree that stores ACLs and has executed its propagation takes up 38.1 MB. In fact, from the figure, ACL-propagated name-trees use 3.75 times more space, on average, than basic name-trees:

$$\begin{aligned} size(T_{\mathrm{AC}}(n)) &= 3.75 \cdot size(T_{\mathrm{BASIC}}(n)) \\ &= 3.75 \times [(0.0185 \cdot n) + 0.40]\ \mathrm{MB} \end{aligned} \tag{8}$$

For relatively small name-trees, this difference is not substantial. But, as the number of name-records grows fairly large, the difference in the name-tree sizes is significant. The computational resources required to store the name-trees become large as the system scales. As the number of name-records grows, the sizes of the intermediate ACLs also grow accordingly. Note, this is the worst case scenario. Even though each ACL has 10 different entries, it is very possible that entries can be repeated across resources, thereby somewhat limiting the size of the trees. Despite this fact, integrating access-control into INS requires additional memory in the name routers.

Nevertheless, storage is cheap and can be solved simply by adding more memory to each INR. However, saving time is not as simple as installing additional components to each router. As such, the storage-time tradeoff is one that is worth making.

6. CONCLUSION

This paper has experimentally verified the merits of our resource discovery system that integrates access control by comparing it with alternative systems. The resource retrieval time is greatly reduced using this architecture, while security is not compromised. This allows our system to scale to levels that traditional resource discovery systems wishing to implement access control would be unable to efficiently reach. While the implementation and execution of this system does require additional memory in each intentional name router, sacrificing storage for time and efficiency is a worthwhile tradeoff.

Together with the proxy-based security model, this architecture meets all the goals of a secure system. It features efficient and scalable access-control for all resources while integrating with a powerful resource discovery system. We believe that this architecture is a flexible and generalized security infrastructure ready to support the pervasive computing trends that will surely dominate the future.

ACKNOWLEDGEMENTS

This work was funded by Acer Inc., Delta Electronics Inc., HP Corp., NTT Inc., Nokia Research Center, and Philips Research under the MIT Project Oxygen partnership, and by NSF under the contract CCR-0208631.

REFERENCES

1. Banavar G, Beck J, Gluzberg E, Munson J, Sussman J, Zukowski D. Challenges: An application model for pervasive computing. *Proceedings of ACM MOBICOM*, August 2000.
2. Eronen P, Nikander P. Decentralized Jini security. *Proceedings of the Network and Distributed System Security Symposium (NDSS 2001)*, San Diego, CA, February 2001.
3. Hewlett-Packard. CoolTown. http://cooltown.hp.com.
4. Berkeley UC. The Ninja Project: Enabling Internet-scale Services from Arbitrarily Small Devices. http://ninja.cs.berkeley.edu.
5. Burnside M, Clarke D, Mills T, Maywah A, Devadas S, Rivest R. Proxy-based security protocols in networked mobile devices. *Proceedings of ACM SAC02*, March 2002; 265–272.
6. Rivest R, Lampson B. SDSI—a simple distributed security infrastructure. http://theory.lcs.mit.edu/~rivest/sdsi10.ps.
7. Adjie-Winoto W, Schwartz E, Balakrishnan H, Lilley J. The design and implementation of an intentional naming system. *Operating Systems Review* 1999; **34**(5):186–301.
8. Martin DL, Cheyer AJ, Moran DB. The open agent architecture: A framework for building distributed software systems. *Applied Artificial Intelligence* 1999; **13**(1/2):91–128.
9. University of Washington. Portolano: An expedition into invisible computing.

10. Clarke D, Ellen J-E, Ellison C, Fredette M, Morcos A, Rivest R. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security* 2001.
11. Raman S. A secure framework for access-controlled resource discovery in dynamic networks. *Master's Thesis*, Massachusetts Institute of Technology, 2002.
12. Fall K, Varadhan K. *The ns Manual*. http://www.isi.edu/nsnsam/ns/ns-documentation.html.
13. Bertsekas DP, Gallagher R. *Data Networks* (2nd edn). Prentice-Hall: Englewood Cliffs, NJ, 1991.