6-1-2001

A Directory Infrastructure to Support Mobile Services

Ammar Khalid
Dartmouth College

A Directory Infrastructure to Support Mobile Services

Ammar Khalid

Ammar.Khalid.01@alum.dartmouth.org

Senior Honors Thesis

Dartmouth College Computer Science

Technical Report TR2001-391

Advisor: David Kotz

June 1, 2001

Abstract

Traditional Voice-over-IP applications such as Microsoft NetMeeting assume that the user is on a machine with a fixed IP address. If, however, the user connects to the Internet, via a wireless network, on a handheld device, his IP address frequently changes as he moves from one subnet to another. In such a situation, we need a service that can be queried for the most current IP address of a person whom we wish to contact. In this project, we design and implement such a directory service. The service authenticates all callers and callees, is robust against most host failure, and scales to several thousand registered users.

1 Introduction

Most existing network applications assume that one is on a machine that has a fixed IP address during the duration of the application. With the advent of mobile devices and wireless networking, this assumption is no longer valid, for the IP address of a user (person) on a mobile

This project is supported by the Cisco University Research Program, by Microsoft Research, and by DoD DURIP contract number F49620-00-1-0212.

device connected to a wireless network, changes as he moves from one subnet to another. Let us briefly consider the implications. Suppose Alice and Bob are two users on laptops connected to a wireless network, who communicate with each other via an instant-messaging application; many such applications exist, such as ICQ[1] and AOL Instant Messenger.[2] As Bob walks around and switches subnets, his IP address changes. The application's directory is unaware of this change. When Alice sends Bob a message, the directory unsuccessfully attempts to route it to Bob's old IP address. We use a simplistic model of an instant-messaging application, but this example emphasizes the general problem that exists. The directory cannot support mobility.

With the growth of the Internet, more users want to be online. The advent of wireless networking for laptops fueled the desire to be online from anywhere at anytime. This trend will increase dramatically as it becomes easier to connect the millions of Personal Digital Assistants (PDAs) to the Internet. People want their PDAs and laptops to be like cellular phones; online while on the go.

The advent of wireless networks has enhanced mobility, but led to trouble because IP addresses are no longer fixed. The network breaks, because the Application and Transport layers typically assume a stable IP layer. Research on Mobile IP attempts to handle IP changes in the IP layer, and IPv6 promises to support a similar scheme [1]. Some applications have been modified to detect and deal with IP address changes in the Application layer. But Mobile IP is rarely deployed, is inadequate for many applications, and IPv6 will not be deployed for years, if ever.

The instant-messaging application example illustrates the kind of problems encountered due to changing IP addresses of clients on wireless networks. Instant-messaging, text-chat, Voice-over-IP (VOIP), and videoconferencing applications enable users to communicate with one another in real-time. For these applications to service mobility, clients must obtain the most current IP addresses of other clients, quickly.

[1] http://www.icq.com/
[2] http://www.aol.com/aim/

G. Ayorkor Mills-Tettey, also at Dartmouth, designed Mobile Voice-over-IP (MVOIP) [2]. Her design allows clients to recognize an IP address change during an active VOIP conversation, to obtain the new IP address of the other party, and to continue the call. This paper focuses on the other problem associated with changing IP addresses: obtaining the most current IP address of a client when trying to establish a new connection. We design and implement a hierarchical directory structure that keeps track of clients' IP address changes. Each user has an alias (or user name) with the directory. A client wishing to connect to a mobile user's client, queries the directory, by name, for the most recent IP address of that user's client. Our goal is for the design to be robust, scalable, and secure.

Though designed originally to support MVOIP, our directory is really a tool for locating mobile people; people who move from one machine to another using machines that may or may not be mobile devices. *User mobility* [3] is the ability for a user to maintain the same identity on different terminals (machines) or terminal types, while *terminal mobility* [3] is the ability for a terminal to change physical location. Bob may not move physically, but may move from his desktop to his laptop to his PDA, maintaining his identity on the different types of machines (user mobility), his IP address changing on every move. Bob can then walk with his PDA – terminal mobility – while using a network service. Since our directory associates IP addresses with people, and not with machines, it keeps the most current IP address of Bob. Therefore, the directory provides *service mobility* – the ability for a user to obtain a particular service independent of user and terminal mobility. The directory can also be used to keep track of the IP address for mobile objects such as robots, for an application service provided on a mobile device, or for a mobile agent implementing a service.

This paper is organized as follows. In Section 2, we elaborate on the overall design of the directory infrastructure, looking at some of our design decisions and the reasons for them. Section 3 discusses the protocols of communication between the components of the directory during normal operation. Section 4 details how the directory is tolerant to failures. We describe

extensions that make the system secure in Section 5. We briefly discuss the implementation of a prototype of the directory in Section 6. Section 7 is a discussion of the pros and cons of our approach. Section 8 compares our model to existing directory service models. Section 9 outlines further work that we plan to pursue to improve this model. In Section 10, we draw conclusions.

2 Design

Locating clients that have a changing IP address is a challenge in wireless networks. The goal of our model is to provide a means to quickly obtain the most current IP address of a mobile client, knowing only the name of its user, without compromising the security of the network, and ensuring privacy of users. Communicating with mobile users should be as simple as sending an e-mail. Our model is designed to scale to several thousand users, securely, and with tolerance of failures.

Figure 1 illustrates the general layout of the directory. We discuss each component of the example in Figure 1.

Central Database

The central database contains the user names of all users registered to use the services provided by this directory. Each user is assigned a unique user name. Many institutions already maintain a database of all registered users of their network. Although our prototype uses a simple database for this feature, our architecture accommodates the use of any similar database through an appropriate protocol.

Master Server

At the top level of this directory service is the Master server. The Master authenticates users that wish to use the service, either to allow other users to find them, or to look up other users. It knows the number of clients registered (logged into) with the system. The Master server also contains a list of the IP addresses of all Agent servers that operate below it.

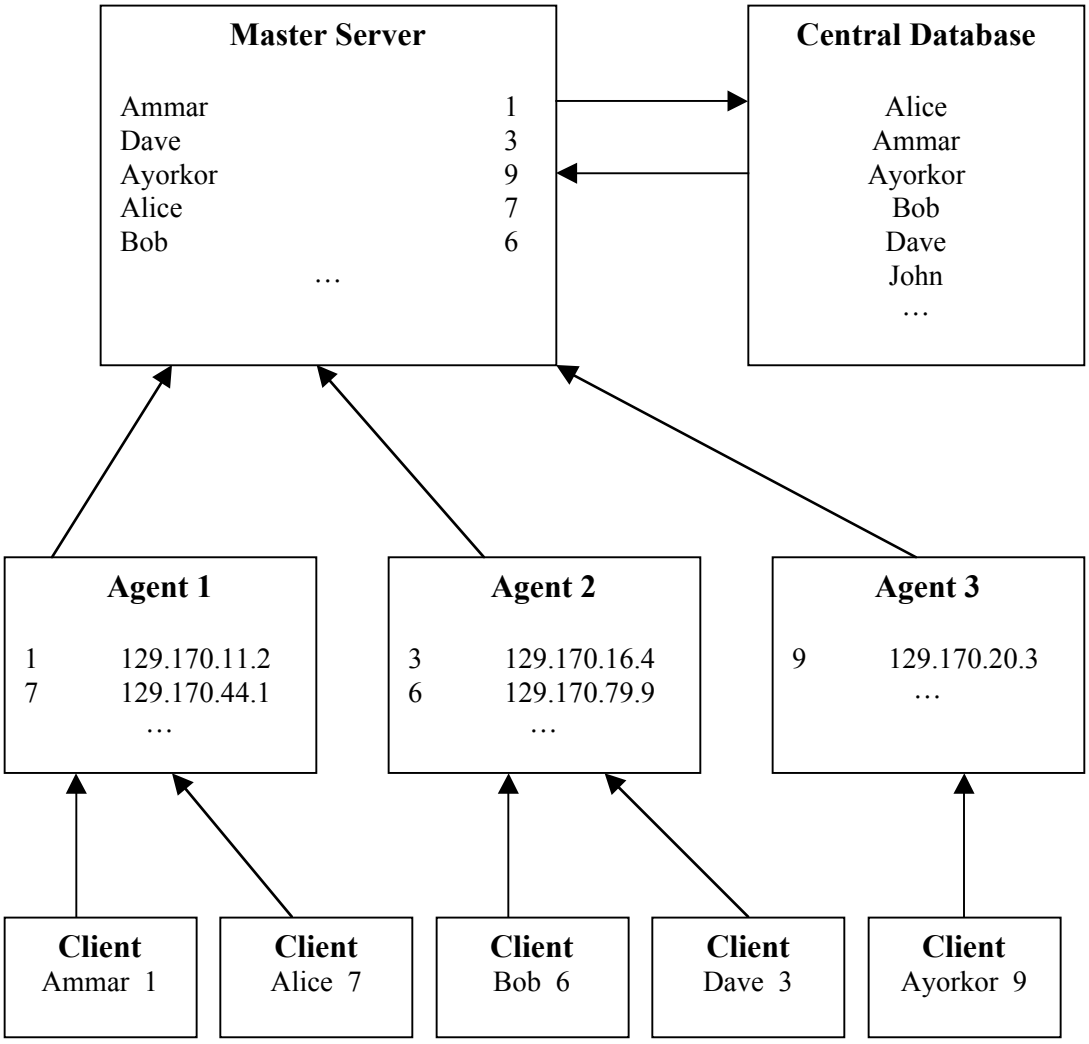


Figure 1: Layout of Directory

Agents

Under the Master server operate numerous Agent servers, simply called Agents.[3] Agents register with the Master server when they start up. Each Agent maintains the current IP addresses of a small subset of registered mobile clients. Our expectation is that one Master server cannot handle the IP address updates from all clients, so the Master delegates and distributes this responsibility among several Agents.

[3] The name "Agent" was chosen by loose analogy to the "agents" in Mobile IP or SNMP, and bears no relation to the use of that term in the "intelligent agents" or "mobile agents" communities.

Clients

Clients represent users registered with the directory service. The Master pairs each client with an Agent. The client is then visible to other clients, via the directory, and it can query the directory for the current IP address of other clients.

3 Communication during normal operation

This section shows how the different components of this service interact during normal operation. A network administrator starts the Master server. The Master is at a well-known IP address (or has a well-known hostname) and has "well-known" ports for registering Agents and clients with the system.

In the following discussion, we use the following notation to signify the contents of a message sent from one component of the directory to another: X→Y: (A, B, C, …). This notation means that X sends a message to Y with contents A, B, and C. We refer to the initiator of a communication channel, be it voice, video, or data, as the *caller*, and the client that receives the request from a caller, the *callee*.

3.1 Agents registering with the Master

The number of Agents in the system is proportional to the system load. A network administrator, however, must make the choice of the number of Agents to use, and he starts an Agent manually. The system has the flexibility that Agents can be added to it at any time, even when the system is already running. Therefore, we provide a mechanism for an Agent to inform the Master server that it is running and the Master can assign it clients.

An Agent knows the IP address of the Master server. When an Agent starts, it opens a TCP connection to the Master on the known Agent-Registration port. The registration request is, Agent→Master: (Agent IP address, Clients' Port, Agent-Query-Port, Logout-Port). The field

"Clients' Port" is the port where the Agent listens for IP-update packets (see Section 3.3) from clients. The Master server forwards this port number to clients when they register (see Section 3.2). The Master forwards a request for an IP address of a client to the Agent-Query-Port. Clients notify the Agent on the Logout-Port when they logout (see Section 3.5).

When the Master receives this message, it adds the Agent's information in its list of Agents, and sends a message, Master→Agent: (Agent ID, Master-Heartbeat-Port). We describe the use of the Master-Heartbeat-Port in Section 4.1. The Agent ID is a unique integer assigned by the Master. A crashed Agent is assigned the same ID when it is restarted, provided it has its old IP address. Figure 2 depicts the protocol.

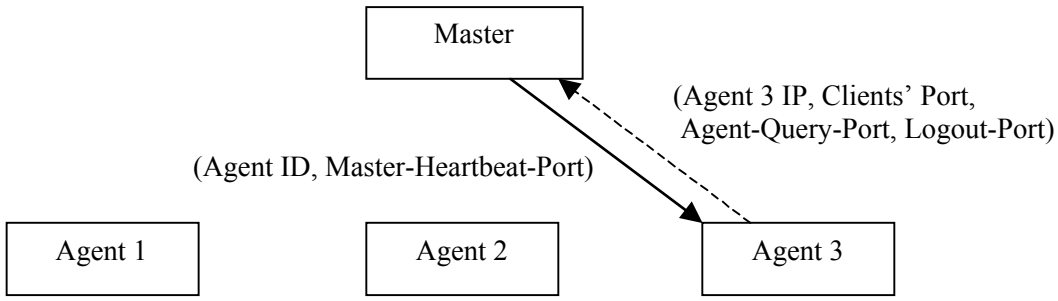


Figure 2: An Agent (Agent 3) registering with Master

3.2 Clients registering with the Service

A user wishing to use the service to communicate with other users must first register his client with the directory service. The client knows the IP address (or at least the host name) of the Master server. The client makes a TCP connection to the well-known Client-Registration port on the Master server and sends a message, Client→Master (user name). The Master listens on different ports for Agent and client registrations.

The Master server checks for errors upon receiving the registration request. It sends a "User Not Found" error to the client if the supplied user name is not in the central database. In the

extreme case that no Agent server is running, the Master server sends the client the error, "No Agent Running."

If the abovementioned errors do not occur, the Master registers the user's client. The Master first maps the user name to a new, unique integer ID. It then hashes the client (using its ID) to an Agent, thereby assigning the client an Agent. The Master knows how many Agents are running. Moreover, the Master assigns the client IDs sequentially. We use a simple hash function:

```
Agent ID = (Client ID) mod (Number of Agents)
```

The Master also keeps a counter for each Agent that tells how many clients are assigned to each Agent. The Master increments the counter of the Agent assigned to the client.

The Master stores the client's ID and the ID of the client's Agent in its dictionary of clients, using the unique user name as the key. The Master server then replies to the client with a confirmation message, Master→Client: (Client-ID, Master-Query-Port, Master-Logout-Port, Agent IP, Agent Port, Agent-Logout-Port). The client needs all of this information. The Master listens on the Master-Query-Port for requests for IP addresses of users' clients, i.e., queries of the form, "What is the current IP address of Alice?" The Master listens on the Master-Logout-Port for logout requests from clients. The client's Agent listens on the Agent Port for IP-update packets (described below). The Agent listens on the Agent-Logout-Port for logout requests from its clients.

We chose different ports for different message types, rather than one port with different message types. This approach frees the Master and Agents of checking the message type before processing a message. This simplicity becomes important when Agents need to process a thousand packets every minute. Dedicated ports process messages faster, at the expense of using more of the port-number space on their host.

Figure 3 depicts the client registration process.

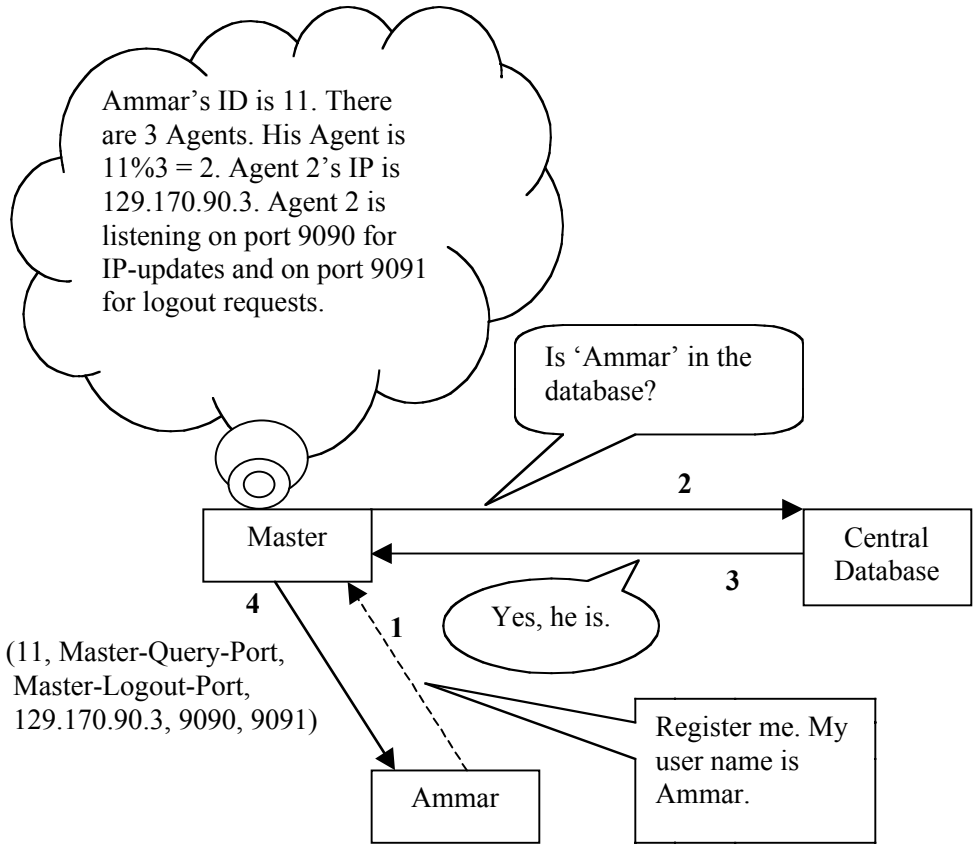


Figure 3: The Client registration process

3.3 Updating of Client IPs

When a client registers with the service, the Master assigns it an Agent. Clients send their current IP address to their assigned Agent every minute, Client→Agent: (Client ID, Current IP Address, Client-Listening-Port, Subnet-Count[4]), as Figure 4 shows.

When a client registers with the service, the Master server does not inform the client's Agent of the assignment. After receiving its Agent's IP address, the client immediately sends it an IP-update packet. The client's first IP-update packet serves to notify the Agent that it has registered and been assigned to that Agent. Of course, it also serves to tell the Agent the current IP address of the client. As client IDs are unique, when an Agent receives the first IP-update

[4] The "Subnet-Count" field is explained in section 5.2.3.

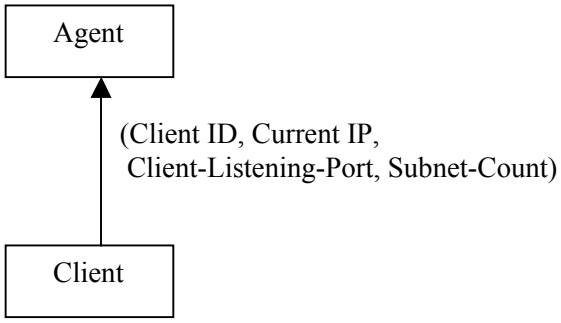


Figure 4: Client sending IP-update packet

packet, it does not have any information associated with the client's ID. The Agent adds an entry in its dictionary for this client. This approach enables the Agent to discriminate between new clients, and clients that are merely updating their IP address. This distinction becomes important when a crashed Agent is restarted (see Section 5.2.3).

There are two benefits of periodic IP-update packets: the Agents have the most current IP address of each client, and they know if a client is still functional. An Agent records the timestamps of the last IP-update packet received from each client. If a particular client has not sent an IP-update packet for the past three minutes, the Agent assumes that the client is no longer registered (possibly due to the client crashing or the user's machine crashing), or no longer reachable on the network, and removes the client from its tables.

If, however, a client changes subnets, its IP address changes. As soon as the client obtains the new IP address, it immediately sends an IP-update packet with the new IP address. This ensures that the directory service has the current IP addresses of all clients.

3.4 Obtaining a Callee's IP address

The purpose of this service is to obtain the current IP address of a user's client, knowing only the user name. Suppose Alice and Bob, mobile users on laptops, register with the service.

Alice now wishes to talk to Bob, for which she needs his client's IP address. Alice's client sends a message to the Master-Query-Port, Client→Master: (Callee's user name, Caller's IP address, IP-Request-Reply-Port, Caller's ID). Alice's client starts to listen on the IP-Request-Reply-Port for a reply. It waits for one minute after which it times out and assumes that Bob is unavailable.

The Master replies to the caller, Alice's client, on the caller-specified IP address and port number (IP-Request-Reply-Port) with a "User Not Found" error if the callee's user, Bob, is not a valid user. If the callee is not registered, the Master server sends the caller a "Callee Not Registered" error message. Otherwise, it obtains the information about the callee from its dictionary of clients, using the name of the callee's user as the key. The Master forwards the IP-request to the callee's Agent, through the Agent's Agent-Query-Port: Master→Agent: (Callee's ID, Caller's IP address, IP-Request-Reply-Port).

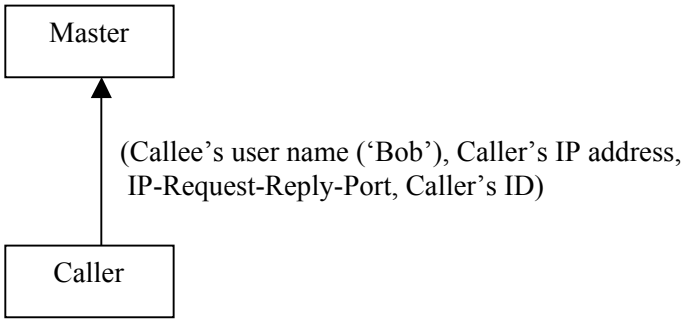
Upon receiving this message, the Agent looks in its dictionary for the specified client ID. If the ID is not found, it means that the callee is no longer registered with the system and so the Agent sends an error message, "Callee Not Registered," to the caller, and not to the Master.

When the Agent finds the ID, it obtains the IP address of the callee and replies to the caller (Alice's client) at the caller-specified IP and port: Agent→Client: (Callee's Current IP address, Callee's Client-Listening-Port[5]). This completes the protocol for obtaining an IP address of a user's client. The caller can now initiate communication with the callee, independent of the directory. Figure 5 shows the protocol.

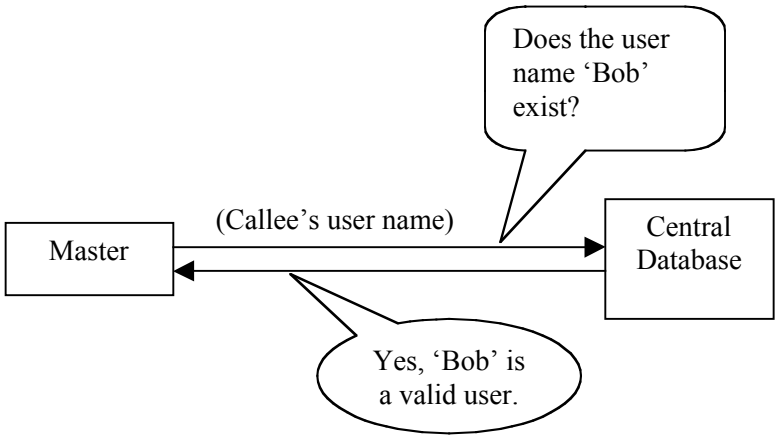
3.5 Clients Logging Out

Clients should inform the directory when they logout of the service, so that all information about them is deleted from the directory. When the client logs out, it sends a Logout message to its

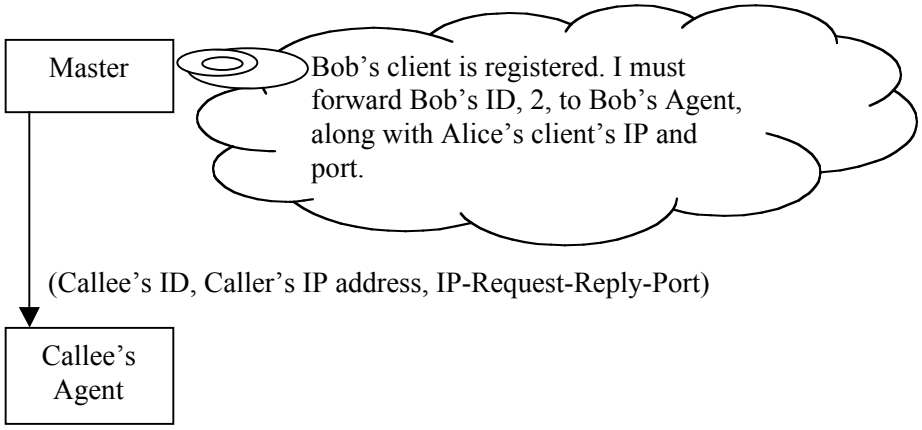
[5] See Section 3.3 for an explanation of the Client-Listening-Port.



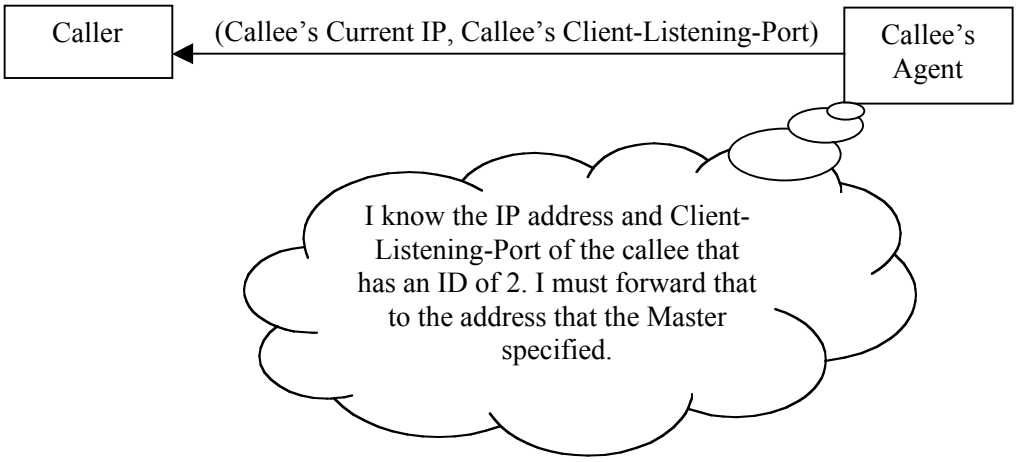
Step 1: Caller sends an IP-request to the Master



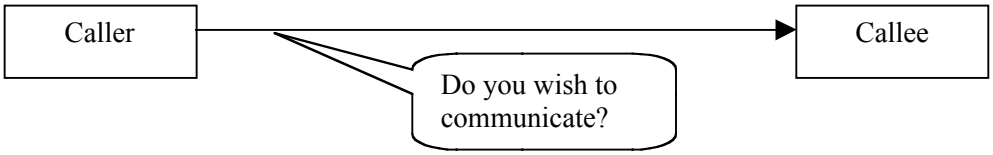
Step 2: The Master queries the Database for the validity of the Callee



Step 3: The Master forwards the IP-request to Callee's Agent



Step 4: Callee's Agent sends the Caller, Callee's IP and port



Step 5: Caller initiates a request to communicate with Callee

Figure 5: A successful IP address lookup

Agent and to the Master: Client→Agent: (Client ID) and Client→Master: (Client user name) (Figure 6). The Master decrements the counters of the total number of registered clients, and of the number of clients assigned to the client's Agent.

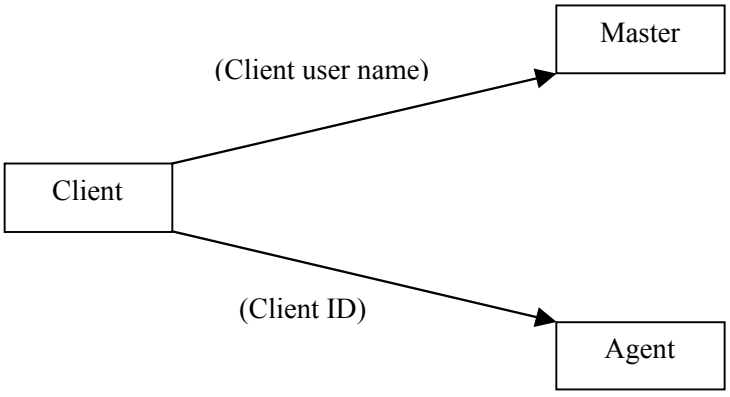


Figure 6: The Client Logout Process

4 Error handling and recovering from failures

Section 3 elaborates the communication protocols in the light of normal operation. This section relaxes the assumption that each component operates perfectly. The above discussion identifies some errors that the Master or Agents can detect. This section discusses failure recovery. We explore possible failures and how the system recovers from them.

4.1 Agent Heartbeats

Agents are servers that are prone to failures, of course. If an Agent (or its host machine) crashes, we lose the IP addresses of all clients assigned to that Agent. Hence, the crashed Agent's clients are now unreachable. We need a mechanism to ensure that if an Agent crashes, we know which Agent crashed, and we do not lose any client information. Our goal is to survive the failure of a single Agent. A second failure before the first Agent is replaced, may result in loss of client information, but should be unlikely.

To detect that an Agent crashed, and specifically which Agent crashed, Agents send "heartbeat" packets to the Master every minute. This packet contains the IP address and the ID of the Agent, so the Master knows which Agent sent the packet. The Master server records the timestamp of the latest heartbeat received from each Agent. If any timestamp becomes older than three minutes, the Master assumes that the Agent has crashed, and no longer assigns clients to the non-functional Agent.

The directory must not lose any client information due to Agent failures. Thus, each client is assigned two Agents, which both have the information about that client. The probability that both Agents of a client are non-functional at the same time is significantly lower than if only one Agent maintained the client's information.

When a client registers with the system, the Master assigns the client two Agents (the exact algorithm is discussed in the next section). The Master communicates the Agents' IP

addresses to the client, Master→Client: (Client-ID, Master-Query-Port, Master-Logout-Port, 1st Agent IP, 1st Agent Port, 1st Agent-Logout-Port, 2nd Agent IP, 2nd Agent Port, 2nd Agent-Logout-Port).

The Master server may need three minutes to realize that an Agent has crashed. The Master could, unfortunately, assign new clients to the crashed Agent during this "three-minute blind window." This poses no problems, as clients are assigned two Agents, and we assume, with high probability, that the other Agent is functioning.

4.2 Client Heartbeats

Clients send periodic IP-update packets to their Agents (see Section 3.3). A client sends identical IP-update packets to both of its Agents, ensuring that both Agents have its updated information (Figure 7).

The client IP-update packets serve to update the client's IP address with its Agents, and serve as a heartbeat. An Agent records the timestamp of the last IP-update packet received from each client. If the timestamp is more than three minutes old, the Agent assumes that the client has crashed and deletes all information about that client.

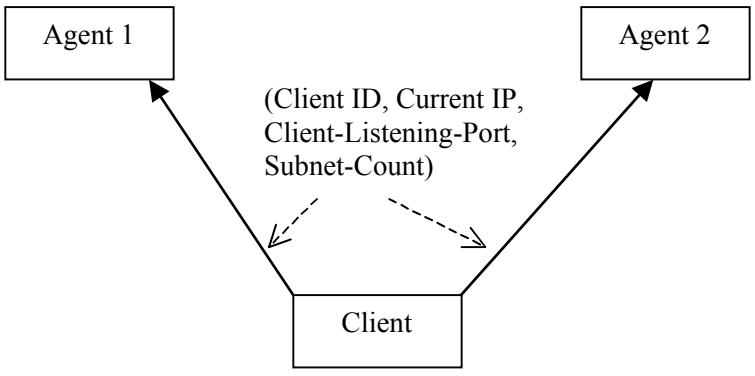


Figure 7: A Client updating its IP with two Agents

The Master does not need notification from Agents about crashed clients. When the Master receives an IP-request for a crashed client, it forwards the request to one of the client's Agents. The crashed client's Agent, however, has deleted all information about the client, and so replies to the caller with a "Client Not Registered" error message. When the client re-registers, the Master assigns it a new ID and "new" Agents, and overwrites the old information about the client with the new information. This is a desirable behavior even if the user is registered from another client and is registering from a new client, for the current IP address of the user is no longer the IP address of the "old" client. If the user returns to his old client, he must register with the service again.

If one of a client's Agents crashes, the client does not know that the Agent has crashed (we use UDP to send IP-update packets). It does not need to know this; it can continually send IP-update packets to the crashed Agent. We can be reasonably sure that the other Agent is still functional, and so the client is updating its IP address with the directory, successfully. If a client loses one Agent, or is assigned a dead Agent, it never notices, and continues to use the same IP address of the dead Agent. As a result, the crashed Agent must be restarted with its old IP address, otherwise we are vulnerable to failure of the second Agent.

When the crashed Agent is restarted, it suddenly starts to receive IP-update packets from its original clients. It treats those IP-update packets as first IP-update packets of "new clients" that are just registering and thereby rebuilds its tables of client information.

4.3 Agent-Assignment Process

When a client registers with the system, the Master assigns it two Agents. Section 3.2 discusses how the Master selects an Agent for a new client. The hash function is:

```
Agent ID = (Client ID) mod (Number of Agents)
```

This serves as the first Agent of the client. This Agent, however, may be a crashed Agent. The Master knows, with reasonable accuracy, which Agents are functional and which are not. If the Agent obtained from the hash function is functional, it becomes the first Agent of the client. If the Agent is non-functional, the Master chooses another Agent using the function:

$$1^{st}\ \text{Agent} = (\text{Agent ID} + 1) \bmod (\text{Number of Agents})$$

where Agent ID is the non-functional Agent obtained from the original hash function. If 1^{st} Agent is a non-functional Agent, the Master repeats the same process; otherwise, it becomes the first Agent of the client. The same function is then applied to obtain the second Agent of the client:

$$2^{nd}\ \text{Agent} = (1^{st}\ \text{Agent} + 1) \bmod (\text{Number of Agents}).$$

The Master repeats this process, as it did when finding the first Agent, until it obtains a functional second Agent for the client. The Master records the Agents assigned to the client. As mentioned above, the probability that both Agents assigned to the client are both in the "three-minute blind window" of the Master is very low.

When the Master detects an Agent crash, it does not assign any clients to the Agent. When the Agent comes up, however, it has fewer clients assigned to it than the other Agents, assuming that clients registered while the Agent was non-functional. To ensure a balanced distribution of clients to Agents, the Master assigns the "new" Agent as the first Agent for new clients until the "new" Agent has roughly the same load as the other Agents. The Master employs a different technique to pick the second Agent for a new client. The Master starts from the 0^{th} Agent and assigns the i^{th} Agent, the i^{th} new client that registers after the "new" Agent starts running, skipping the "new" Agent. This ensures that other Agents also get a balanced load of the new clients. Other probabilistic algorithms can be used to ensure a balanced load, especially since our model cannot handle multiple "new" Agents simultaneously.

5 Security and Authentication

Wireless networks are especially prone to malicious attacks. To make the directory service "safe" from such attacks we take some measures to provide security and authentication. The components must mutually authenticate one another, which makes it impossible to pose as the Master, as an Agent or as a client for a user. Our goal is to make it difficult for an intruder to match a user name to an IP address without authenticating himself and making an explicit query. We use Kerberos [4] for authentication and encryption, and therefore our service requires synchronized clocks (within a few minutes [5]) on the wireless LAN.

Section 5.1 discusses how the components mutually authenticate one another. Section 5.2 focuses on the security and privacy issues and what we do to ensure secrecy and integrity of all communication between the components of the directory. Section 5.3 highlights various cases in which a malicious intruder may try to get around the security measures, and how the system thwarts such attempts.

5.1 Authentication

We mutually authenticate all parties involved in this service. This section explains how we use Kerberos to authenticate clients and Agents with the Master, and clients with Agents.

When Agents and clients register with the service, they mutually authenticate themselves with the Master. The Master server obtains a Kerberos *ticket*[6] when the Master server starts running. Figure 8 is a depiction of how a client or an Agent authenticates itself with the Master server before registration. The Master can be an Agent as well in this diagram.

We now elaborate the messages exchanged [6]. The notation is as follows: K stands for a key, T stands for a ticket, and A stands for an authenticator (which is a timestamp). To represent that a datum X is encrypted with a key K, we write: {X}K.

[6] A ticket is a proof of identity, and is used for authentication.

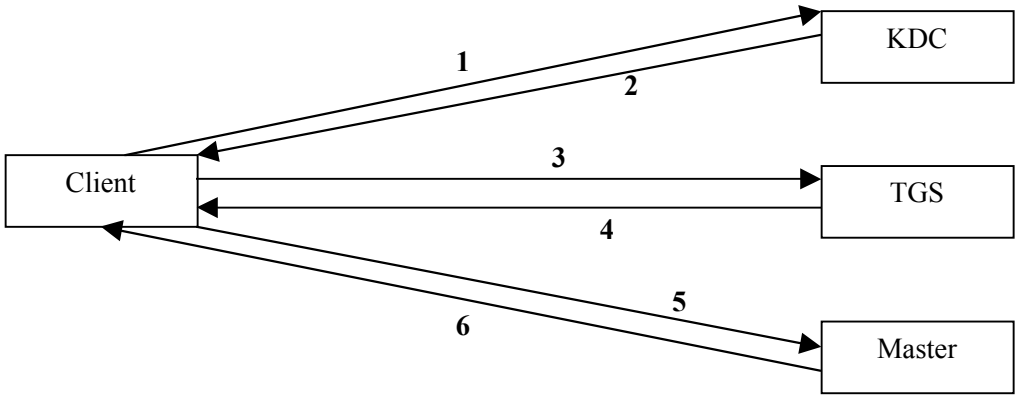


Figure 8: Kerberos Authentication Process before a Client/Agent registers

Messages 1 & 2: The client obtains a Ticket-Granting Ticket (TGT) from the Key Distribution Center (KDC).

1: Client→KDC: (user name)

The client attempts to obtain a TGT from the KDC.

2: KDC→Client: ($\{K_{Client/TGS}, \{T_{Client/TGS}\}K_{TGS}\}K_{User}$)

The KDC returns to the client a TGT, $T_{Client/TGS}$, encrypted with the Ticket-Granting Service's (TGS) secret key, and a session key to use with the TGS, $K_{Client/TGS}$. The whole message is encrypted with the secret key of the client's user.

Messages 3 & 4: The client obtains a ticket to register with the Master server.

3: Client→TGS: (Master, $\{T_{Client/TGS}\}K_{TGS}$, $\{A_{Client}\}K_{Client/TGS}$)

The client decrypts message 2 with K_{User}, the password of the client's user. The client then asks the TGS for a ticket to authenticate itself with the Master, presenting the TGT and a constructed authenticator, A_{Client}, (to prevent replay attacks), as proof of identity.

4: TGS→Client: ($\{K_{Client/Master}, \{T_{Client/Master}\}K_{Master}\}K_{Client/TGS}$)

The TGS sends back to the client a session key for communication between the client and the Master, $K_{Client/Master}$, and a ticket for the Master, $T_{Client/Master}$, encrypted with the Master's secret key, all encrypted with the Client/TGS session key, $K_{Client/TGS}$.

Messages 5 & 6: The client asks the Master server to allow it to register.

5: Client→Master: $(\{T_{Client/Master}\}K_{Master}, \{A_{Client}\}K_{Client/Master})$

The client decrypts Message 4 using $K_{Client/TGS}$ and then presents $T_{Client/Master}$ and a constructed authenticator, A_{Client}, as a proof of identity. The session key, $K_{Client/Master}$, is contained in the ticket $T_{Client/Master}$. Hence, the Master can obtain it.

6: Master→Client: $(\{A_{Master}\}K_{Client/Master})$

The Master sends the client a constructed authenticator, A_{Master}, to prove to the client that it is in fact communicating with the Master server.

This message exchange completes the mutual authentication between the client and the Master server. The process for an Agent registering with the Master server is identical, as is the process of a client authenticating itself to its Agents.

The client can now communicate with the Master under the covers of the session key, $K_{Client/Master}$, and can register securely. All communication between the Master and the client is encrypted with the session key. Similarly, all communication between an Agent and the Master, or a client, is encrypted with a session key.

For security purposes, we require that a user wishing to know the IP address of a user's client, register with the system, and thus authenticate himself via Kerberos. We realize, however, that PDAs have limited resources in terms of processing power. PDAs do not have the ability to execute the computation-intensive client-side Kerberos code. We can employ *Charon* [6], a proxy-based Kerberos authentication model developed by Armando Fox and Steven Gribble at the University of California, Berkeley, for authenticating users on PDAs.

5.2 Security and Privacy

This section covers the protocols we detail in Section 3 from the point of view of privacy. As we show in Section 5.1, all protocols are encrypted, providing some privacy. Below, we detail how that encryption is used, and identify the limits to privacy protection in our system.

5.2.1 Agents registering with the Master

After an Agent authenticates itself with the Master server, it has a session key, $K_{Agent/Master}$, for communicating with the Master. The Master server records these session keys, along with the other information about the Agent. The Agent sends a request to the Master to register, Agent→Master: ({Agent IP address, Clients' Port, Agent-Query-Port, Logout-Port}$K_{Agent/Master}$). The Master assigns an ID to the Agent and replies to the Agent, Master→Agent: ({Agent ID, Master-Heartbeat-Port} $K_{Agent/Master}$). Figure 9 illustrates the protocol with encryption.

5.2.2 Client Registration

Once a client authenticates itself with the Master, the Master and the client share a session key, $K_{Client/Master}$. The client registers with a message, Client→Master: ({user

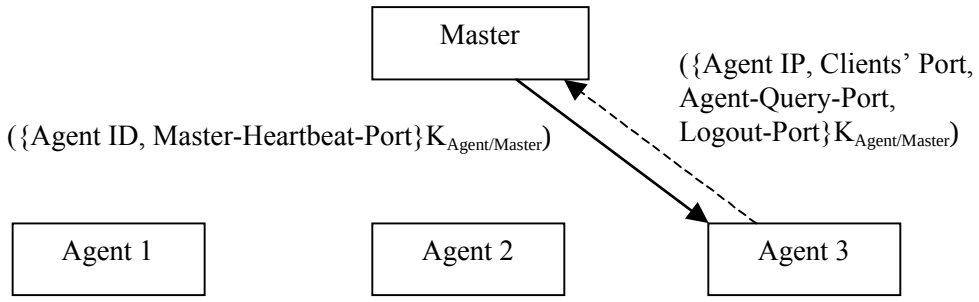


Figure 9: Agent registering with Master using Encryption

name}$K_{Client/Master}$). The Master assigns two Agents to the client, and registers the client, Master→Client: ({Client-ID, Master-Query-Port, Master-Logout-Port, 1st Agent IP, 1st Agent

Port, 1st Agent-Logout-Port, 2nd Agent IP, 2nd Agent Port, 2nd Agent-Logout-Port}$K_{Client/Master}$). Should someone sniffing the network obtain these messages, he is unable to decrypt them, as only the Master and the client know the session key.

The KDC contacts the network's database to validate a user before issuing a TGT to the user's client. The Master, therefore, does not need to validate the user by contacting the database. Hence, Steps 2 and 3 in Figure 3 are redundant. This does not make the central database redundant though, for the Master still needs to validate a callee's user name on an IP-request (see Step 2 of Figure 5). Figure 10 illustrates the modified client registration protocol with encryption.

The Master server stores the session key for each client, along with the client's ID.

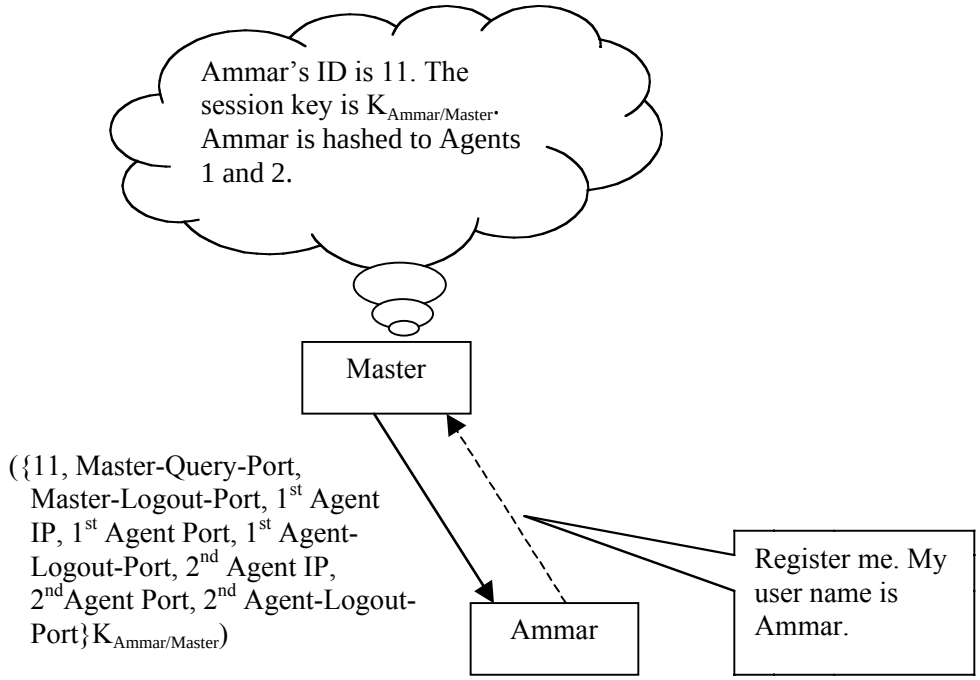


Figure 10: A successful Client registration with Encryption

5.2.3 Updating of Client IPs

A client must authenticate itself to its Agents before it begins to send IP-updates. The client obtains a session key to use with each Agent, $K_{Client/Agent}$, as Figure 8 depicts. The session

key with one Agent is different from the one with the other Agent. Once the client authenticates itself with both Agents, it proceeds to send IP-update packets to its Agents, Client→Agent-1: (Client ID, {Current IP, Client-Listening-Port, Subnet-Count}$K_{Client/Agent-1}$) and Client→Agent-2: (Client ID, {Current IP, Client-Listening-Port, Subnet-Count}$K_{Client/Agent-2}$) (Figure 11). An Agent records the session key it has with each client. When it receives an IP-update packet, it can tell which session key to use to decrypt the rest of the packet based on the ID in the packet.

The IP-update packet contains a field, "Subnet-Count," which is a counter (initialized to 0) that keeps track of how many times a client has changed subnets (and hence its IP address). Whenever a client changes subnets, it increments this counter.

For efficiency, the Agents do not decrypt every IP-update packet they receive. Each Agent records the IP address and the encrypted part of the IP-update packet (see Figure 11) of each client. An Agent determines which client sent an IP-update packet by looking at the

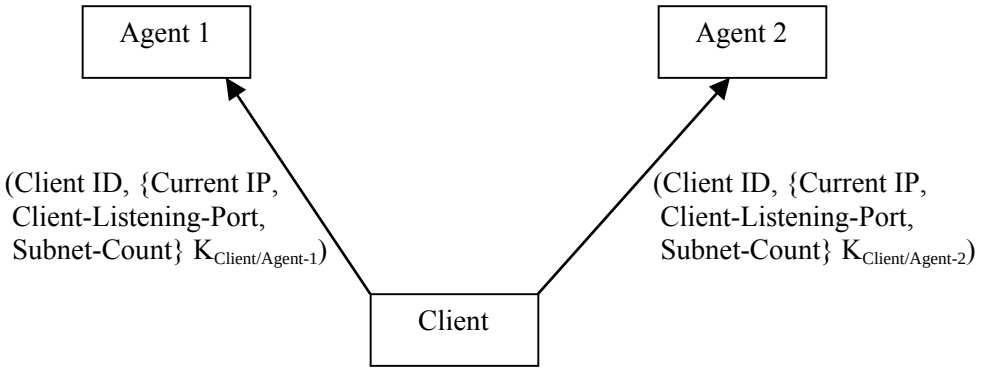


Figure 11: A Client updating its IP with its Agents using Encryption

plaintext "Client ID" field in the packet. It then compares (bit wise comparison) the encrypted part of the packet with the encrypted part that it has stored for the particular client. If the two ciphers are identical, then the client's IP address has not changed since the last IP-update packet (and neither has its subnet-count) and the Agent can disregard the packet without performing the expensive operation of decryption. If the two ciphers differ in even one bit, the Agent decrypts

the packet and updates its record of the IP address, subnet-count, and ciphertext of the particular client. This approach minimizes decryption operations to whenever a client changes subnets. A client employs a similar technique. Clients store the cipher, {Current IP, Client-Listening-Port, Subnet Count}$K_{Client/Agent}$. If a client's IP address is the same as it was when it sent the last IP-update packet, the client merely replays this cipher. When the client's IP address changes, it encrypts its new IP address and the incremented subnet-count, and stores the new cipher.

Since this system is capable of recovering from failures, we ensure that it does so without the loss of security. When an Agent crashes, the Master is able to tell which Agent has crashed and is able to adjust accordingly. When a failed Agent is restarted, it starts to receive IP-update packets from its clients. Note that the IP address of each client in these packets is encrypted with a unique session key that was established for communication between the Agent and the respective client. The Agent lost the session keys when it crashed. Obtaining new session keys for each client is inefficient due to the number of clients. We propose a more efficient solution to this problem.

When the client registered with the Agent, it obtained, from the TGS, a session key with the Agent, $K_{Client/Agent}$, as well as a ticket for the Agent that contained this session key, $\{T_{Client/Agent}\}K_{Agent}$ (see Message 4 in Section 5.1). The client caches this ticket, which is encrypted with the Agent's secret key. When a restarted Agent receives an IP-update packet, it checks if it has an entry for that client (using the plaintext client ID from the packet). The Agent sends a TCP error message to the client if it does not have its entry (and hence the session key) in its tables, requesting the client to re-send the encrypted ticket, $\{T_{Client/Agent}\}K_{Agent}$. The Agent obtains the IP address of the client from the IP Layer. The Agent already knows the default port where the client is listening for requests of this nature. The client complies by sending the cached ticket to the Agent, who decrypts it with its secret key and obtains the old session key.

5.2.4 Obtaining a Callee's IP Address

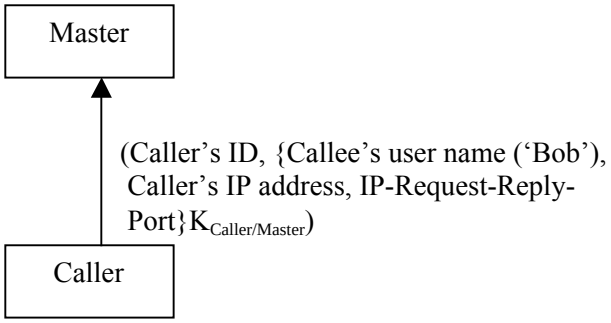
The caller uses the session key, $K_{Caller/Master}$ (see Section 5.2.2), to encrypt its IP-request, thus ensuring privacy, Caller→Master: (Caller's ID, {Callee's user name, Caller's IP address, IP-Request-Reply-Port}$K_{Caller/Master}$). The Master looks up the session key it has with the caller (using the plaintext caller ID from the message), and decrypts the rest of the message. The Master selects a functional Agent of the callee and forwards the request to that Agent, Master→Callee-Agent: ({Callee's ID, Caller's IP address, IP-Request-Reply-Port, $K_{Caller/Master}$}$K_{Master/Callee-Agent}$). This message differs slightly from its description in Section 3.4: it has the Caller/Master session key. We explain its significance shortly.

The Agent decrypts the message it gets from the Master, looks up the IP address of the callee, and replies to the caller, Agent→Caller: ({Callee's Current IP address, Callee's Client-Listening-Port}$K_{Caller/Master}$). Since the caller knows its session key with the Master, it decrypts the message.

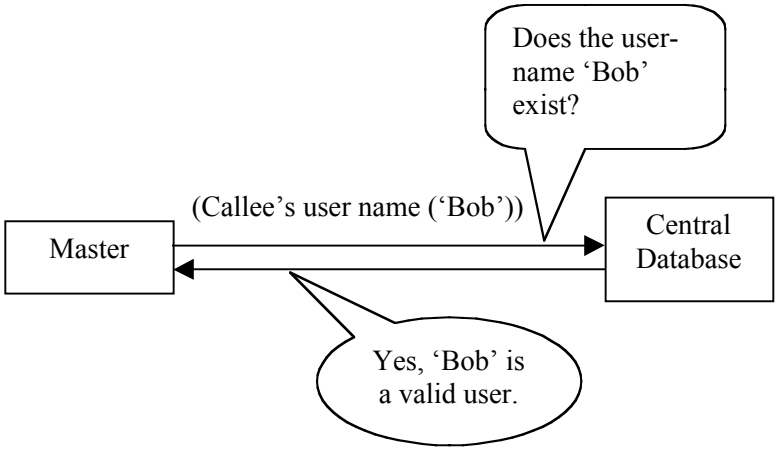
The callee's Agent may not necessarily be the caller's Agent. Therefore, the replying Agent may not have a session key established with the caller. Instead of obtaining a new session key or obtaining the caller's session key from its Agent, the callee's Agent poses as the Master server. The Master sends the session key, $K_{Caller/Master}$, to the callee's Agent, who encrypts the IP address of the callee with this key and then sends the reply to the caller. The caller thinks the Master server is directly responding to its request. Note that the Master encrypts the message sent to the Agent with $K_{Master/Callee-Agent}$. Therefore, the caller's session key with the Master is not compromised. Agents are trusted elements once they have authenticated themselves with the Master server. Figure 12 illustrates the protocol.

We can extend our protocol to provide a ticket for the callee so that caller-callee authentication is possible. The Master can generate a ticket for the callee and forward it to the

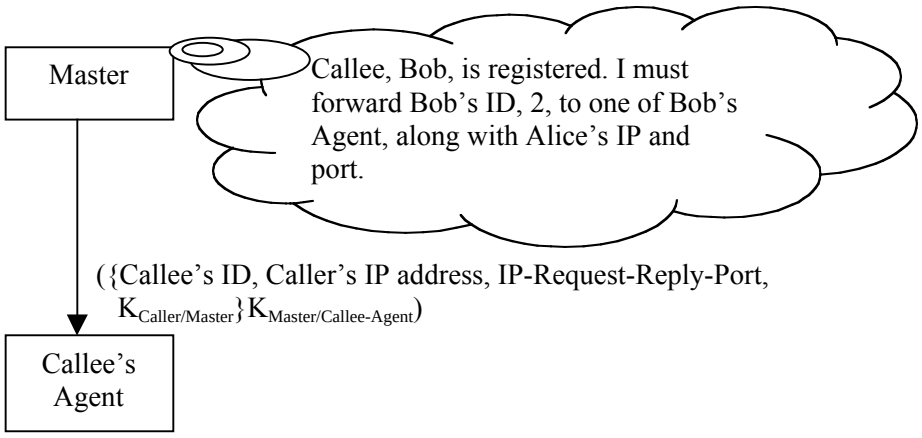
Agent, who can then include the ticket in its reply to the caller. The Master can generate a ticket such as, {{Caller's name, Caller's IP, Caller's Port, Timestamp}$K_{Callee/Master}$}$K_{Caller/Master}$.



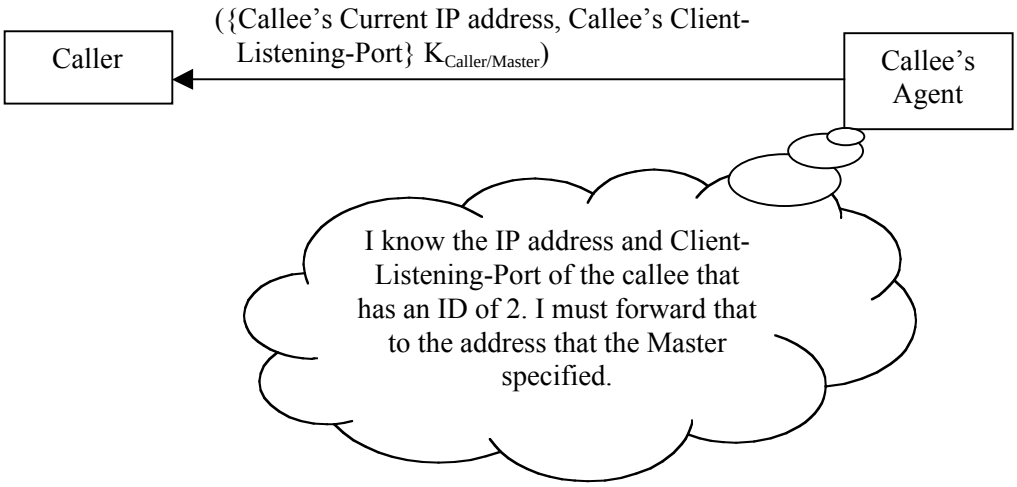
Step 1: Caller sends an IP-request to the Master



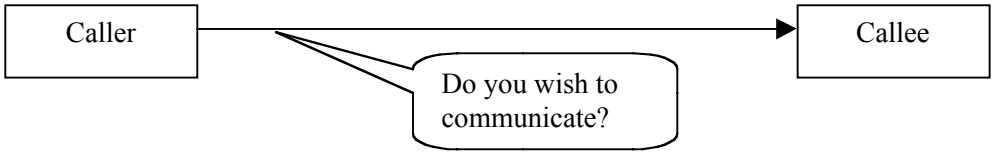
Step 2: The Master queries the Database for the validity of the Callee



Step 3: The Master forwards the IP-request to one of Callee's Agent



Step 4: Callee's Agent sends the Caller, Callee's IP and port



Step 5: Caller initiates a request to communicate with Callee

Figure 12: A successful IP address lookup with Encryption

5.2.5 Agent Heartbeats

Agents encrypt their IP address in the heartbeat message they send to the Master, Agent→Master: (Agent ID, {Agent IP address}$K_{Agent/Master}$). Using the Agent ID in the packet, the Master is able to retrieve the correct session key that it has with the Agent, to decrypt the IP address of the Agent.

5.3 Analyzing the privacy measures

This section discusses various ways a malicious intruder can try to eavesdrop on communication among the components of the directory or pose as one of the components. The goal is to prevent an eavesdropper from obtaining a client's IP address, and finding out the user

name associated with that client, from the messages exchanged between the components. If an eavesdropper can match a client's IP address with a user name, he will know the location of the client's user.

The use of client IDs, instead of user names, to communicate with the service after the registration process is a privacy measure, making it harder to learn the identity of a client's user.

It is also important to make it difficult to determine a user's location. Since an IP address determines the subnet, and subnets are often determined by location, we attempt to hide clients' IP addresses from eavesdroppers.

Since we encrypt the IP address when it appears in the payload of messages, the eavesdropper cannot obtain it directly from IP-update packets or Agent→Caller messages. As IP-update packets are sent via UDP, however, the intruder can obtain a client's IP address from the UDP header, and the plaintext client ID from the payload. Fortunately, it is impossible for the intruder to obtain the mapping between client ID and user name, because the user name is encrypted everywhere it is exchanged. Even a caller, after looking up a callee, is not given the callee's ID.

Trudy, a malicious intruder, cannot change the contents of any message because she does not have the session keys, so she cannot, for example, insert her IP address into an Agent→Caller message or an IP-update packet. Trudy can, however, modify the plaintext client ID in an IP-update packet and transmit the "corrupt" packet. This constitutes a denial-of-service attack; if Trudy changes Alice's ID in her IP-update packets, her Agents will assume that Alice has logged out. Moreover, if Trudy changes Alice's ID to Bob's ID, in her packets, Bob's Agents may have Alice's IP address for Bob. Fortunately, Agents compare the encrypted IP address in the payload with the IP address in the UDP header, and disregard the packet if there is a mismatch.

Trudy can register with the service and query it for Alice's IP address, and hence obtain Alice's Agents' IP addresses. Trudy can then eavesdrop near Alice's Agents and track changes to Alice's IP address. Hence, Trudy can obtain the user name to IP address mapping. The directory

does expose this mapping, but only to registered users, and only as an explicit query that might be logged.

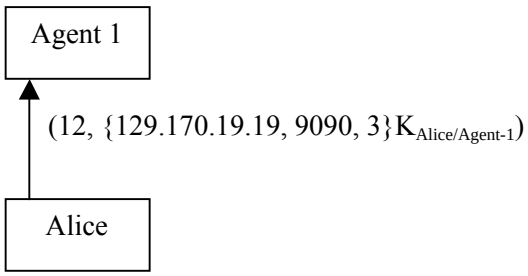
Suppose Trudy is an impostor, trying to fake a component of the directory. We consider seven different cases in which Trudy tries to act as a valid component of the system and how the system prevents such attacks.

1) Suppose Trudy sniffs the network and obtains an IP-update packet of Alice. Assume that Alice's client does not change her IP address and is sending the same IP-update packets. Trudy begins to replay Alice's packet, as is. Alice's Agent receives two identical packets, one from Alice, and the other from Bob. These extra packets do not change the state of Alice or the Agent. Trudy cannot change the packet because she does not have the session key.
2) Now suppose Trudy obtains one of Alice's IP-update packets and Alice logs out (see Section 3.5) of the system. Trudy begins to replay Alice's packet. One of her ex-Agents believes that Alice registered with the service again, whereas she did not. Suppose Trudy's machine obtains the IP address that Alice's client had, in hopes of receiving calls for Alice. If an IP-request for Alice comes, the Master determines that Alice is not registered, and so never forwards the request to Alice's former Agent. Thus, Trudy is unable to receive any calls intended for Alice.
3) Trudy's machine obtains Alice's IP address and Trudy replays Alice's IP-update packets after she logs out. Trudy now tries to make a call to Judy, pretending to be Alice. Trudy contacts the Master server to obtain the IP address of Judy, Trudy→Master: ({'Judy', Alice's IP address, IP-Request-Reply-Port}$K_{Alice/Master}$, Alice's ID). Trudy, however, needs the session key, $K_{Alice/Master}$, which she does not have (in fact the Master does not have it either after Alice logs out). Therefore, Trudy is unable to encrypt the message and cannot query the directory for an IP address. Even if Trudy replayed an older IP-request packet of Alice, the Master does not have the session key of Alice any longer and so disregards

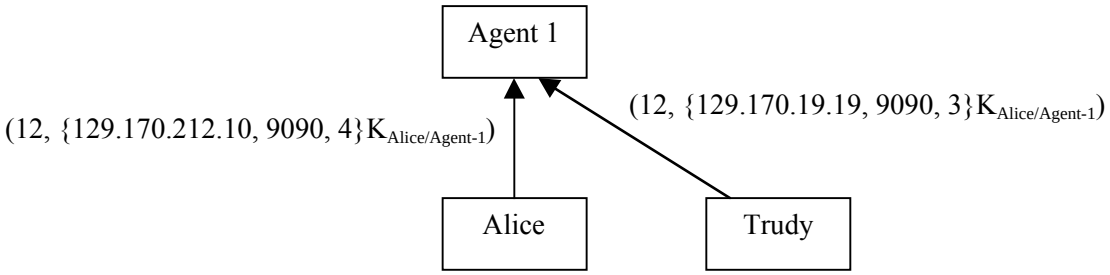
the request. Although Trudy has faked herself as Alice, she can do no harm. Note that Alice is not denied service when she registers again, as a new client ID is assigned to her.

4) Trudy obtains one of Alice's IP-update packets. Suppose Alice's ID is 12, IP address is 129.170.19.19, and Subnet Count is 3. Alice now switches subnets and her IP address changes. Her new IP address is 129.170.212.10 and her Subnet Count is now 4. Alice sends her new IP-update. Trudy replays Alice's old IP-update packet (Figure 13).

Alice's Agent compares the encrypted parts of both packets to its stored cipher of Alice. As Alice's new packet has a different cipher than the one the Agent has, the Agent



(a) Alice in her old subnet



(b) Alice in her new subnet and Trudy replaying her old IP-update packet

Figure 13: An impostor replaying an old subnet packet after a Client switches Subnets

decrypts it and observes that the Subnet Count field in the packet has incremented. The Agent updates its information about Alice. When the Agent decrypts Trudy's replay, the Subnet Count is lower than the updated value, and so it disregards the replay. Therefore, the system subverts Trudy's attempt to carry a denial-of-service attack.

When Alice switches subnets, Trudy can trap and discard all of Alice's new IP-update packets, obtain Alice's old IP address and replay Alice's old packets. Alice's Agents think that Alice's IP address has not changed, and so send Alice's old IP address to callers who query the directory for Alice. Hence, Alice is not able to receive any calls. This denial-of-service attack is a limitation of our system. Similar denial-of-service attacks are possible, especially if there is no caller-callee authentication.

5) Suppose Alice sends an IP-request to the Master to obtain the IP address of Judy, Alice→Master: ({'Judy', Alice's IP address, IP-Request-Reply-Port}$K_{Alice/Master}$, Alice's ID), and Trudy obtains this message. Now Trudy replays this message to the Master (Figure 14).

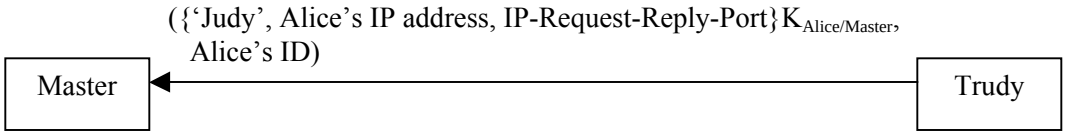


Figure 14: Impostor conducting a replay attack to obtain an IP address

The Master thinks that Alice is asking for Judy's IP address. It forwards the request to one of Judy's Agents. The Agent sends Judy's IP address to the IP address specified in the initial request, which is Alice's IP as Trudy cannot decrypt Alice's IP-request message to put her IP address in it, instead of Alice's. Suppose Alice logged out and Trudy managed to obtain Alice's IP address. The Master has no record of Alice and disregards the request. If Alice's machine had crashed, the Master has Alice's information, so construes the request as valid, and forwards it to one of Judy's Agents. Judy's Agent forwards the request to the specified IP address, and Trudy receives the reply, Judy-Agent→Trudy: ({Judy's IP address, Judy's Client-Listening-Port}$K_{Alice/Master}$), as Figure 15 illustrates.

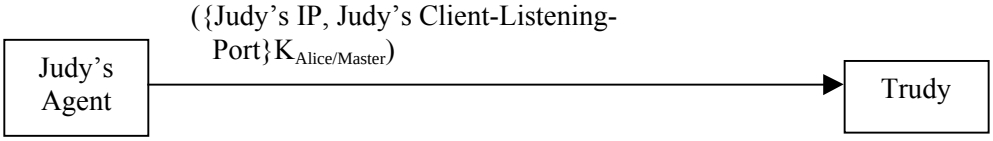


Figure 15: Impostor obtaining an IP address by a replay attack

Trudy does not, however, know the session key, $K_{Alice/Master}$, and is unable to decrypt the reply to obtain Judy's IP address. Though Trudy successfully posed as Alice, she cannot initiate any communication with Judy, and any other user for that matter.

6) When an Agent crashes, Trudy might obtain the IP address the Agent had. Trudy then starts a process to listen for IP-update packets on the port that the crashed Agent was listening (Figure 16).

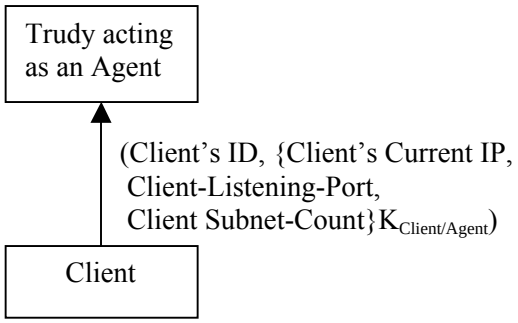


Figure 16: Impostor acting as an Agent

Trudy receives IP-update packets from clients but she cannot obtain the IP address of clients from them, because they are encrypted with session keys that the Agent shared with the respective clients. Hence, Trudy can do no harm.

7) Suppose Trudy obtains Alice's IP-update packet meant for Agent Y, and Y then crashes. When Y starts running again, Trudy replays the packet. Y is re-constructing its tables (see Section 4.2), and so asks Trudy (thinking it is Alice) to re-send the ticket that Alice obtained from the TGS. Even if Trudy got a hold of the message that the TGS sent to

Alice originally (Message 4 in Figure 8), Trudy cannot extract the ticket meant for Y, $T_{Alice/Y}$, since the message is encrypted with the session key between Alice and the TGS, $K_{Alice/TGS}$. Hence, Trudy is unable to comply with Y's request and Y ignores the replay packet of Trudy.

We realize that this is not an exhaustive study of all potential cases of a malicious attacker breaking the security of the system. Nevertheless, we feel that the security features of the directory handle most "reasonable" ways to attack.

6 Implementation

We developed a prototype of the directory service in C++ using libraries developed by Equivalence [7]. We did not, however, implement security and authentication features, making the prototype simpler to implement. We ensure that the components all use different ports. We chose high numbers for our ports so that the components do no interfere with other applications. In particular, we ensure that the H-series multimedia protocols (H.323 is one of them) do not use these port numbers. Since all components use unique port numbers, we can run the Master server, an Agent and a client on the same machine.

Apart from the agent and client heartbeats (see Sections 4.1 and 4.2, respectively), all communication uses TCP. UDP is better suited for heartbeats because of their frequency and small payload. Moreover, the packets should reach the destination as quickly as possible – especially IP-update packets – because of the volatile and time-sensitive nature of the data (client IP addresses) in them. We can afford to drop a heartbeat packet every now and then (as long as three packets are not dropped consecutively), but cannot afford to have the IP-updates reaching Agents with a delay. TCP is used for other communication between the components; namely, client and Agent registration, and the client IP-request process. If UDP was used for the IP-

request process, a client might end up waiting forever, not knowing that its IP request got lost. To avoid this, we would need to implement a timeout mechanism at the Application-layer level. TCP has a timeout mechanism in the Transport layer and so we rely on that to guarantee message delivery, at the expense of the higher overhead involved with TCP compared to UDP.

The Master and Agents send error messages to clients when they detect errors. Master→Client and Agent→Client messages in the implementation include an additional field called, "Error Code." If the Master and Agents process requests successfully, the "Error Code" field is set to "Success." Otherwise, the "Error Code" contains a description of the error that occurred, and the other fields are empty (set to zero). This approach simplifies the number of message types in the directory's protocols. When a client receives a response, it checks the "Error Code" field and processes the response accordingly.

The prototype runs only on Windows 2000 machines. The code is portable to Linux, with slight modifications.

7 Evaluation

This infrastructure is designed, specifically, to be scalable. The number of Agents can be varied according to the number of users. The system is flexible enough to automatically accept a new Agent while the system is running. We do not need to stop the system to add another Agent to it. Our load-balancing algorithm ensures that the Agents have roughly the same number of clients. Balanced load-distribution is crucial for scalability.

Since Agents are separate entities from the Master, the Agents can be distributed across subnets in the LAN. This has the advantage of a distributed network load. To localize network load within subnets, the Agent-assignment algorithm could be changed to factor in the location of users.

The error-handling and failure mechanisms included in the system ensure, to a great degree, that no client information is lost, provided a crashed Agent is restarted on the same host. The robustness of the system makes it reliable.

The widely deployed and thoroughly tested Kerberos protocol for authentication and encryption is the backbone of the system's security and privacy. Of course, this limits our system to LANs that have Kerberos deployed. Moreover, machines in the LAN must have approximately synchronized clocks (a known, but acceptable, limitation of Kerberos).

Though the system works well under many failures, it fails totally if the Master server crashes. We assume that the Master server never crashes. Ideally, there should be a built-in mechanism so that the system adapts even when the Master fails.

The model attempts to make the system a secure one. Despite the security features included in the directory, there is a possibility of a malicious intruder carrying out denial-of-service attacks (see Section 5.3, Case 4). The mode of attack mentioned in Section 5.3 may be hard to carry out, but it is worth mentioning.

We acknowledge the fact that this directory cannot scale beyond a LAN. Doing so entails clients sending UDP packets over long distances to their Agents. To deploy the system worldwide, we need to come up with a hierarchical name space like the one used for e-mail. The contents of the directory might live at a well-known hostname in each domain, and then there could be a simple syntax for identifying users: Ammar.Khalid@phone.dartmouth.edu. The system scales for a WAN, however, provided Agents are distributed to all geographic parts of the WAN, so that UDP packets do not traverse long distances, and the Agent-assignment algorithm factors in locality. An Agent reassignment algorithm and protocol is needed, however, to cater to clients that move from one part of a WAN to another, so that clients are kept "close" to their Agents. Moreover, such a protocol is necessary when crashed Agents cannot be restarted with their old IP address.

Despite these limitations, we strongly feel that the system is an efficient, scalable, robust, and secure one to deploy in wireless networks.

8 Related Work

This section explores other directory service models that exist, and shows how and why the model we propose is better suited to support service mobility.

8.1 The Dartmouth Name Directory

The Dartmouth Name Directory (DND) [8] stores information about all registered users of the Dartmouth Network, and can authenticate users with a secure-password protocol. It uses a single server and a TCP-based protocol. The DND architecture, however, assumes that a client does not contact the server frequently, at least not at the rate of once every minute. To support mobility, our model requires that each client updates its IP address at least once every minute (see Section 3.3 for when the IP address is updated sooner than one minute). The DND model is not scalable to support mobile users on clients that send packets every minute. One server cannot efficiently process thousands of packets every minute. To achieve scalability, we extend the centralized model by adding Agents. Assume a network of 10,000 users, 40% of whom are registered at any time (4000 clients), and each client updates one of ten Agents, instead of two Agents. Assume a 56-byte IP-update packet (40 bytes for headers). Each Agent then receives `4000*56/10*60 = 373` bytes per second. A single-server model entails `373*10 = 3730` bytes per second at the server. Clearly, as the number of registered users becomes large, a single server, such as the DND, becomes overloaded.

As each Agent caters to a subset of clients, and we can add more Agents as needed, no single Agent is overloaded. The Master server can then operate under the same assumptions as the DND server. The Master server only registers clients, and receives and forwards IP-requests

to an appropriate Agent. The frequency of communication with the Master server is similar to that with the DND server.

Another difference between the two models lies in the way authentication is done. A DND command is a two-step process; namely, the client sends the request to the DND server, which then sends a reply asking for the client's user to authenticate himself. Not only does a user authenticate himself every time his client issues a command to the DND server, but the latter allows lookups without authentication. For efficiency reasons, we feel that authentication should be the first step in the process, and should only be done once. The server should only begin to process a client's request after the client's user has been authenticated, for if authentication fails, the server can disregard the request. For these reasons, we chose to use Kerberos as an authentication mechanism in our model. In our model, therefore, a user authenticates only once, and his client cannot issue any requests until it has valid tickets for the service.

Another concern, especially on wireless networks, is privacy. The DND server sends replies (which contain users' information) in clear text. Replies from our directory service mostly consist of IP addresses, which are an indicator of a user's location. For privacy purposes, therefore, we cannot afford to have the service send replies in plaintext. Kerberos session keys are used to encrypt every network packet that is sent and received, thereby ensuring secrecy and integrity.

Though the DND model is clearly inappropriate for our purpose, it can be incorporated into our design. We can employ the DND as the central database, using the DND protocol. We would need to add encryption to this communication.

8.2 Domain Name System

The Domain Name System (DNS) [9] is a hierarchical, distributed database of hostname to IP address mappings. If each mobile host is assigned a hostname, then we could use the DNS

for looking up IP addresses of mobile hosts. This solution poses a problem. The DNS would need updating whenever a mobile host's IP address changed. For a small number of mobile hosts, this approach might work. It is not, however, scalable. Moreover, it would require assigning a hostname to every mobile host, and it would assume a user has a hostname; both are uncommon in a DHCP world, and the latter limits the possibility of a user using different mobile hosts. A central server would be needed to distribute session keys to ensure security. Doing this for a widely distributed database can be a challenge. Our model overcomes these difficulties, hence providing scalable and secure support to mobile users.

IP address lookups in our system resemble DNS queries. Most DNS servers handle queries recursively: when a server does not have the requested information, it forwards the request to another server. That server may forward it to yet another server, recursively. In our system, we have one level of forwarding, which is sufficient for our desired scalability.

Despite these differences, there are some features of the DNS that can be used in our directory service. We assume that the Master server never crashes. The DNS assumes that any server, at any time, can become unreachable. To rectify this, the DNS data is distributed widely, and replicated across several DNS servers. Moreover, hosts do a lot of caching in the DND model. A host contacts a DNS server if its cache does not have an entry for a hostname. Even though the IP address of a mobile host is volatile, a caller can still benefit from caching a callee's IP address, though a secure caller-callee authentication mechanism would be needed. The *cache consistency* problem would also need to be addressed.

8.3 Gatekeepers in the H.323 protocol

Our directory service provides support for mobile services, such as Mobile Voice-over-IP. The H.323 protocol [10] is widely used for Internet Telephony. An optional component of the H.323 protocol is a Gatekeeper. Terminals (clients) communicate with it using the Registration,

Admission, and Status (RAS) protocol, when it is present. A terminal uses the RAS protocol to register with a gatekeeper and to ask the gatekeeper to let it place a call. A gatekeeper is the central management component of the H.323 protocol, and provides the following main services:

- **Address Translation.** It maintains a database that maps aliases (user names) to network IP addresses. A terminal notifies a gatekeeper its alias and network address, when it registers with the gatekeeper.
- **Admission and Controlled Access of Terminals.** Based on registration privileges of a user, it may deny access to him. Moreover, it can deny service to users based on bandwidth availability.

When an H.323 client application starts, it registers (logs in) with a gatekeeper, either statically or dynamically. In static registration, the client knows the transport address of its gatekeeper a priori. In dynamic registration, the client sends a "gatekeeper discovery" multicast on the gatekeeper's multicast address. A gatekeeper then sends a confirmation back to the client, informing the client of its IP address. When a client wishes to make a call, it contacts the gatekeeper for permission; this process is known as admission.

A gatekeeper assumes that clients have fixed IP addresses (or at least infrequently changing addresses), whereas Agents expect the IP address of mobile hosts to change. Agents, therefore, require clients to frequently update their IP addresses with them. Since a gatekeeper manages a zone[7] and not the whole network, it is capable of handling frequent IP-updates. Mobility support by gatekeepers, however, is not a standard included in the H.323 protocol. The gatekeeper has many modules, one of which is a directory service. We feel that our system can serve as the directory service module of the gatekeeper, allowing the gatekeeper to support mobile users. Unless protocols for gatekeeper-gatekeeper communication are deployed (which they currently are not, but will be in version 3 of H.323), the directory module would contain all the users of the network instead of just the users of the terminals in that zone.

[7] A zone is a subset of terminals and other H.323 components that a gatekeeper manages.

A gatekeeper makes its zone centralized: the gatekeeper handles all incoming and outgoing calls. We feel that this adds extra overhead to a call. In our model, communication with the directory is minimized for making a call, and there is no communication with the directory when receiving a call. A caller directly makes a call itself, once it obtains the callee's IP address. This eliminates the directory service once the call is in progress. Moreover, terminal-gatekeeper communication is done using UDP, necessitating timeout mechanisms at the Application-layer level. We feel that client-Master communication should be done using TCP, since this delegates the responsibility of message delivery to the lower layers (the IP-request process uses TCP at each step).

Security is a chief concern. The H.235 recommendation (security recommendation for H-series protocols) [11] suggests terminal-gatekeeper authentication. It does not, however, propose a way of encrypting the communication between the two. Due to a lack of encryption, we feel that the gatekeeper model is inadequate for our purposes, as an eavesdropper can easily ascertain the identity and location (via IP address) of a user.

The gatekeeper model does have some advantages over our model. A gatekeeper can keep track of the number of minutes a user uses in call-time during a fixed time-period. The Master server, in our model, does not interpose on caller/callee data, and so cannot record such information, which might be useful for accounting purposes.

The gatekeeper model was not designed to support host mobility. We feel, however, that the features available in the gatekeeper model are useful. If we could mesh the two models together, we could have a powerful tool for Internet telephony, at least.

9 Future Work

In the light of the above discussion, we realize the drawbacks of our system and hope to extend the functionality that it offers. Here are some ideas for future work, mostly to address limitations that Section 7 describes.

9.1 Implementation of security features

The prototype of our model does not have the security features that we elaborate in this paper. We hope that someone carries on from where we left off, and implements security and authentication so that the prototype is a complete application of our model.

9.2 Robustness

We plan to add alternative mechanisms so that failure of the Master does not bring the whole system down. One approach is to have more than one Master server, and clients dynamically discover one of the many Master servers by multicasting a "Master discovery" packet when they register. This proposition is similar to the gatekeeper discovery process in the H.323 protocol (see Section 8.3).

9.3 Automatic management of Agents

Currently, Agents must be started manually. Ideally, there should be a process monitoring the number of registered users; it should add more Agents when usage increases, and remove Agents when the load is low. This would give the system the ability to automatically replace a crashed Agent.

Our model requires that a crashed Agent be restarted on the same host, because clients cannot be reassigned to different Agents. If we develop a way to efficiently reassign clients to different Agents when Agents crash, we can remove this limitation.

9.4 Addition of Non-Repudiation as a Security Feature

The directory does not assist in any communication between the caller and the callee, after the IP-request process. Consequently, the directory has no conclusive evidence that a certain caller called a callee. The directory should provide non-repudiation as a security feature. If Alice claims that Trudy called her, and Trudy repudiates her claim, the directory should furnish a proof to prove Trudy wrong.

10 Conclusion

This directory service is a scalable, robust, and secure solution to the problem of obtaining current IP addresses of people on mobile devices. With wireless networks being deployed rapidly, the need for such a system is imminent. The system can be improved if certain existing protocols are incorporated into its design. It can benefit from a protocol such as the DND, for communication between the Master and the central database. On the other hand, the H.323 gatekeeper can benefit from our techniques to support mobility, making it more versatile to the varying demands of network administrators.

11 Acknowledgements

I thank the following people without whom this project would not have been possible:

- Professor David F. Kotz, whose advice has been immense throughout this thesis. His forewarnings of potential pitfalls, and his solutions to seemingly insurmountable walls, have made this model a complete one.
- Equivalence, an Australian company that has coordinated the development of the OpenH323 project.[8] Craig Southeren, the co-founder of OpenH323, said, "The aim is to 'commodotise the protocol.' By giving the stack away, maybe we can stop everyone from obsessing over how to format the bits on the wire, and get them writing applications instead." We used the PWLIB classes from Equivalence in our prototype. Their well-defined APIs make C++ multithreaded and network programming a lot easier.
- G. Ayorkor Mills-Tettey. Ayorkor helped me greatly in understanding the nuances of the PWLIB classes when they seemed overwhelming.
- Steve Campbell and David Gelhar for their help on DND.

References

[1] Charles E. Perkins and David B. Johnson. Mobility Support in IPv6. In *Proceedings of the Second Annual International Symposium on Mobile Computing and Networking*, pages 27–37, White Plains, NY, November, 1996.

[2] G. Ayorkor Mills-Tettey. Mobile Voice Over IP (MVOIP): An Application-level Protocol. Dartmouth College, 2001.

[3] ITU-T. Edgar Martinez (Motorola). Annex-H (User and Service Mobility in H.323). October, 1999.

[4] J. G. Steiner, C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the USENIX Winter Conference 1988*, pages 191-202, Dallas, Texas, USA, February 1988.

[8] http://www.openh323.org/

[5] Srikant Sarda. Loosely Synchronized Clocks in Kerberos. http://web.mit.edu/6.033/1998/www/reports/r05-ssarda.html

[6] Armando Fox and Steven D. Gribble. Security on the Move: Indirect Authentication Using Kerberos. In *MOBICOM'96 Proceedings of the Second Annual International Conference on Mobile Computing and Networking*, pages 155–164, 1996.

[7] Equivalence Pty Ltd. The OpenH323 Project. http://www.openh323.org/

[8] David Gelhar. Protocol Specification: DND Server Interface. Dartmouth College, January, 1998.

[9] Andrew S. Tanenbaum. *Computer Networks, Third Edition*, Upper Saddle River, NJ: Prentice Hall, 1996. Pages 622–630.

[10] The International Engineering Consortium. H.323 Tutorial. http://www.iec.org/tutorials/h323/

[11] ITU-T. Security and encryption for H-Series (H.323 and other H.245-based) multimedia terminals. Version 2, November, 2000.