Location-aware Access Control for Pervasive Computing Environments

by

Nikolaos Michalakis

B.S., Electrical Engineering and Computer Science, MIT, 2002

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2003

© Nikolaos Michalakis, MMIII. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author ....
Department of Electrical Engineering and Computer Science
February 4, 2003

Certified by ....
Stephen Garland
Principal Research Scientist, Laboratory for Computer Science
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Location-aware Access Control for Pervasive Computing Environments

by

Nikolaos Michalakis

Submitted to the Department of Electrical Engineering and Computer Science
on February 4, 2003, in partial fulfillment of the
requirements for the degree of
Masters of Engineering

Abstract

In pervasive computing environments certain applications are interested in a user's location in order to provide a service. Such applications would benefit from an architecture that enables users to prove their location prior to requesting a service. We present *PAC*, an architecture for location-aware access control in pervasive computing environments, where users authenticate their location in order to gain access to resources. PAC preserves user anonymity and uses lightweight security. We evaluate our architecture with respect to its security and its scalability as the number of resources and users increase.

Thesis Supervisor: Stephen Garland
Title: Principal Research Scientist, Laboratory for Computer Science

Acknowledgments

First, I would like to thank Prof. Hari Balakrishnan for discussing the idea of location authentication using the Cricket beacons with me that lead to this thesis. I would like to thank my thesis supervisor Dr. Stephen Garland for his patience for reading and re-reading the numerous thesis drafts that I produced and his valuable feedback during this work and his help on simplifying the system. Also special thanks to Bodhi Priyantha and Dorothy Curtis for their assistance with the Cricket hardware and software. I dedicate this work to my wonderful parents, Mirsini and Konstantinos Michalakis. Their support has always been invaluable and I hope I never stop making them proud.

Contents

1 Introduction **15**

1.1 Motivation . . . 15

1.2 Background . . . 17

1.2.1 SecurID . . . 17

1.2.2 Kerberos . . . 18

1.2.3 Access Control Lists . . . 19

1.2.4 Cricket . . . 20

1.2.5 INS/Twine . . . 20

1.3 Related Work . . . 21

1.3.1 Cooltown . . . 22

1.3.2 E-biquity-Centaurus2 . . . 23

1.3.3 Gaia-Mist . . . 23

1.3.4 Proxy-based Authentication . . . 23

1.4 PAC Overview . . . 25

1.5 Goals . . . 26

2 Infrastructure **29**

2.1 PAC Pseudo-Random Number Generator . . . 29

2.2 LIDCODE . . . 30

2.3 Beacons . . . 31

2.3.1 The LID . . . 31

2.3.2 Beacon initialization . . . 32

2.4 Location Groups . . . 32

2.4.1 Location Group management 33
2.5 LIDAuthority 34
2.6 Service Agent 36
2.6.1 Service Agent Set-up 36

3 Protocols 39
3.1 Synchronization Protocol 39
3.2 Access Request protocol 42
3.2.1 Ticket Request message 44
3.2.2 Ticket Response message 45
3.2.3 Access Request message 45
3.2.4 Access Response Message 46
3.3 Location Authentication Protocol 46
3.4 Ticket Verification Protocol 47
3.5 Continuous Operation Policies 48

4 Building the system 51
4.1 Cricket modifications 52
4.2 LIDAuthority implementation 53
4.3 Service Agent implementation 54
4.4 Client implementation 54

5 System Enhancements 57
5.1 Distributing a LIDAuthority 57
5.2 Grouping the LIDAuthority and Service Agent 58

6 Security Evaluation 61
6.1 Pseudo-Random Number Generator Analysis 61
6.1.1 Seed prediction 62
6.1.2 Seed computation 62
6.2 LIDAuthority/Service Agent Synchronization 63
6.3 LIDCODE proxies 63

6.4 LIDCODE playback 64
6.5 Denial of Service 64
6.6 Stealing Tickets 65
6.7 Beacon issues 65
6.8 Secure communication 65

7 Conclusion **67**

List of Figures

1-1 How a user accesses a service using PAC. 25
1-2 (a) Location information sent to the service via the client. (b) Location information sent via the gateway. 28

2-1 The PAC-PRNG block diagram. 30
2-2 The LIDCODE. 31
2-3 The PAC-PRNG block diagram. 35

3-1 LIDCODE Buffer update when $n' > n$. 41
3-2 LIDCODE Buffer update when $n' < n - 1$. 42
3-3 The Access Request protocol diagram. 1: Ticket Request, 2: Ticket Response, 3: Access Request, 4: Access Response (the dotted line indicates that step is optional). 43
3-4 Continuous operation: the client renews tickets to continuously access a service. The renewals can take place independently from the client-service interaction. 49

5-1 1. LIDAuthority and Service Agent share a symmetric key. 2. LIDAuthority and Service Agent merge. 59

List of Tables

1.1 An example of an ACL 19

1.2 An example of a `/etc/hosts.allow` file 20

2.1 The PAC-PRNG algorithm 30

2.2 Entry format of LID-LIDCODE table 35

2.3 Entry format of LID-LGroup table 35

3.1 The LIDCODE buffer 40

3.2 Ticket request message format 44

3.3 Ticket format 45

3.4 Access request message format 46

3.5 Access response message format 46

4.1 LIDAuthority performance analysis. 53

4.2 Service Agent performance analysis. 54

4.3 Client performance analysis. 55

Chapter 1

Introduction

1.1 Motivation

In a pervasive computing environment it is not always desirable for a user to authenticate her identity in order to access services. Some applications are interested in the user's context (such as location, orientation, etc.) rather than her identity in order to interact with her.

In this work, we focus on access control based on location authentication. In general, users, devices and the physical environment are represented and identified by bits, such as ASCII strings, digital signatures, images, sounds, etc. The fundamental problem that a location-aware access control system for pervasive computing environments must solve is to authenticate a user's location, since any mechanism for access control requires an authentication scheme. This means that a user with a binary representation U must prove that she is in the physical location with binary representation L at time t by providing a proof $P(U, L, t)$ in digital form.

For example, if U is the image of the user's face and L is an image of the building entrance, the user can authenticate her location by taking a photo of her in front of the building entrance, marking it with the current time, and sending it to an authentication authority. The authority uses the photo, U, L and its clock as the proof $P(U, L, t)$ to verify the fact that U is currently at L. If the user wants to remain anonymous, then she would not include herself in the photo. In such a case

the proof would be of the form $P(X, L, t)$, where X cannot be used to identify U. After the user's location has been authenticated the system can allow access to various computing services.

Several scenarios illustrate why such a system is useful:

- An individual grants access to her room devices only from her home and office.
- A restaurant customer requests a song to be played by the speakers near her table.
- A museum website allows access to a tour guide application only to visitors inside the building.
- A company grants access to certain resources only to those inside the company building.
- A network printer allows only users on the printer's floor to print files.
- A location-aware network file system authenticates a user based on where she is to allow access to shared files.

For these scenarios, checking identity to control access doesn't completely solve the problem and is too heavyweight for a solution even when it does. First, checking identity involves authentication (either password, biometrics or a Public Key Infrastructure (PKI)) for potentially large numbers of users that interact with the system for short periods of time. Second, in some scenarios, identity not only is not important but must be kept secret. Third, in some applications, it is difficult to associate identity with location. Even if we authenticate a user's identity, it might still be difficult or impossible to authenticate her location.

In this thesis we explore the problem of location authentication in pervasive computing environments, and we present an architecture for Pervasive Access Control (PAC) that is based on location awareness, is lightweight, is scalable, and preserves anonymity.

1.2 Background

There exist many methods that authenticate a user's identity [37], based on something the user knows (e.g., password), has (e.g., token) or is (e.g., biometrics).

PAC is inspired by time-varying token authentication techniques such as SecurID. Location Authentication uses a ticket based protocol similar to Kerberos. Access is controlled using Access Control Lists (ACLs) per service. PAC uses the Cricket system for location discovery and INS/Twine for service discovery.

In the following sections we give a review of SecurID, Kerberos, ACLs, Cricket and INS/Twine.

1.2.1 SecurID

SecurID token-based authentication is popular for logging remotely into corporate networks [3]. This is because it provides twice the level of security compared to password based authentication, since the user is required to provide both something she knows (password) and something she has (token random number).

In the SecurID system a user holds a SecurID *token card*, which is a device that produces a different number every time it is used. A token card is initialized by providing it with a random seed. The same seed is also provided to a central authentication server, where a "mirror" number sequence is produced.

Without knowing the seed, the sequence of numbers produced is unpredictable, because it is produced by a cryptographically strong Pseudo Number Generator (PRNG).

When a user logs in remotely to a server she enters her password along with the current number that the token displays. The server compares the token number provided by the user to the one it has produced locally. If there is a match then the user is authenticated.

1.2.2 Kerberos

Kerberos [25] is a ticket-based network authentication service widely used today. It is an example of the Needham-Schroeder key distribution protocol [30].

The components of the Kerberos system are the following:

- *Principal:* A principal is an entity that is capable of authenticating itself. Every principal has a symmetric *secret key* in order to prove its identity.
- *AS:* The Authentication Server (AS) knows every principal's secret key. A user attempts to authenticate herself by obtaining a Ticket Granting Ticket (TGT). The AS sends a TGT to the user's workstation encrypted with the user's secret key (password). The user provides her password to decrypt the TGT and use it. The TGT is stored at the workstation for further use.
- *TGS:* The Ticket Granting Service (TGS) receives the TGT from a user and, if the TGT is valid, it returns service tickets to the user to access *kerberized* services.[1]

A client authenticates itself to a service using the following protocol [2]:

First, the client obtains a TGT from the AS.

1. C $\rightarrow$ AS: TGT request.
2. AS $\rightarrow$ C: $\{K_{c,tgs}, \{T_{c,tgs}\}K_{tgs}\}K_c$, where $K_{c,tgs}$ is a session key to use with the TGS generated by the TGS and $T_{c,tgs}$ is the TGT. The message is encrypted with the client's key, K_c.

The client obtains a ticket for the service.

3. C $\rightarrow$ TGS: S, $\{T_{c,tgs}\}K_{tgs}, \{A_c\}K_{c,tgs}$, where S is the name of the requested service and A_c is an authenticator to guard against replay attacks. The authenticator is a bit string used by the client to prove its identity. It includes C and a time-stamp.

[1] services that are accessed through Kerberos.

[2] We use A $\rightarrow$ B to mean "A sends message to B" and $\{M\}K$ to mean "message M encrypted with key K"

4. TGS → C: $\{K_{c,s}, \{T_{c,s}\}K_s\}K_{c,tgs}$, where $K_{c,s}$ is a session key with S and $T_{c,s}$ is a ticket to access S.

The client requests the service.

5. C → S: $\{T_{c,s}\}K_s, \{A_c\}K_{c,s}$

6. S → C: The service checks the validity of the ticket and the authenticator and grants access.

One thing to note about Kerberos is that the AS and the TGS do not need to be different entities, since the only purpose of the AS is to remember a user's Kerberos key. In fact, in earlier versions of Kerberos, the Key Distribution Center (KDC) performed the tasks of both the AS and TGS. Also, since TGTs are stored in a workstation, the workstation cannot be used by multiple users since the TGT can be stolen. It must be removed after the user leaves the workstation.

1.2.3 Access Control Lists

Access Control Lists (ACLs) are used by operating systems to grant or deny access to files in the file system [39]. An ACL is associated with a specific file. Every entry in the list typically contains a user-name and the access rights (read, write, execute) that this user has upon the file. In addition to user names, ACLs also contain group names. Table 1.1 illustrates a typical ACL.

$filename$	
$user_1$	read
$user_2$	read, write
$group_1$	read

Table 1.1: An example of an ACL

The use of ACLs is not restricted to file systems only. ACLs can be used by services in general to grant or refuse access to users. ACLs combined with wild-cards and a set of logical set rules (AND, OR, NOT, ALL, NONE) can be used to create powerful access filters, while keeping the size of the ACL small. For example, remote

access to services in the UNIX operating system is controlled by Host-Based ACLs that are described in the `/etc/hosts.allow` and `/etc/hosts.deny` files [2]. Table 1.2 shows an example of such a file.

hosts.allow
all remote.domain hosts are allowed to access all # services except the untrusted.remote.domain host ALL: *.remote.domain EXCEPT untrusted.remote.domain # host 18.234.0.37 is allowed access to all services # except ftp ALL EXCEPT ftpd: 18.234.0.37

Table 1.2: An example of a `/etc/hosts.allow` file

1.2.4 Cricket

Cricket [31, 32] is indoor location system for pervasive computing environments. It uses a combination of RF and ultrasound technologies to provide a location-support service to users and applications. Wall- and ceiling-mounted beacons are spread through the building, publishing information on an RF signal operating in the 418 MHz AM band. With each RF advertisement, the beacon transmits a concurrent ultrasonic pulse. Listeners attached to devices and mobiles listen for RF signals and, upon receipt of the first few bits, listen for the corresponding ultrasonic pulse. When this pulse arrives, they obtain a distance estimate for the corresponding beacon. The listeners run maximum-likelihood estimators to correlate RF and ultrasound samples (the latter are simple pulses with no data encoded on them) and to pick the best (closest) one. Even in the presence of several competing beacons vying for attention, Cricket accurately pinpoints the right one within a small number of seconds.

1.2.5 INS/Twine

INS [4] is a new naming system intended for naming and discovering a variety of resources in future networks of devices and services. Services have intentional names

that describe *what* a service is, not what its network location is.

The INS resolver architecture consists of a wide-area inter-domain resolver network of Domain Space Resolvers (DSRs) and an intra-domain self-configuring resolver network composed of Intentional Name Resolvers (INRs). It integrates resolution and forwarding that tracks change, and it uses soft-state name discovery protocols that enable robust operation.

Twine [7] is a resource discovery system that builds on INS, using Chord [38] as a distributed hash lookup table. It uses the same naming syntax as INS. It offers both discovery and early binding functionality to client applications. Twine implements a new form of intentional name resolution that achieves scalability via a hash-based partitioning of resource descriptions among the INRs.

Twine does not require pre-configured hierarchies or special naming syntax. It works with arbitrary attribute sets and achieves balanced resource distribution among participating resolvers. It also handles queries based on orthogonal and hierarchical attributes, with no content or location constraints. Twine maps an intentional name into a set of *strands* and stores a key for each strand. A query on an intentional name will return a set of resources that match the longest strand that is stored.

Twine uses a set of resolvers that organize themselves into an overlay network to route resource descriptions to each other for storage, and to collaboratively resolve client queries. Each resolver dynamically specializes in learning about a subset of other Twine nodes, as well as a subset of available resources.

1.3 Related Work

Pervasive computing systems today use variants of traditional password or token-based methods to authenticate mobile users. However, we are not aware of many that are concerned with location-aware authentication.

In the following sections we present some pervasive computing authentication systems, some contextual and some based on user identity, and compare them with our approach:

1.3.1 Cooltown

Cooltown presents a web-based pervasive environment, with capabilities for contextual authentication and, more specifically, location-aware authentication. Access goes through restricted access points where the user can either access an external proxy for the outside inter-net or a reverse proxy similar to a firewall that points to restricted services owned by a user.

Constrained Channels

For location authentication, one approach is by Kindberg [23]. The model is based on the telephone system, where a user can authenticate her location if she can pick up the phone that rings at a specific location. The role of the telephone is played by a channel proxy, which connects the client to the server via a constrained channel.

This approach assumes that the channel proxy is connected to the network and that it shares a secret with the server. Although this technique uses a secure channel between the proxy and the authentication server to transmit the user information data, it consumes a lot of resources, since it requires network-enabled proxies on every location reachable by users.

Web representations of Places

Another approach by the Cooltown project, similar to our approach in terms of how a user authenticates her location, is that of Caswell [10], where a *place manager* is responsible for creating web representations of physical places and associating web representations of users with the place representations.

Short range beacons transmit an encrypted time-stamp, which the client can use to form a cookie to prove where she is. This assumes that the beacon shares a key with the place manager and that their clocks are synchronized. The place manager keeps track where each client lies physically and offers access to the corresponding services. That scheme preserves user anonymity.

1.3.2 E-biquity-Centaurus2

In the Centaurus2 system as described in [11] we see a simplified Public Key Infrastructure, where all the clients and Service Managers have a public-private key pair to authenticate themselves. The authentication is not contextual in this system. Authentication is based on distributed trust where a user can delegate access rights to another user.

1.3.3 Gaia-Mist

Mist [6, 17] uses an extension of Kerberos authentication to provide privacy and access to resources in the system. Locations are represented in a hierarchy. Every user enters the system through a space authentication portal lower in the hierarchy and has a lighthouse higher up in the hierarchy. The lighthouse does not know which portal the user is at; thus her location cannot be pinpointed. The lighthouse is responsible for establishing authentication with the Authentication Server. The user uses a combination of authentication mechanisms (smart card, password,...), and the lighthouse computes the confidence with which the user has been authenticated by measuring the probabilities of false results by each authentication method. If the confidence is high enough then the Server gives tickets to the user. The protocol seems heavy-weight for simple access control by the use of confidence levels and multiple types of authentication.

1.3.4 Proxy-based Authentication

In proxy-based authentication for mobile devices, every user and device in a pervasive environment has a proxy in the network that is used for authentication and access control. The device is considered light-weight and communicates with the proxy via a secure channel. Proxies run on computationally strong computers. Every device has its own protocol for communicating with the proxy. The following sections present current proxy-based systems.

Charon

In the Charon system [15], mobile clients authenticate themselves to proxied services using a variant of the Kerberos authentication protocol [25]. A client obtains Kerberos tickets to access a service via the service's proxy. The proxy obtains the Ticket Granting Ticket from the Key Distribution Center and then a service ticket from the Ticket Granting Server. The client can then use the ticket to access a service via the proxy. The client never reveals to the proxy its Kerberos key, so the proxy cannot impersonate the client.

SPKI/SDSI

In SPKI/SDSI proxy-based authentication [9, 28], proxies communicate with each other to transfer user requests, and they authenticate each other using SPKI/SDSI (Simple Public Key Infrastructure [13], Simple Distributed Security Infrastructure [34]). Proxies are grouped into administrative domains called "proxy farms", and they contain Access Control Lists for granting or refusing access to the corresponding devices. The authentication mechanism follows the client-server model. The client proxy provides a proof of authenticity (signed request) and a proof of authentication (chain of SPKI/SDSI certificates that validate the signature) to gain access to a device.

Our approach is similar to proxy-based systems in terms of how users get access to devices. The difference lies in the fact that in our system the proof of authentication is a ticket issued by a Location Authority proving a user's location, rather than a chain of certificates. In our system trust is placed at the authority, whereas in a proxy-based approach trust is placed at the various proxies and their ability to pick trustworthy neighbors. Although a proxy-based approach provides strong security guarantees for authenticating mobile users, it seems too heavy-weight for access to simple services where quick responses are required (turning on the TV, speakers, lights, etc.).

1.4 PAC Overview

Our architecture uses INS/Twine [4, 7] for scalable resource discovery and Cricket [31, 32] for location discovery.

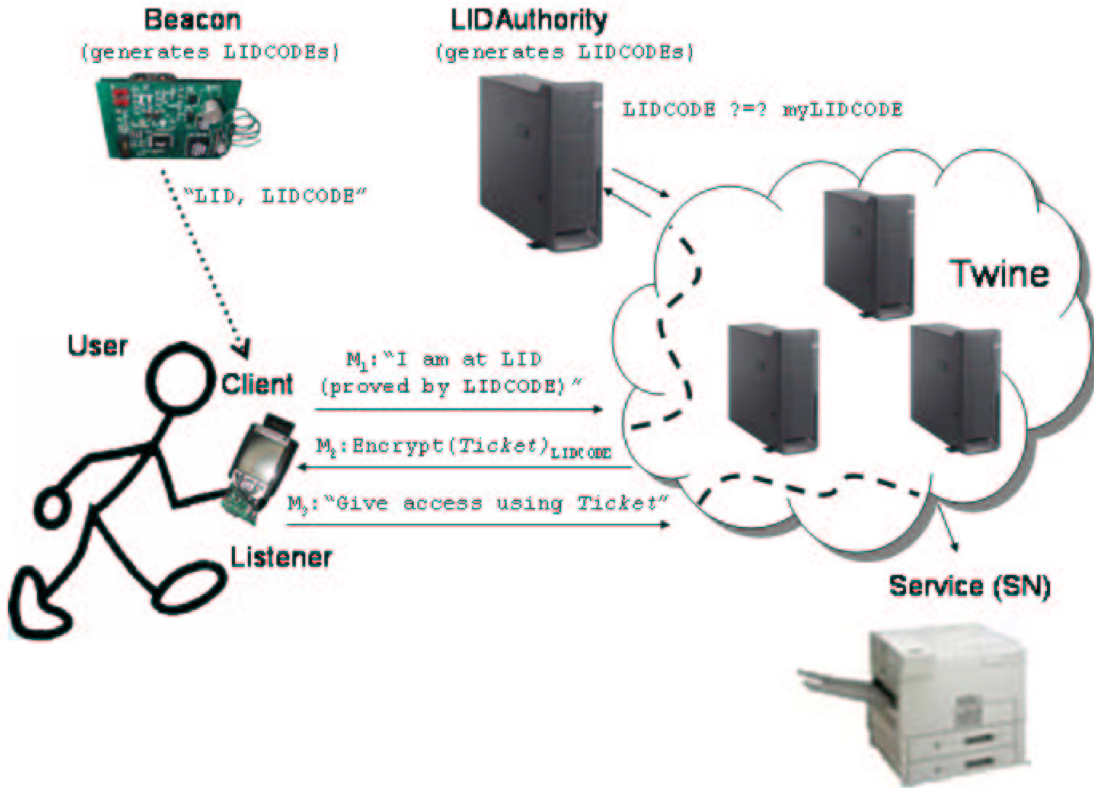

Figure 1-1: How a user accesses a service using PAC.

Users authenticate their location and request access to services as shown in Figure 1-1. A user performs her actions through a mobile client device. The client is registered with a Twine Resolver and is able to issue queries for a service using a Service Name *(SN)*, an intentional name that includes a description of the service and its physical location. In addition to discovering the service to be accessed, the client discovers a Location Authority *(LIDAuthority)*, a service registered in the Twine network that is responsible for authenticating the user's physical location. The client receives a Location ID *(LID)* from a nearby beacon, that describes the client's location. The client queries Twine to discover the LIDAuthority that can authenticate

the location described by the LID.

Along with the LID, a beacon sends a time-varying Location Code *(LIDCODE)*. The client sends a message M_1 to the LIDAuthority, including the LID and the Service Name of the desired service (SN). It includes a keyed hash of the contents of the message using the LIDCODE as the key, to prevent an eavesdropper from finding the LIDCODE or tampering with the message. The LIDAuthority verifies the message using its own copy of the LIDCODE (see Section 2.5) and then returns a message M_2, including a *ticket* that contains an expiration time, SN and the *location group* that this LID maps to (see Section 2.4). The ticket is encrypted with the LIDCODE to prevent an adversary from stealing it. The client uses that ticket in a message M_3 to access the service.

1.5 Goals

The mechanism used by PAC to authenticate a user's location and request access to services is only one of many alternatives that could be considered if we take into account the systems described in Section 1.3. Our goal was to design PAC such that the system can be deployed at low cost, access to services can be performed quickly using light-weight security, and the system can scale to support a large number of locations and users while keeping administration simple. On making these design decisions in order to solve the location authentication problem and to provide access control, we traded off other qualities for our system. In particular the trade-offs that arise for pervasive access control are the following:

- *Security vs. Fast Access:* The higher the security requirements of the system, the longer it takes to authenticate a user's location, both because more messages are involved and because public-key cryptography is required to bind users to locations. On the other hand, if fast access is required, then fewer credentials need to be presented by the user, and it might be easier for a malicious user to obtain a valid binary representation L of a certain physical location that she is not currently at and gain unauthorized access (i.e., having an insider leaking

beacon information to the wide-area network).

- *Security vs. Cost:* A system with high security standards might require higher costs to build the system infrastructure. For example, in Section 1.3.1, using constrained channels would require short-range access points in every room, corridor and hall. In addition, such equipment is more expensive than short-range beacons that broadcast information at regular time intervals.

- *Security vs. Anonymity:* Proving that some *anonymous* user is in location L is a harder problem than proving user U is in location L, since the user identity must be hidden. L is just a digital description of the location, and once obtained, can be replicated and communicated to other users. Since there is no user identity bound to L, anyone can use a proof of the form $P(X, L, t)$ illegally. Therefore, to preserve anonymity while proving a user is at L, the proof must be sent to the authentication authority via a secure channel. In the example in Section 1.1, if the user wants to preserve her anonymity, she will not include her face in the photo when she authenticates her location. However, she could send the photo to another user that is not in front of the building entrance.

- *Scalability vs. Administration:* A system that supports only a small number of locations can be administered centrally. A system that needs to support many different locations and services is harder to administer centrally, particularly as the number of service owners increases. A better solution is to distribute administration. This however requires additional care to ensure there will not be any administration inconsistencies that lead to security holes.

In addition to the trade-offs presented above, there are additional ones that depend on the way a digital representation L of a location is obtained by a client and presented to a service. Today the most popular methods to obtain a digital representation of locations is either through a beacon (such as GPS and Cricket) or through a gateway (e.g., wireless phone tracking via the cell towers). If we assume that the basic components of a location authentication system are a client, a beacon/gateway

and a service, there are two possible ways to have the service know that the client is at the location L as shown in Figure 1-2: either the client sends information obtained from the beacon or the gateway sends information obtained from the client.

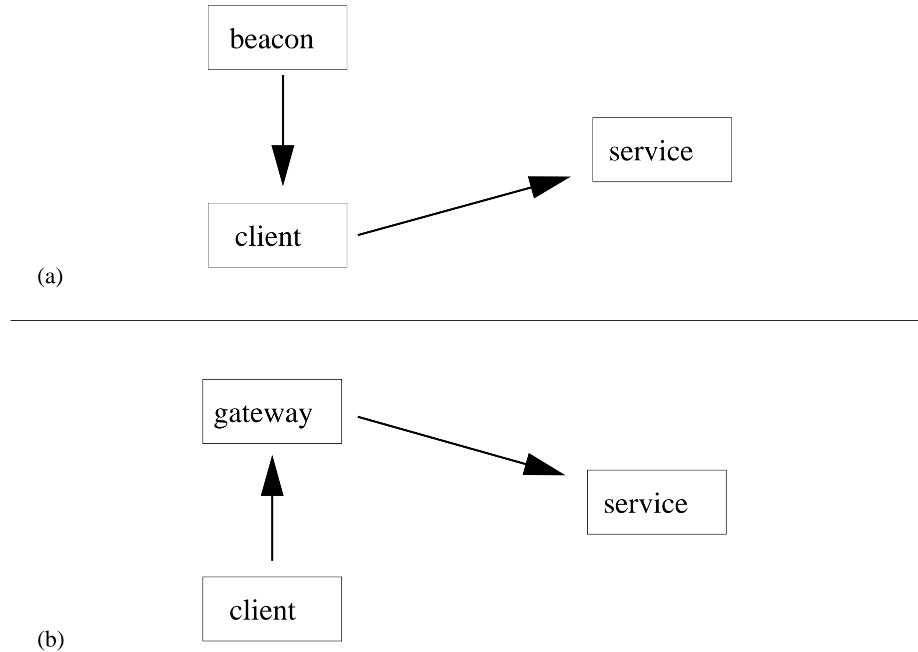

Figure 1-2: (a) Location information sent to the service via the client. (b) Location information sent via the gateway.

Sending L via the gateway has the advantage that the client never receives any sensitive information from the beacon, so a malicious client cannot leak out any secrets. In addition, the user can stay anonymous, since it can send a message to the gateway without any identification or the gateway could hide the client identity, when talking to the service. This approach is used by the constrained channels system described in Section 1.3.1. However, this approach is expensive since it requires more sophisticated equipment and additional network wiring to connect all the gateways to an existing wired network infrastructure. For that reason, in PAC we chose to send L via the client. We traded off some security to preserve anonymity, since the client receives the LIDCODE from the beacon in the clear. We also traded off some security for less cost by using beacons that only transmit information. A more sophisticated beacon could receive an identity from the client and send a token constructed using the LIDCODE and the client identity, and thus make the token unique to the client.

Chapter 2

Infrastructure

In this chapter we describe in detail the various components of the infrastructure for PAC and how they are managed. In particular, we describe the algorithm used for producing random numbers, the LIDCODE format, Location Groups, the LIDAuthority and the Service Agent.

2.1 PAC Pseudo-Random Number Generator

The PAC Pseudo-random Number Generator(PAC-PRNG) produces an unpredictable, but determistic number sequence given a seed value. The PAC-PRNG generates cryptographically strong random numbers by using hash functions as suggested by the RSA bulletin [27]; in particular it uses the MD5 hash algorithm.

The PAC-PRNG consists of two consecutive MD5 blocks, where the output of the first is used as an input for the second block. The output of the second block is the value used to produce the LIDCODE the user reads.

The first input to the PAC-PRNG is a seed that is a collection of random bits. Random bits are usually obtained by timing keyboard keystrokes, or recording mouse pointer screen coordinates. More information on such methods can be found at the Suggestion for Randomness Request for Comments [12].

The output of the first block S is transformed using a linear transformation $T_1(S) = S000$, which is the addition of $3 \times 128 = 384$ zeros to make the input

$S_0 = MD5[SEED]$
$S_i = MD5[T_2(S_{i-1}]$, for $i > 0$ (first block)
$output = MD5[T_1(S_i)]$ for $i \geq 0$ (second block)

Table 2.1: The PAC-PRNG algorithm

512 bits. For subsequent values another transformation of S is used as a feedback input to the same block, $T_2(S) = SSSS$, which is the concatenation of S four times to make the input 512 bits.

Table 2.1 describes the PAC-PRNG algorithm and Figure 2-1 illustrates the block diagram:

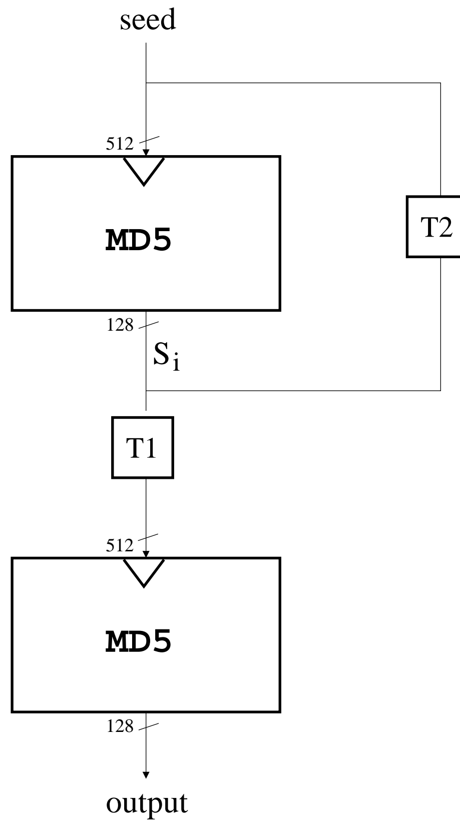

Figure 2-1: The PAC-PRNG block diagram.

2.2 LIDCODE

The LIDCODE consists of 20 bytes. The first 16 bytes are the LIDCODE-value, a number produced by the PAC-PRNG. The last 4 bytes are the LIDCODE-counter,

an index used by the PRNG synchronization protocol described in Section 3.1. It identifies the position of a specific LIDCODE in the PRNG sequence. The LIDCODE is illustrated in Figure 2-2.

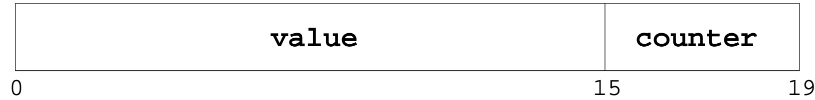

Figure 2-2: The LIDCODE.

The counter can take 2^{32} values. Even if a new LIDCODE is produced every second it would take approximately 136 years until the counter rolls over. Therefore, we assume the counter cannot roll over during the lifetime of a beacon.

2.3 Beacons

The Cricket beacons operate independently of the network infrastructure applications rely upon. They are not capable of heavy computations, nor of large storage. They run a basic implementation of a PRNG sufficient to produce a new LIDCODE every 60 seconds. They announce their LID and LIDCODE periodically.[1] They transmit both ultrasound and RF signals. In general, we can assume that a user far from the beacon cannot receive the signal transmitted. In Section 6 we discuss in detail the security issues with beacons.

2.3.1 The LID

The LID is a string that describes a certain location using a naming system that every administrator agrees upon. In PAC we use an attribute-value naming system with nested attribute-value pairs, as is used in INS [4], so a LID would be of the following form:

[building = B [floor = F [room = R]]][beacon = b]

[1]The period is approximately 250ms.

The first part is a location string describing the particular location inside an indoor space. The second is an identification string for the particular beacon. All administrators must agree on the format of the location string. However, the identification string can be of any format the particular beacon administrator wishes. This string is opaque both to the user and the LIDAuthority. The location string is used by a mobile client to discover services using INS/Twine, while the identification string is used by the LIDAuthority to map a client to a location group.

2.3.2 Beacon initialization

When a beacon is initialized by an administrator, it is given a LID and a random seed to start its PAC-PRNG. The administrator establishes a secure channel to the LIDAuthority and sends the seed to the LIDAuthority. The LIDAuthority then registers the beacon and starts a PRNG that synchronizes and generates LIDCODEs in parallel with the beacon. The same procedure is used by an administrator to re-initialize a beacon (battery replacement, etc.). When that happens, old state stored for that beacon in the LIDAuthority is removed upon re-initialization.

2.4 Location Groups

Location groups are a means to describe locations that facilitates both location authentication and service administration. They abstract away the details of the LIDs and the LIDCODEs.

Many different LIDs can identify an administrative or geographical group of locations. For example, a long hall in a building can have two or more beacons that advertise locations. We can group these beacons together under the hall's location group. The LIDAuthority keeps track of the mappings between location groups and their LIDs as described in Section 2.5. The following example illustrates the concept. Four beacons advertise:

beac01: [building = NE43 [floor = 5 [room = left-hall]]] [beacon = 500-C1]

beac02: [building = NE43 [floor = 5 [room = left-hall]]] [beacon = 500-C2]

beac03: [building = NE43 [floor = 5 [room = right-hall]]] [beacon = 500-C3]

beac04: [building = NE43 [floor = 5 [room = right-hall]]] [beacon = 500-C4]

An administrator groups the first two beacons in a location group named [building = NE43 [floor = 5 [room = left-hall]]] and the last two in a location group named [building = NE43 [floor = 5 [room = right-hall]]].

In addition, the administrator can create a location group that is named [building = NE43 [floor = 5]] as a supergroup for the two location groups created.

A user belongs to a location group as long as she can present a valid LIDCODE from a beacon that belongs to the group or prove that she belongs to one of its subgroups if there are any. Every location group is maintained by a set of administrators. A group may have at most one supergroup, but several subgroups. This facilitates group administration similarly to directory administration in a file system.

We call a *location map*, the set of all location groups inside an indoor location (usually a building).

2.4.1 Location Group management

We have assumed that indoor physical locations are owned by individuals that reside in the building, so it makes sense to distribute the management of the corresponding location groups in the same manner.

Since groups represent locations it is straightforward for an administrator to name the groups after the location they represent. Location Groups follow the hierarchy of a building (building, wing, floor, room, etc.), so as mentioned earlier every location group might have one or more subgroups, but it can have at most one supergroup. This hierarchy has a tree structure and can be built top to bottom or bottom to top, when managing the system. Every group includes a set of *beacons*, a set of *owners* and a set of *members*.

Only *owners* have rights to modify the group and its subgroups (modify, add, delete). They can add beacons to the group (after they place them physically), remove them (and maybe physically), reinitialize them, modify the list of owners and

members, and so on. All these modifications refer to the computing environment. In the physical world anyone could add and remove beacons, but unless she is an owner of a group she cannot do the corresponding modifications in the computing environment.

Members have rights only to create subgroups and place themselves as owners to them. All the modifications take place by interacting with the LIDAuthority. Appropriate accounts and passwords must be setup to protect from unauthorized access to location group information stored at the LIDAuthority.

Each *beacon* belongs only to one location group. In addition, an owner usually will not name the LID after the group the beacon will belong to for several reasons, so the LIDAuthority has a mapping between LIDs and location groups as described in Section 2.5 to facilitate the administrator in setting beacons for certain groups.

Creating a new group makes sense if at least one new service is to be owned by the group or at least one new beacon becomes a member. Someone that has an account with the LIDAuthority can create a group by making a new entry and providing a name; then she can add some beacons that she has initialized and put on the ceiling. Finally she can add other people as owners or members of the group she created.

2.5 LIDAuthority

The LIDAuthority is a special service registered in the Twine network that authenticates the location of users. It stores the mappings between location groups and LIDs and keeps track of the corresponding LIDCODEs as they change with time. After the LIDAuthority authenticates a user's location using the Location Authentication Protocol described in Section 3.3, it returns a ticket to the user that can be used to access services. The LIDAuthority has a public-private key pair (*LIDA.PUB, LIDA.PRI*) to authenticate itself to the services and to sign tickets for users. To facilitate location authentication, the LIDAuthority organizes its information using the *LID-LIDCODE* table, the *LID-LGroup* table and the *LIDTree*.

The LID-LIDCODE table contains entries that have information about the beacon

LID	init_time	current LIDCODE-counter	pointer to LIDCODE buffer

Table 2.2: Entry format of LID-LIDCODE table

LID	Location Group

Table 2.3: Entry format of LID-LGroup table

identified by a LID as well as LIDCODE information about the beacon. More specifically, each entry contains the LID, the beacon initialization time, the LIDCODE-counter of the currently active LIDCODE and a pointer to a LIDCODE buffer used by the synchronization protocol as described in Section 3.1. Table 2.2 illustrates the format of a table entry in the LID-LIDCODE table.

The LID-LGroup table contains mappings from LIDs to location groups. Table 2.3 illustrates the format of an entry in the LID-LGroup table.

The LIDTree is a tree data structure that contains the hierarchy of the location groups. It can be used to find the supergroups and subgroups that are included in the ticket issued to clients.

Figure 2-3 illustrates the format of the LIDTree.

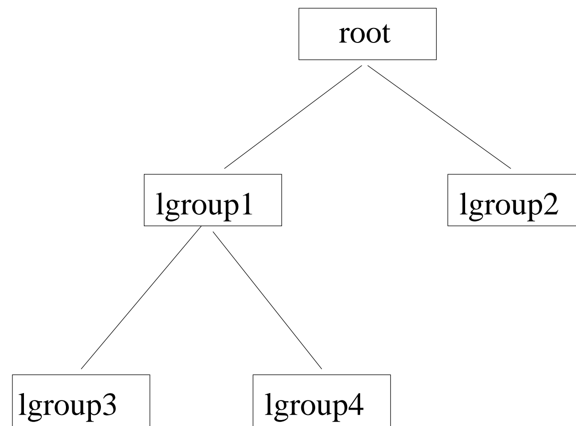

Figure 2-3: The PAC-PRNG block diagram.

The LIDAuthority runs *LIDA-PRNGs* that produce the same sequence as the PAC-PRNGs of the beacons it knows about. There is one LIDA-PRNG for each beacon PAC-PRNG. This is similar to the SecurID Server that keeps track of the numbers in SecurID tokens [3]. The LIDA-PRNG receives securely the seed from each beacon when that beacon is initialized (Section 2.3.2). After initialization the

two PRNGs do not communicate. The LIDA-PRNG runs the same algorithm as the PAC-PRNG. In addition, it updates the LID-LIDCODE table and the corresponding LIDCODE buffer with the newly generated values, and it synchronizes itself with the PAC-PRNG of the corresponding beacon in case it detects drift (Section 3.1).

A LIDAuthority advertises its name in the Twine network using an intentional name that includes the root of its LIDTree. One building is not necessarily served by a single LIDAuthority. Many LIDAuthorities can be deployed to cover a large location map. Using Twine a client can find the closest LIDAuthority that serves the location group that the nearest beacon to the user belongs to. In Section 5.1 we describe some enhancements in PAC so that the LIDAuthority can be distributed.

2.6 Service Agent

A Service Agent sits between the client and a service and handles access requests for that specific service. It verifies access tickets (Section 3.4) and grants or refuses access to clients. It advertises SN, its intentional name that includes a location (e.g., [service=printer][location=[room=504]]) in Twine. The Service Agent communicates with the corresponding service via a secure channel, so that the service can only be accessed through the Service Agent. For example a printer microprocessor and the Service Agent can share a symmetric key and establish a secure channel that way.

A Service Agent maintains an *access set* that is similar to a host-based ACLs that contains the location groups from which it can be accessed. In addition to the access set it stores the nonces of the tickets it has seen, until the tickets expire (Section 3.4).

2.6.1 Service Agent Set-up

To set up a service the owner needs to decide the following: whether the service operates using one-time requests or operates in a continuous mode, and which groups are included in the access set. If the service will operate in continuous mode then the Service Agent must advertise in SN a renewal period during which a client can access the service without reauthenticating its location.

Also, the owner requests the LIDAuthority public key (LIDA.PUB) from an administrator (offline). After this information is determined, the Service Agent can be connected to the Twine network and start advertising its intentional name.

The access set contains as elements either individual location groups or sets of location groups. A name that defines a set of groups might be *thisLGroup.children* or *thisLGroup.subGroups*, the difference between children and subgroups being immediate subgroups versus all subgroups. In addition it may contain logical operations such as ALL and EXCEPT. Such definitions merely facilitate the insertion and deletion of groups instead of inserting or deleting groups one by one.

An owner can query the LIDAuthority and get the available location groups and their hierarchy before picking groups for its access set.

The system can have several set definitions to make the policies more flexible. In other words, the access set can be any subset of the location map. It is up to the owner to pick the appropriate sets that maximize the security of the service.

Although intuition may lead to the assumption that the owner of a location group is also the owner of all the services being accessed by the location group, this is not necessarily true. The owner of the service might not be the same person as the owner of the location group by which this service can be accessed. Therefore, setting up Service Agents is an independent task from setting up location groups.

Chapter 3

Protocols

In this chapter, we describe in detail the protocols that are followed to authenticate a user's location, validate tickets and grant clients access to services. In this chapter we also describe the protocol for synchronizing PRNGs. In addition, we describe the format of the messages that the client exchanges with the LIDAuthority and the Service Agent.

3.1 Synchronization Protocol

Since our authentication method is time based, synchronization between the beacon PAC-PRNG and the LIDA-PRNG is an important issue of our architecture. The main reasons for drifts between the PAC-PRNG and the LIDA-PRNG sequence are delays during beacon initialization and differences between the LIDAuthority and the beacon clock implementations as well as beacon failures.

For two PRNGs started by the same seed to be synchronized, we require that each PRNG will produce the same number within a time interval that is less than the time period between the generation of two consecutive numbers. In other words, if the PRNG produces a new number every S seconds, then the PRNG pair is synchronized if the two PRNGs produce number N at times t_1 and $t_1 + dt$ where $-S < dt < S$.

As explained in Section 2.3, the beacons are lightweight devices so they are limited to the basic operation of producing and transmitting a new LIDCODE every time

period. On the other hand the LIDAuthority is assumed to run on a machine with a powerful CPU and large memory. Therefore, we decided that the LIDA-PRNG will manage synchronization.

If the PRNGs are synchronized, the LIDAuthority compares the LIDCODE-value provided by the client with the most recent one that the corresponding LIDA-PRNG has generated. Because the client might send a value right after the LIDA-PRNG has generated a new value, the LIDAuthority also uses the previous LIDCODE-value for comparison (Section 3.3).

If the PRNGs are not synchronized, the protocol uses the LIDCODE-counter provided by the client to resynchronize them. The LIDAuthority can only hear about the current LIDCODE of a beacon indirectly, when users try to authenticate their location. The corresponding LIDA-PRNG maintains a LIDCODE buffer that holds all the LIDCODEs starting from the one previous to the one it has last heard to the one it has currently generated.[1] The LIDCODE buffer has a fixed maximum size that is determined by an administrator. If the drift between the two PRNGs exceeds the maximum size of the buffer, then the beacon must be re-initialized, because the LIDAuthority will not allow access using LIDCODEs that are too old even if they are correct. Table 3.1 illustrates the format of the LIDCODE buffer.

LIDCODE-counter	LIDCODE-value
$last - 1$	LIDCODE-value[$last - 1$]
$last$	LIDCODE-value[$last$]
.	.
.	.
.	.
$current$	LIDCODE-value[$current$]

Table 3.1: The LIDCODE buffer

If the beacon is located in a popular location, then the possibility for large drifts is less likely. However, if a beacon is located in a less popular location, then high drifts might have occurred before a user tries to authenticate herself again from that

[1]More specifically, the invariant maintained is that the buffer starts with the LIDCODE with counter 1 less than the valid LIDCODE with the highest counter that the LIDAuthority has heard so far.

location.

To be more specific, suppose the LIDA-PRNG counter is n, the counter last heard is l, the beacon PRNG counter is n', and the LIDCODE-value provided by the client is L.

We have the following two cases:

- If $n' > n$ then the LIDA-PRNG runs $n' - n$ cycles forward to reach LIDCODE-value$[n']$ and compares it with L. If the value doesn't match it returns an error; otherwise it updates the LIDCODE buffer as shown in Figure 3-1.

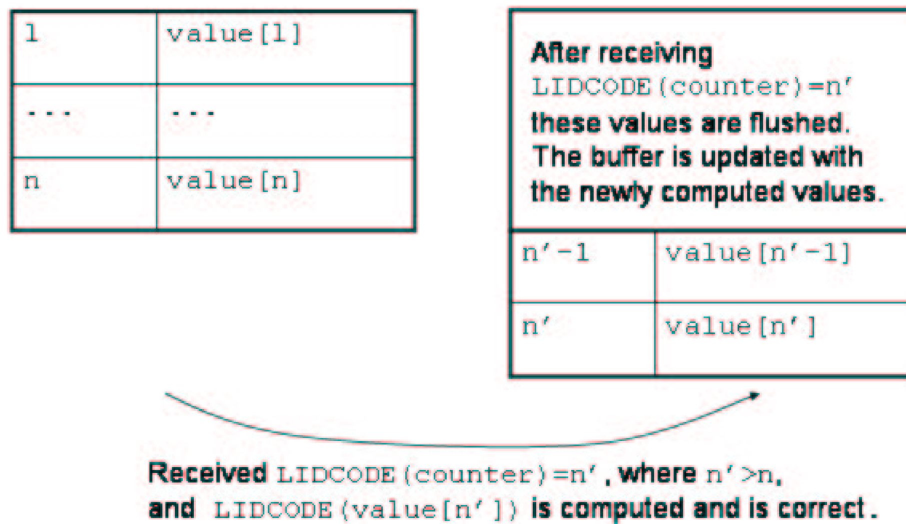

Figure 3-1: LIDCODE Buffer update when $n' > n$.

- If $n' < n - 1$ then it checks if $n' \geq l$. If not, it returns an error; otherwise it picks LIDCODE-value$[n']$ and compares it with L. If the value doesn't match then it returns an error; otherwise the LIDA-PRNG postpones the generation of new LIDCODEs until $n - n'$ periods pass, while using LIDCODEs from the

buffer to update the LID-LIDCODE table. It updates the buffer as shown in Figure 3-2.

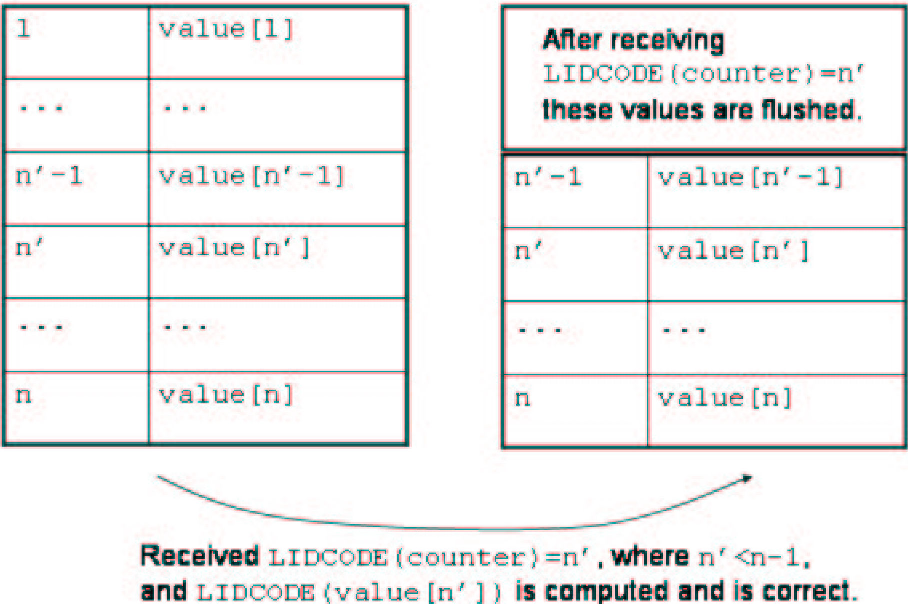

Figure 3-2: LIDCODE Buffer update when $n' < n - 1$.

3.2 Access Request protocol

The access request protocol consists of 4 steps: the ticket request, the ticket response, the access request and the access response. It is illustrated in Figure 3-3. The protocol is similar to the Kerberos authentication protocol, where the LIDAuthority plays the role of the TGS and the LIDCODE the role of the Ticket Granting Ticket $T_{c,tgs}$ as explained in Section 1.2.2.

- **Ticket Request:** The client sends a Ticket Request message to the LIDAuthority containing its current LID and LIDCODE requesting a ticket to access a specific service. It serves the same purpose as message 3 in Kerberos as

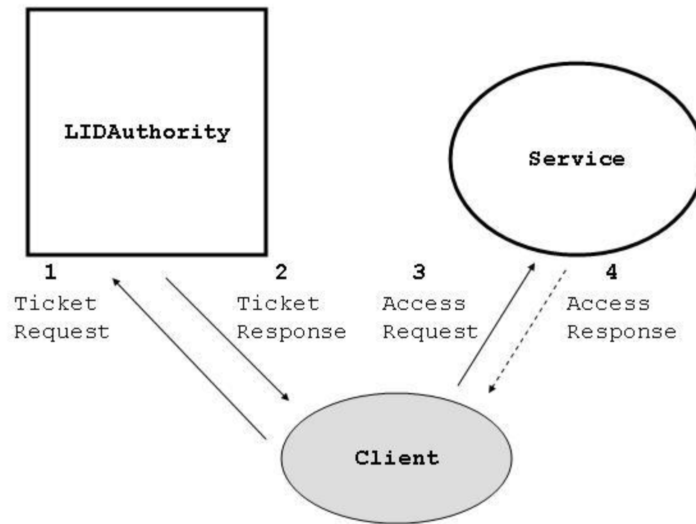

Figure 3-3: The Access Request protocol diagram. 1: Ticket Request, 2: Ticket Response, 3: Access Request, 4: Access Response (the dotted line indicates that step is optional).

presented in Section 1.2.2, where the client uses the TGT to obtain a service ticket.

- **Ticket Response:** The LIDAuthority verifies the location of the client using the Location Authentication Protocol (Section 3.3) and sends a ticket to the client to access the requested service. The Ticket Response serves the same purpose as message 4 in Kerberos.

- **Access Request:** The client sends the ticket to the Service Agent, optionally including data. The ticket is included in an Access Request message. The Access Request is similar to message 5 in Kerberos.

- **Access Response:** The Service Agent verifies the ticket using the Ticket Verification Protocol (Section 3.4) and replies with an Access Response message. The message contains an error if the verification fails. If it succeeds, the Service Agent forwards the data to the service and optionally returns an acknowledg-

ment or a handle to the client to connect to the service. The Access Response is similar to message 6 in Kerberos.

3.2.1 Ticket Request message

The Ticket Request message consists of the following parts:

- *nonce (N):* This is a random number generated by the client that identifies the message and it is used to protect from replay attacks.
- *LID:* the name of the location that the user is currently at (extracted from the beacon announcement).
- *service name (SN):* the service name of the requested service.
- *LIDCODE(counter) (n)* This is the LIDCODE counter as it is received from the beacon. It is the last 4 bytes of the transmitted LIDCODE.
- *HMAC:* The keyed hash of all the above information with the LIDCODE as the key, $HMAC_{LIDCODE}(N, LID, n)$. This way the message can be verified by someone who knows the LIDCODE (i.e., the LIDAuthority) and also cannot be figured out by an eavesdropper.

N	LID	n	$HMAC_{LIDCODE}(N, LID, n)$

Table 3.2: Ticket request message format

For our keyed hash we use the HMAC construction of Bellare, Canetti, and Krawczyk [8]. The HMAC function is described as

$$HMAC_k(x) = F(k, pad_1, F(k, pad2, x))$$

where the commas represent concatenation, k is the key, pad_1 and pad_2 are sequences of a known constant, F is a cryptographic hash function, and x is the data being authenticated.

3.2.2 Ticket Response message

The ticket that is included in the Ticket Response consists of the following parts:

- *nonce (N'):* The LIDAuthority generates a random number as a nonce for the ticket. N' protects against replay attacks as explained in Section 6.
- *Location ID path (LIDpath):* The current location group of the client along with the chain of its parents up to the root of the LIDTree. It uniquely identifies the user's location.
- *service name (SN):* The name of the service the client wants to access. Its use is explained in Section 6.
- *ticket expiration time (TE):* This is the time after which the ticket expires. It is the number of milliseconds, since midnight, January 1, 1970 UTC.
- *Signature:* This is a signature on the above information with the LIDA.PRI as the key, $\sigma\{N', LIDPath, TE\}_{LIDA.PRI}$. That way the Service Agent can verify that the message comes from the LIDAuthority and that it is not faked by a third party (or the client).

N	LIDPath	SN	TE	$\sigma\{N', LIDPath, SN, TE\}_{LIDA.PRI}$

Table 3.3: Ticket format

Table 3.3 shows the format of the ticket. The actual Ticket Response message contains an encrypted version of the ticket. The ticket is encrypted using the LIDCODE sent by the client as a symmetric encryption key. This ensures that only a client at the location, where the LIDCODE is being advertised, can decrypt the message and obtain the ticket.

3.2.3 Access Request message

The Access Request message has the following format:

- *ticket:* This is a copy of the ticket that is obtained after decrypting the Ticket Response message using the LIDCODE.
- *[data]:* This is an optional field. These are client specific data which may vary from a command (e.g., turn projector on) to a file for storage. They are interpreted by the service.

ticket	[data]

Table 3.4: Access request message format

3.2.4 Access Response Message

The Service Agent returns an error if the client is not permitted to access the service. It is not required however to acknowledge access permission being granted. If sent, the message has the following format:

- *status:* The status either contains an error or optionally a handle for access to the service.
- *[response data]:* The Service Agent optionally returns response data to the client in addition to the status/handle.

status	[response data]

Table 3.5: Access response message format

3.3 Location Authentication Protocol

To verify a client's location from a Ticket Request message the LIDAuthority performs the following steps:

- *step 1:* it reads the LIDCODE counter from the message and compares it with the ones that it keeps in the LID-LIDCODE table. If the counters match then it

goes to the next step. If not then it calls the Synchronization Protocol (Section 3.1). If the synchronization protocol returns an error then the LIDAuthority returns an error; otherwise it proceeds to the next step.

- *step 2:* it verifies the HMAC of the message using the LIDCODE(value). If the calculated HMAC doesn't match with the one provided then it returns an error. If they match it goes to the next step.

- *step 3:* it checks that the nonce hasn't appeared before. If it has then it sends an error to the user. If not then it proceeds to the next step. Nonces sent during a particular LIDCODE are removed from memory when that LIDCODE is removed from the LID-LIDCODE table and the synchronization buffer.

- *step 4:* it determines the LIDPath from the LID-LGroup table and the LIDTree given the LID from the message. It sets TE to be its current time plus an offset that is large enough to account for processing and network delays, but also small enough such that the user cannot move far from her current location before it uses the ticket. Assuming users cannot move very fast in indoor locations, the offset can safely be between 1 and 5 seconds.

- *step 5:* It produces a ticket with SN, TE and the LIDPath and signs it with its private key. Then it encrypts the ticket with the LIDCODE provided by the client and sends a Ticket Response message with the encrypted ticket to the client.

3.4 Ticket Verification Protocol

The Service Agent receiving an Access Request message performs the following steps before it grants access to the user:

- *step 1:* it checks whether SN matches its intentional name. If not, it returns an Access Response message with an error.

- *step 2:* it verifies the signature of the ticket using the LIDA.PUB. If the signature doesn't match it returns an Access Response message with an error.

- *step 3:* it checks that the ticket expiration time is larger than its current time. In other words $TE > currentTime$. If not, it returns an Access Response message with an error.

- *step 4:* it uses the LIDPath value from the Access Request message and its access set to check whether it accepts requests from that location group. If the location group is contained in the access set, then the ticket is accepted; otherwise the Service Agent returns an an Access Response with an error. Once a ticket is accepted the nonce is kept in memory until the ticket's expiration time.

The Ticket Verification Protocol guarantees that a ticket can be used only once at a particular service. To access a different service at the same location, a client must obtain a new ticket for that particular service.

3.5 Continuous Operation Policies

To provide continuous access to a service, we let the service choose its own policy which can be realized on top of the basic ticket mechanism. We present some common policies here:

- Every time the client sends a message to the Service Agent it has to include a new ticket. To renew tickets the client has to keep repeating the Access Request Protocol. This policy is suitable for low message frequency communications between the client and the service. Such examples include actions of one message (turn lights on, print file.txt, turn projector on, etc.) or periodic messages (TV controller change channel, projector next slide, etc.).

- The first time the client accesses the service, the Service Agent verifies the ticket, sets a ticket duration value TD and remembers the nonce of the ticket

for TD seconds. It sends TD to the client in the Access Response message and lets the client open a connection to the service. The service continues to accept data from the client until TD expires. To continue accessing the service longer than TD, the client gets a new ticket and sends it without breaking the communication. This is shown in Figure 3-4. If the client fails to renew its ticket, then the Service Agent closes the connection to the service. This policy is suitable for cases where the session between the client and the service involves continuous data transfers. Such examples include downloading and uploading files, streaming data, etc.

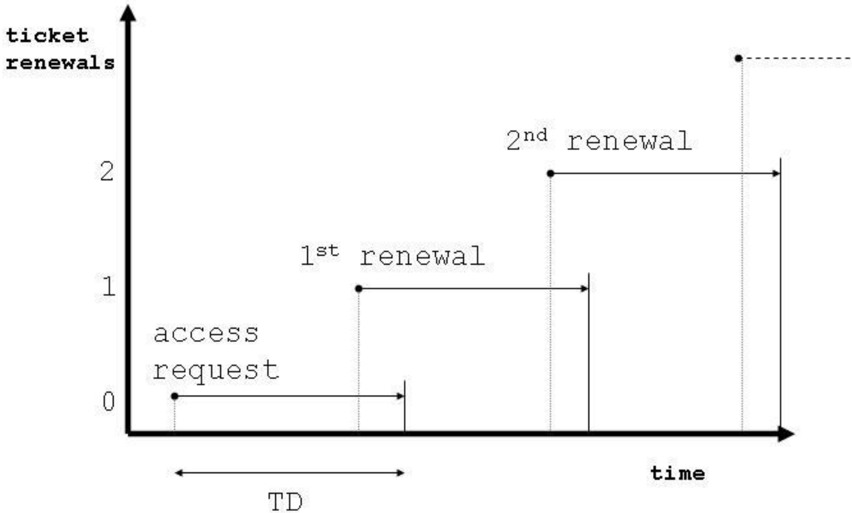

Figure 3-4: Continuous operation: the client renews tickets to continuously access a service. The renewals can take place independently from the client-service interaction.

Chapter 4

Building the system

We built the PAC infrastructure using the Cricket beacons and listeners [32] and the Twine/INS naming infrastructure [7]. More specifically, we modified Cricket beacons to receive LIDCODEs upon initialization and listeners to receive and transmit LIDCODEs in addition to the standard location information (LIDs). The LIDAuthority and Service-Agent were implemented in Java using the standard Twine application API. For our HMAC implementation we used SHA-1 as our hash function. To sign messages we used the RSA signature scheme provided by the Java 1.4.0 Cryptography API. To encrypt the ticket response we used the Blowfish encryption algorithm. Descriptions of all these algorithms can be found in detail in [35].

To ease the development of applications for PAC, both for clients and services, we provide an API that abstracts away the details of PAC as explained in Sections 4.3 and 4.4.

We tested our architecture by implementing a generic client, Service Agent and the LIDAuthority. We measured the performance of the client, Service Agent and LIDAuthority on a Pentium II processor machine running RedHat Linux version 2.4.2-2 kernel. All three applications were run on the same machine to avoid network delays in our measurements. The client was set up to send an access request with an 8 byte message attached as data every 3 seconds to the Service Agent

In addition we implemented a printer client and service agent as our first PAC applications. Our printer Service Agent was configured to to accept requests only

from within the author's office. The mobile agent was allowed to check the printer queue, remove print jobs from the queue and print documents only from within his own office.

4.1 Cricket modifications

The Cricket beacons were modified to receive a seed in addition to a LID from a serial interface. We implemented a Beacon Initialization application that received random bytes by recording keyboard or mouse movements and sending the seed via a secure channel to the SeedReceiver, the module of the LIDAuthority that registers beacons.

We also implemented the PRNG using the MD5 algorithm. Since the LIDCODE must be received correctly by the listener, the beacon computes also a CRC32 checksum [24] of the LIDCODE. The LIDCODE and the checksum are appended together and are sent as ASCII characters in their hexadecimal representation. They are added as a new field in a Cricket packet. The beacon updates the LIDCODE that is transmitted every 60 seconds. The additional length added to a cricket packet was 32 bytes for the LIDCODE-value, 8 bytes for the LIDCODE-counter and 8 bytes for the checksum (all 3 in hex format). The beacon transmits a packet every 250 ms. When running the system for a long time we noticed that the beacon slowed down as the batteries run low on power. In particular, we noticed that the LIDCODE generation period slowed down to 90 seconds from 60 seconds originally. With a 50% increase on the time period the LIDAuthority had a hard time synchronizing the LIDA-PRNG with the beacon PAC-PRNG.

The listener was modified to accept the LIDCODE/CRC32 as a new field and transmit it from its serial port. We didn't detect errors in the received LIDCODE at the listener to keep it simple. Any errors, due to RF noise, were detected at the client.

The LIDCODE is received by the client via GPSD, an application that parses Cricket packets received serially and runs a daemon that transmit the parsed packet information through TCP/IP. We modified the GPSD daemon to recognize LID-

CODEs as part of Cricket packets. The client subscribes to the GPSD daemon, receives the LID and the LIDCODE, and verifies the correctness of the LIDCODE by computing the CRC32 of the received bytes. If the LIDCODE is valid, then it can use it to gain access using PAC.

4.2 LIDAuthority implementation

The LIDAuthority was implemented as a Twine application. Since the state of the LIDAuthority is very sensitive, we decided to store it in a database. For our implementation we used MySQL [1]. We implemented an SQLAgent to translate reads and writes on the LIDAuthority state into SQL statements and contact the database. The reason for storing the data was because a database allows easy recovery in case the LIDAuthority application crashes while not adding a large overhead for storing and receiving data. Easy recovery is important especially when keeping track of the LIDCODE sequence for each beacon.

Running our tests using the generic client and Service Agent, we recorded the performance of the LIDAuthority in processing requests. We measured the total time to process a ticket request and the time to construct and sign a ticket response. We also measured the time it took to validate the HMAC of the access request while processing the ticket request, which is a part of the total processing time. The total time consumed by the LIDAuthority is the sum of the ticket process time plus the ticket response construction time. The numbers presented are the averages over many client requests. Table 4.1 summarizes the results.

process ticket request	32.5 ms
construct ticket response	129.2 ms
validate HMAC	25.9 ms

Table 4.1: LIDAuthority performance analysis.

4.3 Service Agent implementation

The Service Agent was built using the Twine application API. Applications that want to provide a service using PAC can be built on top of our generic Service Agent implementation. The details of the PAC protocols are abstracted away from the application. More particularly, applications only need to define an access set, a set of location groups from which they can be accessed, and what action they should perform when access is granted. The first can be input to the application from a file and the latter by providing an implementation of the method `performAction(data)`, which takes as input data passed by the client and is called by the Service-Agent when access is granted. The method `performAction` may return data, in which case the Service-Agent includes them in the Access Response message back to the client.

Our performance analysis for the generic Service Agent is summarized in Table 4.2. More particularly we measured the total time for the Service Agent to process an access request and reply to the client. We also measured the time to verify the LIDAuthority signature, which is a part of the total time. The values presented were averaged over many trials.

process access request	24.0 ms
verify signature	8.0 ms

Table 4.2: Service Agent performance analysis.

4.4 Client implementation

We implemented a generic client as a Twine application. Client applications that want to access PAC services can be built on top of our generic client. Developers can use the API that we provide that abstracts away the details of the PAC protocols. To send requests applications can use the methods `sendData(data)` or `sendDataWithResponse(data)`. The only difference is that the latter returns any data that the service includes in its Access Response message. In addition, applications that have continuous access to a service and want to renew their tickets in

parallel can use the `sendRenewal(SN)` method for re-authenticating their location with the LIDAuthority and sending a new ticket to SN.

Our performance analysis for the generic client is summarized in Table 4.3. More particularly, we measured the total time to request access to a service starting from the moment after receiving the LID/LIDCODE information from a beacon until the moment the client received an access response from the service (request service). Furthermore, we break down the total time into the following parts: 1) the time to discover both the LIDAuthority and Service Agent using Twine (discover), 2) the time to open a connection to the LIDAuthority request a ticket, receive it and close the connection (request ticket), 3) the time to open a connection to the Service Agent, request access, receive and access response and close the connection (request access). The values presented were averaged over many trials.

request service	278.4 ms
discover	39.2 ms
request ticket	191.0 ms
request access	48.2 ms

Table 4.3: Client performance analysis.

Chapter 5

System Enhancements

In this section we list a few enhancements on our architecture that could provide a better solution for special classes of location-aware applications.

5.1 Distributing a LIDAuthority

Each LIDAuthority must process $u = |L| \cdot |U_L|$ requests (users) periodically, where L is the set of LIDs it knows about and U_L is the set of users at the location advertised by a LID. If each user on average sends u_r ticket requests per second to the LIDAuthority, then the LIDAuthority must handle $r = u \cdot u_r$ ticket requests per second. Since U_L is bounded by a maximum size, because of the short range of beacons, the maximum value of $|L|$ is $\frac{r}{U_L \cdot u_r}$, which is bounded by r.

Our LIDAuthority thread-based prototype can handle on average $r = 20$ ticket requests per second on a Pentium 4. To measure this number we made a different setup by having a single client open one TCP connection to the LIDAuthority, send 100 requests, and close the connection. We measured the total time to process those requests over several trials and averaged the results. The difference between these better results and the ones presented in Table 4.1 is due to the multiple threads processing requests in parallel versus a single thread receiving a request every 3 seconds.

If we assume that a user sends 1 ticket request every 30 seconds (to renew service or access a new service), and that no more than 5 users can be near the same beacon,

then the LIDAuthority can handle a maximum of 120 LIDs (beacons).

Such performance is good for small areas (e.g., building floors), but not satisfactory when we move to large spaces (e.g., museums) where a lot of beacons are needed to cover a location. We believe this delay is processor bound, since signing a message with an RSA private key takes time on the order of milliseconds. We expect future implementations to increase this performance, so that a single LIDAuthority can handle more users per location. However, even the fastest implementation of RSA encryption takes time on the order of 1 millisecond [36], so we can improve by at most one order of magnitude.

The above bottleneck is not a problem for the overall scalability of the system, because the LIDAuthority can be divided among a set of smaller instances, due to the hierarchical nature of location groups. Each instance is responsible for handling requests for a subtree of the LIDTree and the LIDAuthorities can form a hierarchy. Every LIDAuthority contains entries for all the LIDs of its children LIDAuthorities in the LID-LGroup table. If it doesn't process requests from a specific LID, then in the LID-LGroup table it has a pointer to the child LIDAuthority that handles that LID, instead of a location group. A user starts from the root LIDAuthority and after at most $O(log(|LIDAuthorities|))$ it finds the LIDAuthority closest to the service she is looking for. The user will need to ask the root again, whenever she moves to a location that is not serviced by the current LIDAuthority.

Because of this hierarchy, the architecture can scale proportionally to the number of beacons. By using more LIDAuthorities we can manage a larger crowd of users.

5.2 Grouping the LIDAuthority and Service Agent

The Service Agent has to verify the signature of the ticket whenever it receives an Access Request. For some applications this overhead might not be acceptable. There are two alternatives that avoid that overhead:

- the LIDAuthority and the Service Agent can share a symmetric secret key. Instead of using a signature the LIDAuthority authenticates the ticket by append-

ing a keyed hash of the contents of the ticket with that secret key. Validating this hash is much faster than verifying the signature.[1]

- the Service Agent provides its own LIDAuthority that is responsible for authenticating a small set of locations that the Service Agent is interested in. Since the LIDAuthority is part of the Service-Agent, there is no need for a public-private key pair for the LIDAuthority. In addition, there is no need for a ticket request-response, because the LIDAuthority can forward the client request after validating the client's location.

Those alternatives are shown in Figure 5-1.

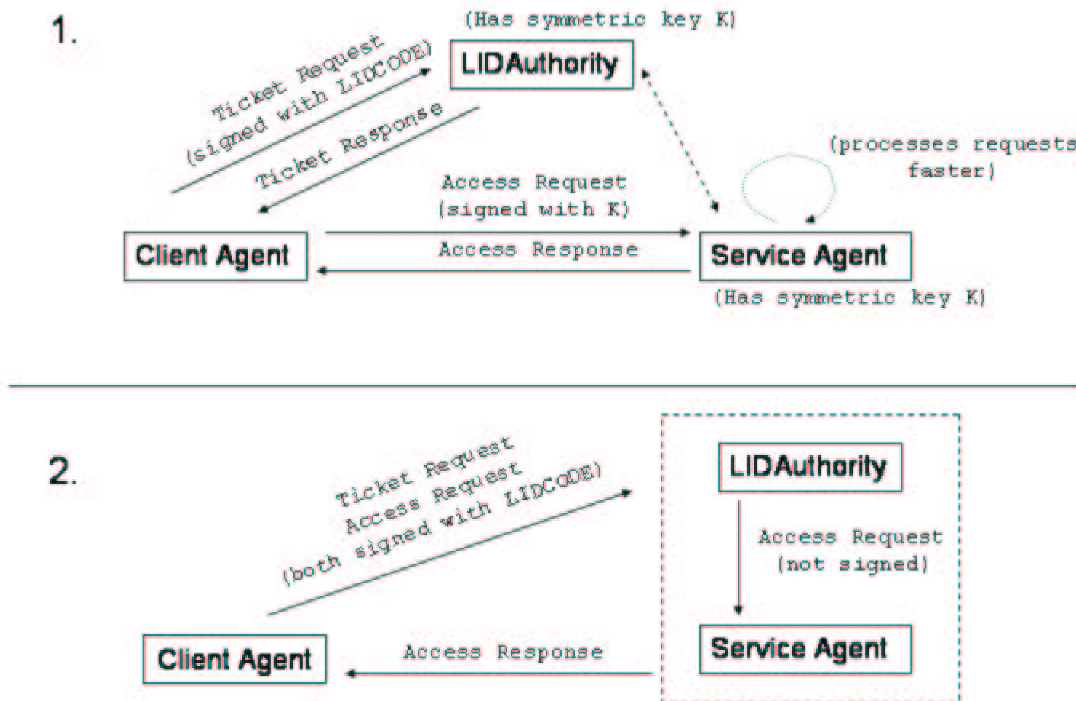

Figure 5-1: 1. LIDAuthority and Service Agent share a symmetric key. 2. LIDAuthority and Service Agent merge.

[1]According to [40] MACs can be computed three orders of magnitude faster than digital signatures.

Chapter 6

Security Evaluation

Using a small number of additional components and a small overhead by our protocols we can authenticate a user's location and avoid a large set of attacks. We leave doors open only to very elaborate attacks that only have localized effect. In addition, we provide a flexible design so services can provide their own access control.

We avoid a PKI and instead use LIDCODEs for a lightweight and simple location authentication for clients. The most sensitive piece of data is the LIDCODE and that is hidden both in the Ticket Request message and the Ticket Response message.

In addition, the LIDAuthority authenticates the ticket expiration time with its private key (LIDA.PRI) as described in Section 3.2. This adds more state to the LIDAuthority, but it saves additional overhead from the Service Agent, which would otherwise have to sign its own ticket expiration times.

In this section we present a detailed security analysis of the system and how it responds to different attacks.

6.1 Pseudo-Random Number Generator Analysis

The PRNG used in PAC is broken once the seed is determined. We analyse the strength of the PRNG both in terms of how easy is to predict the seed or compute it using the knowledge of the PRNG algorithm.

6.1.1 Seed prediction

Sources of truly random numbers are very difficult to find, especially when using deterministic methods to collect them. To elaborate on the issue, suppose we collect random bits by recording the x-coordinate of the mouse as it moves around the screen at constant short time intervals. The randomness of this input depends on the fact that it is impossible to replicate one's wrist movements as one moves the mouse around the screen. However, since the movement is continuous, the x-coordinates of neighboring intervals are not independent of each other, as the mouse scribes arcs on the screen. Therefore, additional bits have to be collected to increase independence among seed bits.

For every random bit that is required for the seed, a byte needs to be collected [27]. A random bit secret has entropy h that is usually less than its bit size b as illustrated by the mouse example. A guessing attack requires 2^h calculations to find the seed. Although there are no specific estimates, to be safe we collect $8 \cdot b$ bits to achieve entropy close to b when the collected bits are used to produce the final secret of size b.

6.1.2 Seed computation

The sequence is unpredictable as long as an adversary cannot guess the output of the first MD5 block.

The unpredictability of the PRNG sequence is based on the fact that MD5 is a one-way hash function. This guarantees that someone cannot guess the S_i's and recreate the number sequence. Furthermore the feedback input is not exactly the output of the first block (S_i), but a modification of it ($S_iS_iS_iS_i$). Therefore someone running an MD5 hash on the LIDCODE cannot find S_{i+1}.

A simpler construction of the PRNG could be the following: A single one-way hash block is used to output the LIDCODE. The input to the block is a preselected random seed and a sequence of numbers that is public (e.g., 1, 2, 3, ...). The output is deterministic since the same seed and sequence will produce the same outputs. It

is unpredictable since the input cannot be predicted by reversing the hash function.

6.2 LIDAuthority/Service Agent Synchronization

In order for the Access Request Protocol to work, the LIDAuthority assumes that the clock of the Service-Agent differs from its own clock by at most 1-5 seconds. This assumption is reasonable if the machine that runs the LIDAuthority is geographically close to the one running the Service Agent (e.g., same building) and an administrator has configured the clocks of the two machines. In general, both machines should use Universal Coordinated Time (UTC) or Greenwich Mean Time (GMT) to synchronize their clocks.

6.3 LIDCODE proxies

The LIDCODE is transmitted in the clear, so a malicious user in a particular location can broadcast the LIDCODE to other users anywhere in the network. We trust that a user will not have a motive to transmit the LIDCODE, because she has certain benefits by being there. For snooping-based attacks that do not use accomplices, the malicious user has to set up a "proxy" at that specific location that will keep reading the new LIDCODEs and broadcasting them.

Our lightweight security approach is not designed to deal with such an attack. To protect against it, the following configuration would help: 1) the user actively reveals her presence to the beacon by communicating directly with it; 2) the beacon is connected to the network and shares a secret with the LIDAuthority; 3) client messages go to the LIDAuthority through the beacon. Such a configuration of a "constrained channel" between the beacon and the LIDAuthority appears in [23]. However, this would add a large overhead both computational (for the beacon) and for key management.

6.4 LIDCODE playback

If no user presents a LIDCODE to the LIDAuthority from a specific beacon for a long time, then the LIDCODE buffer (Section 3.1) will reach its maximum size. When that happens, there is a "dead" time interval equal to the maximum size of the buffer times the LIDCODE generation period. A malicious user could be sitting under a beacon collecting LIDCODEs without using them to gain access to services. If the beacon is not by any other user, then the malicious user can go to another location and playback the old LIDCODEs as long as they are still inside the buffer. To be more concrete, let's assume that the LIDCODE generation period is 1 minute and the maximum size of the buffer is 30. A malicious user can collect 15 LIDCODEs, then go to some other location, which can be as far as 15 minutes from her current location, and start playing back the collected LIDCODEs. This attack is not easy since it requires the user to sit idle for 15 minutes under the beacon collecting LIDCODEs just to use them 15 minutes later, while assuming that no one else will pass by during this time.

6.5 Denial of Service

It is possible for a malicious user that listens to a certain LIDCODE to deny access to other legitimate users that listen to the same LIDCODE. She could read the encrypted ticket off the Ticket Response message and send an Access Request before the others. However, that requires a client that is faster at decrypting the Ticket Response message and faster at sending it to a Service Agent. The effect of such an attack is highly localized, since a single malicious user can only steal LIDCODEs from one beacon. Deploying an effective attack to a large set of beacons is costly and difficult to realize.

6.6 Stealing Tickets

A malicious user cannot obtain the ticket by eavesdropping on the Ticket Response message, since it is encrypted with the LIDCODE. However, she could capture and use a ticket sent to service SN by listening to Access Request messages. Since every service keeps a list of the nonces for the tickets it has heard, this replay attack will not work. Nonces are forgotten only after tickets have expired. Also reuse of the same ticket to access another service will not work, since SN is included in the ticket.

6.7 Beacon issues

A Cricket beacon sends all the information in the message using RF. Since the RF signal leaks through building walls, it is possible for a malicious user to listen to LIDCODEs from rooms that she is not in. Further improvement in the beacon hardware is required to transmit the LIDCODEs using more constrained media, such as infrared.

A second issue with beacons is that they have to be physically protected, so that malicious users cannot detach them from the ceiling and carry them as they move to different locations. This problem can be solved by detecting beacon detachment in a similar manner as when detaching fire alarms from ceilings.

6.8 Secure communication

If more security-demanding tasks need be performed then we can add extra security orthogonally to the PAC architecture, such as a PKI infrastructure, if personal identification or service authentication is necessary. Also, if the client needs secure access, it can connect to the service over SSL after it presents its ticket.

Since we do not authenticate services, a malicious user might set up a service (e.g., a printer) to be accessed by a location group of an organization that she does not belong to. If someone contacts her "outside" service from within the organization,

she could steal internal information (such as an email that is sent to her "outside" printer). However there are simple ways, orthogonal to our design, that avoid such scenarios by authenticating services (e.g., firewalls, certificates etc.).

Chapter 7

Conclusion

In this thesis we have described PAC, an architecture for providing access control based on a user's location. The PAC architecture uses a light-weight location authentication scheme based on time-varying random numbers called LIDCODEs that are broadcast by beacons at a particular location. A location authority uses the LIDCODE as a proof that a user is at a certain location and issues a ticket that can be used to access a service. Due to the small number of messages involved, clients get quick access to services. If the authority and the service are combined in a single entity then with only a single message a client can request access to a service and send data, without time consuming public key cryptography operations.

In terms of security, we have decided to provide a low cost solution that preserves user anonymity. Because of this the PAC system is susceptible to beacon proxy attacks, where an insider leaks LIDCODEs to the network in the clear. The reason that such an attack can be effective is that beacons are very simple in terms of functionality, are computationally weak, and are only able to transmit information at regular intervals using simple transmission protocols. Using more sophisticated beacons that are able to receive information from users and "personalizing" LIDCODEs to users, PAC can be further improved in the future to limit proxy attacks.

We have implemented a version of the PAC system and have used it build a printer Service Agent that allows access to printers based on user location within a building floor, while moving from room to room. Although we have a working location-aware

authentication system, there are several problems that need to be solved before PAC can be widely deployed. First, beacons must be physically secured, so that they cannot be tampered with or stolen. Second, beacons must have reliable lifetime guarantees, so that they can serve a location without frequent re-initializations. Currently, beacons are operating using batteries, so they cannot run for a long time reliably. Third, the beacon range must be short enough, so that it does not span more than the physical location they advertise. Transmitting the LIDCODE information using low power RF can still suffer from leakage through building walls. Fourth, a more sophisticated synchronization protocol that uses feedback techniques to estimate the period with which beacons transmit LIDCODEs is necessary, so that large drifts do not occur even if the LIDAuthority does not hear from a particular beacon for a long time.

We believe that the new field of pervasive computing is stirring interest in the development of context-aware applications and more particularly of location-aware applications. Authenticating a user's location in such applications is as important as authenticating a user's identity in today's desktop-based computing applications. We hope the work done in this thesis will motivate further interest not only in building secure location-aware applications, but also in exploring context-authentication in pervasive computing environments in general.

Bibliography

[1] Mysql database. http://www.mysql.com/.

[2] Red Hat Linux 8.0: The Official Red Hat Linux Reference Guide. http://www.redhat.com/docs/manuals/linux/RHL-8.0-Manual/ref-guide/s1-tcpwrappers-accesscontrol.html.

[3] RSA SecurID. http://www.rsasecurity.com/products/securid/index.html.

[4] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The Design and Implementation of an Intentional Naming System. In *17th ACM SOSP, Kiawah Island, SC*, Dec 1999.

[5] J. Al-Muhtadi, D. Mickunas, and R. Campbell. Wearable Security Services. Dept. of Computer Science Univ. of Illinois at Urbana-Champaign.

[6] J. Al-Muhtadi, A. Ranganathan, R. Campbell, and M.D. Mickunas. A Flexible, Privacy-Preserving Authentication Framework for Ubiquitous Computing Environments. Department of Computer Science, University of Illinois at Urbana-Champaign.

[7] M. Balazinska, H. Balakrishnan, and D. Karger. INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery. In *Pervasive 2002 - International Conference on Pervasive Computing, Zurich, Switzerland*, Aug 2002.

[8] M. Bellare, R. Canetti, and H. Krawczyk. Keying Hash Functions for Message Authentication. In *N. Koblitz, editor, Advances in Cryptology-Crypto 96, number 1109 in Lecture Notes in Computer Science. Springer-Verlag*, 1996.

[9] M. Burnside, D. Clarke, T. Mills, A. Maywah, S. Devadas, and R. Rivest. Proxy-based security protocols in networked mobile devices, 2002.

[10] D. Caswell and P. Debaty. Creating web representations for places. Hewlett-Packard Laboratories, 2000.

[11] A. Cedilnik, L.Kagal, F. Perich, J. Undercoffer, and A. Joshi. A Secure Infrastructure for Service Discovery and Access in Pervasive Computing. Technical Report TR-CS-01-12, Department of Computer Science and Electrical Engineering University of Maryland Baltimore County.

[12] D. Eastlake, S. Crocker, and J. Schiller. Randomness Recommendations for Security, Dec 1994. RFC 1750.

[13] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI Certificate Theory. Technical Report RFC 2693, Network Working Group, Sept 1999.

[14] N. Feamster and J. Furman. Group-based Authorization and Authentication in the Intentional Naming System. MIT Laboratory for Computer Science, Dec 2000.

[15] Armando Fox and Steven D. Gribble. Security on the move: Indirect authentication using kerberos. In *Mobile Computing and Networking*, pages 155–164, 1996.

[16] K. Fu, E. Sit, K Smith, and N. Feamster. Dos and Don'ts of Cient Authentication on the Web. In *Proceedings of the 10th USENIX Security Symposium, Washington, D.C.*, Aug 2001.

[17] C. K. Hess, R. H. Campbell, and M. D. Mickunas. The Role of Users and Devices in Ubiquitous Data Access. Technical Report UIUCDCS-R-2001-2226, UILU-ENG-2001-1733, Department of Computer Science University of Illinois at Urbana-Champaign, May 2001.

[18] L. Kagal, T. Finin, and A. Joshi. Moving from security to distributed trust in ubiquitous computing environments. In *IEEE Computer*, Dec 2001.

[19] L. Kagal, T. Finin, and Y. Peng. A Delegation Based Model for Distributed Trust. Computer Science and Electrical Engineering Department, University of Maryland, Baltimore County.

[20] M. Killijian, R. Cunningham, R. Meier, L. Mazare, and V. Cahill. Towards Group Communication for Mobile Participants. [Extended Abstract],Department of Computer Science Trinity College of Dublin.

[21] T. Kindberg, J. Barton, J. Morgan, G. Becker, D. Caswell, P. Debaty, G. Gopal, M. Frid, V. Krishnan, H. Morris, J. Schettino, B. Serra, and M. Spasojevic. People, Places, Things: Web Presence for the Real World. Hewlett-Packard Laboratories, 2000.

[22] T. Kindberg and J.Barton. A Web-based Nomadic Computing System. Internet and Mobile Systems Laboratory, Hewlett-Packard Laboratories, Jul 2000.

[23] T. Kindberg and K. Zhang. Context authentication using constrained channels. Hewlett-Packard Laboratories, 2001.

[24] J. Kohl and C. Neuman. The CRC-32 Checksum (crc32), Sep 1993. RFC 1510-6.4.1.

[25] J. Kohl and C. Neuman. The Kerberos Network Authentication Service (V5), Sep 1993. RFC 1510.

[26] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in Distributed Systems: Theory and Practice. In *ACM Trans. Computer Systems*, pages 265–310, Nov 1992.

[27] T. Matthews. Suggestions for Random Number Generation in Software, Jan 1996. RSA Laboratories Bulletin ♯ 1.

[28] T. Mills, M. Burnside, J. Ankcorn, and S. Devadas. A proxy-based architecture for secure networked wearable devices. Technical report, Laboratory for Computer Science, May 2001.

[29] Mudge and Kingpin. Initial Cryptanalysis of the RSA SecurID Algorithm. `http://www.atstake.com`, Jan 2001.

[30] R. M. Needham and M. D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. In *Communications of the ACM*, pages 993–999, Dec 1978.

[31] N.B. Priyantha. Providing Precise Indoor Location Information to Mobile Devices. Master's thesis, Massachusetts Institute of Technology, Jan 2001.

[32] N.B. Priyantha, A. Chakraborty, and H. Balakrishnan. The Cricket Location-Support System. In *6th ACM International Conference on Mobile Computing and Networking (ACM MOBICOM), Boston, MA*, Aug 2000.

[33] R. Rivest. The MD5 Message Digest Algorithm, Apr 1992. RFC 1321.

[34] R. Rivest and B. Lampson. SDSI-A simple Distributed Security Infrastructure. Oct 1996.

[35] B. Schneier. *Applied Cryptography, Second Edition: Protocols, Algorithms, and Source Code in C.* John Wiley and Sons, Inc., 1996.

[36] A. C. Shantilal. A Faster Hardware Implementation of RSA Algorithm. `http://islab.oregonstate.edu/koc/ece679cahd/s2002/ajay.pdf`.

[37] R. E. Smith. *Authentication: From Passwords to Public Keys.* Addison Wesley, 2002.

[38] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[39] A. S. Tanenbaum. *Modern Operating Systems, Second Edition.* Prentice Hall, 2001.

[40] Michael J. Wiener, Helena Handschuh, Pascal Paillier, Ronald L. Rivest, Eli Biham, and Lars R. Knudsen. CryptoBytes vol. 4, nr. 1: Performance Comparison of Public-Key Cryptosystems, SmartCard Crypto-Coprocessors for Public-Key Cryptography, Chaffing and Winnowing: Confidentiality without Encryption, DES, Triple-DES and AES.