An Instant Messager Application based on OpenHash
6.824 Distributed Computing
May 6, 2004

Abstract

We propose an open, distributed instant messaging architecture that provides end-users with the freedom of defining their own feature set and allows independent developers to extend the application with their own implementation. To shorten the development cycle, we implemented OpenHash Instant Messager Library (OHIMlib), a library for presence detection. OHIMlib utilizes OpenHash, a publicly available Distributed Hash Table (DHT) as its distributed storage medium and directory service. To illustrate the usability of OHIMlib, we implemented a simple chat client. Other developers may choose to implement their own client on top of OHIMlib and thus the system achieves customizability, extensibility, and ease of development.

1. Introduction

With the growth of the Internet, instant messaging has reached every corner of the globe from small homes to large enterprises to mobile devices. AIM, the messaging service provided by AOL, represents one of the largest instant messaging communities in the United States with over 50 million active users each month [1]. Instant messaging empowers users with a convenient tool to communicate with their peers at any time regardless of their geographic location. An Internet connection and an appropriate IM client is all that is necessary to maintain one's presence with hundreds of people and to have multiple spontaneous private conversations in parallel.

The most popular IM services currently available to the general public, such as AOL AIM and MSN Messenger, are all proprietary software. These applications have a predefined set of features that may be desirable to some users and undesirable to others. Furthermore, the closed architecture of these systems disallows outside developers to extend the IM functionalities with their implementation. The lack of *customizability* and *extensibility* in current IM architecture motivates us to design an open, distributed system. Furthermore, we wish to achieve *ease of development* by black-boxing some of the underlying protocol and mechanism, and providing an API that allows independent developers to quickly build and deploy their customized IM clients with relatively little time and effort.

While the various commercial clients such as AOL AIM and MSN Messenger have different sets of features and user interface, we argue that these commercial clients can be implemented on the same presence detection engine. Presence detection is defined in RFC 2778 [2]. The engine provides a user with means to find, retrieve, and subscribe to changes in the location and on-line status of other users. Each user has a list of her friends' usernames. As friends log on and off, the system automatically updates the list. A user can then identify friends who are online and initiate conversation. We want to develop a platform that provides an interface for a presence detection engine, so an independent

developer may build a specially tailored client on top of this platform without having to maintain her own set of servers. The presence detection engine must support the following functionalities:

1. The system must remember each user's buddy list because a user may log in from a different host each time.

2. The system must be able to locate a user while she is logged on, so that other users can find her.

3. The system automatically updates a user's buddy list when her buddy logs on or off.

A summary of our goals follows:

1. Customizability - This is a long-term goal where the end result is an IM application that defies current proprietary clients by allowing the end-user to define the feature set.

2. Extensibility - The average developer should be able to extend the Presence Detection engine to achieve her personal goals, for example, developing a client that behaves more like Yahoo Messenger than AOL IM.

3. Ease of Development - The API should be easy to use so that other developers can pick it up and get started quickly.

The rest of the paper is structured as follows. Section 2 introduces the design of the presence detection engine, called the OpenHash Instant Messager Library (OHIMlib). Section 3 gives an overview of our implementation of a client based on OHIMlib and section 4 discusses future research. Section 5 concludes the paper.

2 Design Overview

Conceptually, we divide the functionality of an instant messaging application into two parts: presence detection and all other functionalities such as text messaging. The presence detection side of the application can be implemented by using a Distributed Hash Table (DHT). A DHT serves as an efficient and scalable distributed storage medium and lookup service. With this distributed mechanism in place to provide presence detection, all other functionality can be placed within a client application. Client side functionality is deliberately left unspecified. The reason is that the mechanism to locate buddies provides a general enough infrastructure to establish a point-to-point connection from which almost any feature can be implemented.

To achieve the goals we specified for our system, we need a DHT that is accessible to the average developer. However, algorithms for building DHTs and applications that use DHTs are still very much a research topic in the academia. Keeping a DHT running

continuously can be a daunting task if the developers are not familiar with the DHT code and deploying and testing a DHT-based application requires significant resources that are often unavailable to outside developers [4]. OHIMlib achieves a DHT-based distributed architecture by using OpenHash, a publicly available DHT service. The OpenHash interface provides an abstraction barrier that insulates the client from the algorithms and hardware involved in the DHT. It defines a clean boundary between the DHT code and the application code through a minimal put() and get() interface allowing developers to write their application without worrying about the intricacies and complexities involved in building and maintaining a DHT.

The rest of this section discusses the design of the presence detection engine, OHIMlib.

2.1 Presence Detection

As discussed in section 1, presence detection involves three main responsibilities: storing a user's buddy list, determining the online state of each buddy, and locating a buddy.

First, a user must register for a unique username, add entries to her buddy list, and save it to OpenHash. With the buddy list stored on OpenHash, the user can log on from any other machine that can connect to OpenHash. To connect to OpenHash, each OHIMlib client keeps a record of several nodes in OpenHash. The client selects the OpenHash node with the lowest latency.

Second, presence detection requires a mechanism to display the online status of buddy in the buddy list. As each buddy signs on and off, the user is notified and the online status of the buddy is updated. Later, we account for the situations where a buddy's client machine fails before sending the notification and where a “buddy relationship” is not mutual: A's buddy list contains buddy B but B's buddy list does not contain buddy A.

Third, the user must have a method of locating a buddy. When the user chooses to communicate with a buddy, there must be a way of locating that buddy and directing messages to her machine.

2.1.1 Keys and Values

Before we proceed, we introduce how data is stored on OpenHash. The presence detection engine stores two separate pieces of information to OpenHash: the buddy list and the user's current location. The buddy list is mapped to the key defined as username concatentated with “'s buddy list”. The location is mapped to the key defined as username concatentated with “'s location”. For example, for user Milhouse, his buddy list is mapped to the string “Milhouse's buddy list” and his location is mapped to “Milhouse's location”. Figure 1 shows a table describing these mappings.

key	*value*
Username + "'s buddy list"	Byte represenation of the buddy list
Username + "'s location"	Byte representation of the location

Figure 1. The key/value mappings for buddy list and location.

2.1.2 Creating New Username

To create a new username on OpenHash, a user must register this username with a password. This process checks that a buddy list and location do not already exist for the username. If this process succeeds, a blank buddy list and location are stored on OpenHash and write-protected with the password. The purpose of storing these entries is to prevent another user from claiming that username. If either buddy list or location already exist for the username, registration fails.

Currently, OpenHash does not support the protection of written data thus currently usernames are not exclusively protected for a given user. This security issue will be discussed in Section 4.

2.1.3 Modify Buddy List

The user can add or remove a buddy from her buddy list. To do so, the client application simply creates a new buddy list containing the updates and stores the list on OpenHash.

2.1.4 Sign On

To sign-on, a user enters her username and password. Once the engine connects to an OpenHash node, it first retrieves the buddy list. The engine then reconstructs the entire buddy list and sets the current location of the user.

2.1.5 Sign Off

When a user signs off, the engine sets a blank location to indicate the user is not signed on.

2.1.6 Notification of Buddy Sign-ons and Sign-offs

The buddy list should reflect the online status of each buddy as accurately as possible. There are two ways of determining the online status of a user. First, the user may notify other users when she signs on or off. When the user signs on, her client queries OpenHash for each buddy's location. For each valid location, the client tries to connect and "handshake" with the remote client. The handshake is a general process that verifies the identity of a user to the remote buddy and the remote buddy to the user. The implementation of the handshake is left to the developer. If the handshake succeeds, the buddy's online state is updated. Similarly, when the user signs off, her client notifies all buddies who are online.

The problem with this notification scheme is that if, for example, Lisa is on Milhouse's

buddy list but Milhouse is not on Lisa's buddy list, then Milhouse would not receive notification about Lisa's activity. However, we believe this is uncommon and is tolerable because Milhouse has other ways of finding out Lisa online status. To overcome this problem, the client periodically polls OpenHash to see if a buddy's location has been updated. To reduce load on OpenHash, if Milhouse knows of Lisa's locations, which suggests that Lisa is online, he can poll Lisa directly. For example, an IM Ping could be used that not only checks if Lisa's machine is online but also checks if Lisa is running an OHIMlib based client. If Lisa does not respond, it may be that Lisa's client application has crashed. The result is that her location becomes stale because it is not updated on OpenHash. For the offline buddies, a client may continually poll OpenHash for each buddy until the buddy signs on.

3. Implementation

OHIMlib consists of a set of interfaces and abstract classes that provide an abstraction barrier around the operations involving OpenHash and a set of abstract methods for operations that are left to the client to define. Client functionality includes specifying the protocol for exchanging data between two buddies.

To demonstrate how a IM client can be built using OHIMlib, we implemented “Basic IM,” or Basic. Basic defines a network protocol for communication between pairs of clients. 'Basic' features a graphical user interface, buddy sign-on and sign-off notifications, and text messaging functionality. However, it is very rough in its current form and is not meant to replace any commercial IM client. Instead, this particular implementation demonstrates how an IM developer can implement the abstract methods in a way to satisfy particular requirements.

3.1 Presence Detection

The goal of this library was to provide an interface that was extensible enough to build any IM client application on top of it. To do this we created an abstract class with methods that provide essential IM functionality and methods that a developer must override before they can use the class.

First, certain methods like signOn, signOff, connect, commitBuddyList, and register are included in the class to provide the basic functionality required to implement most IM applications. register, signOn, signOff, and commitBuddyList are relatively straightforward. They behave as described in section 2.1. connect gets the location of a buddy off of OpenHash and attempts to create a socket connection to that location. These functions make no assumptions about the structure of BuddyLists. The developer is expected to write her own methods that convert BuddyLists to and from byte forms so that the BuddyLists can be stored in OpenHash. For example, in our implementation we had the byte form be a list of buddy strings delimited by a comma and encrypted by the user's password. Thus, by leaving these byte conversion methods to the future developers we give them as much control as possible about things like security.

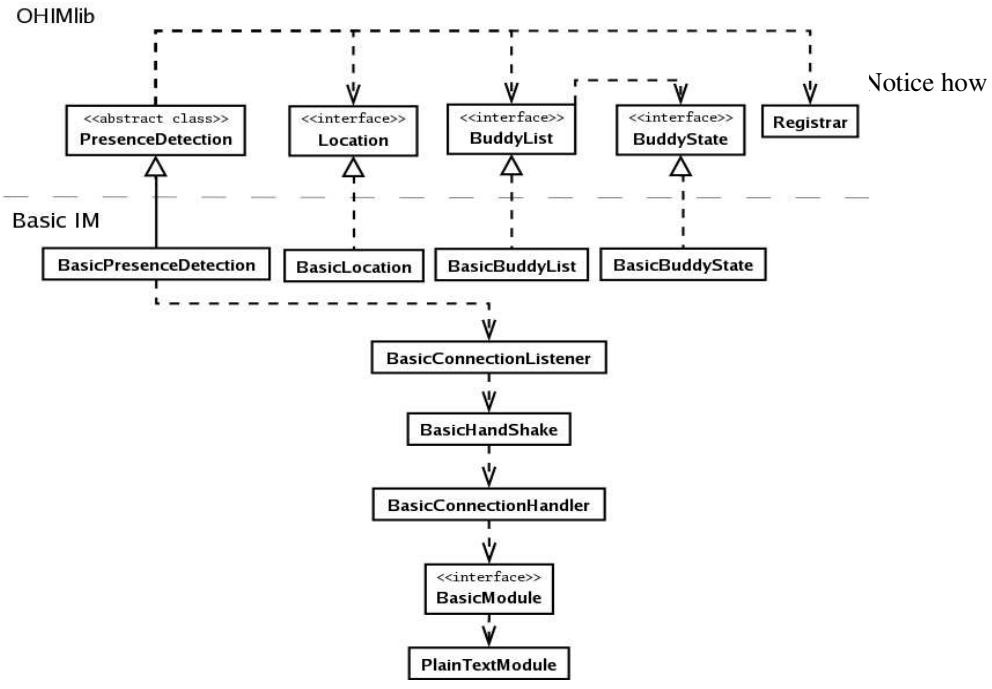
In addition to these basic methods, we defined abstract methods: signOnPost and signOffPre. These methods are fairly straightforward. They are called after signOn and before signOff respectively. Thus, a developer can use these methods to start and stop certain client-side operations. We decided to use these methods to provide any developer the flexibility they might need in creating a client application. For example, if the user can start pinging all the users on their BuddyList every so often in the method signOnPost.

3.2 Interface for Client Functionality

Many approaches exist to connect buddies and exchange data. AOL IM clients direct all text messages to the server, called Oscar, where the messages are routed to the destination buddy[3]. On the other hand, without a server such as Oscar, our implementation of Basic establishes a TCP connection between each buddy pair. In the OHIMlib interface, we leave the specification open to the developer.

3.3 Basic OHIMlib-based Client

We illustrate the extensibility of OHIMlib by implementing a basic instant messaging client (Basic) based on OHIMlib, the presence detection engine. Basic allows the user to register, sign on and off, modify the buddy list, and exchange text messages with buddies via a graphical interface. We concretize the Presence Detection abstraction in BasicPresenceDetection and implement the location, buddy list, and buddy state interfaces in BasicLocation, BasicBuddyList, and BasicBuddyState, respectively. Figure 2 shows a diagram describing these object relations.



While OHIMlib provides methods for storing and retrieving serialized buddy lists and locations in OpenHash, it is up to the client to decide how the data should be serialized and parsed. BasicPresenceDetection provides this layer of translation. In addition, it encrypts the data before invoking methods for putting the data into OpenHash to prevent other users from reading the buddy list. This demonstrates that it is possible to add security features to the system through the client implementation.

In Basic, each pair of buddies is connected through a TCP stream. During sign-on, Basic checks the online status of each buddy on the buddy list and initiates a TCP connection for each online buddy. It also starts up the BasicConnectionListener, which will listen throughout the session for incoming connection requests from newly signed on buddies as well as users not on the list. All these events take place during signOnPost. Once Basic determines that a buddy is online or receives a connection request, it creates a new BasicConnectionHandler for the buddy. BasicConnectionHandler uses the BasicHandshake module to facilitate a handshake exchange which leads to the establishment of a connection upon mutual consent. Finally, Basic polls OpenHash periodically to check for buddies' online status. To improve efficiency, Basic directly polls those buddies with whom TCP connections have previously been established.

Basic is capable of supporting multiple messaging protocols by delegating the message handling responsibility to protocol-specific modules. Each protocol is identified by a unique epoch, which is a randomly generated big number. Every message must be tagged with the epoch of the protocol used. A BasicConnectionHandler has modules for those protocols it supports. All modules have a common interface --- the handle(data) method. When a buddy's BasicConnectionHandler receives a message, it dispatches it to the appropriate module. Basic currently supports the plain text messaging protocol. The PlainTextModule simply echoes messages received to the GUI and forwards the user's input text to the buddy.

4. Future Research

As mentioned in section 3.2, commercial IM clients route messages through a central server, for example, AOL IM routes all text messages through Oscar. Without a server like Oscar to route messages to a specific buddy, 'Basic' connects each buddy pair directly through a TCP connection. An IP address is required to establish a TCP connection. However, clients that are located behind a NAT firewall does not possess a globally unique IP address. It is difficult for clients outside of a NAT to connect to clients within a NAT [5]. Coping with this difficulty is a challenging problem and is beyond the scope of this paper.

4.1 Security

Currently our implementation of the IM client has several security holes. We tried to leave the design extensible enough however that security could be added by developers later if they wanted it. For example, right now it would be very easy for one buddy to impersonate another by learning and hacking our protocol. However, if a developer wanted to make this

more secure, they could add in some authentication or encryption protocols to our existing protocols.

Another problem that the system currently has is that a user can overwrite another user's values on OpenHash. Currently, because of the stage of development of OpenHash, there is no easy fix to this problem. We spoke with the OpenHash developers and they said they are working on a fix to this problem that may involve cryptographically signing the values put to OpenHash so that when a user gets them they can be verified to belong to that user. Once this solution gets implemented in OpenHash it should be incorporated into our OpenHash library.

5. Conclusion

We described and implemented the presence detection engine OHIMlib. To illustrate some potential uses of OHIMlib, we implemented an IM client supporting basic text messaging. We intentionally avoided defining client attributes and focused on standardizing a general presence detection interface. The strength of our design lies in its extensibility. We believe that OHIMlib makes it possible to build IM clients with any application-specific functionality. Furthermore, this OpenHash-based framework allows independent developers to easily implement and deploy custom IM clients with minimal infrastructure setup. We hope our project will help foster an open community of IM developers and enthusiasts dedicated to driving the technology and the culture forward.

Acknowledgment

We are grateful to Brad Karp for his guidance on OpenHash. We wish to thank Sanjit for inspiring us to pursue this project. Special thanks to Professor Robert Morris for providing us valuable feedbacks each step along the way.

References

[1] Aim Developer. http://www.aim.com/developer.adp?aolp=

[2] M. Day, J. Rosenberg, H. Sugano. “A Model for Presence and Instant Messaging” RFC 2778, February 2002.

[3] FAIM: FAIM/AIM/Oscar Protocol. http://aimdoc.sourceforge.net/OSCARdoc/

[4] Brad Karp, Sylvia Ratnasamy, Sean Rhea, and Scott Shenker. Spurring Adoption of DHTs with OpenHash, a Public DHT Service. *Proceedings of the 3nd International Workshop on Peer-to-Peer Systems (IPTPS '04)*, February 2004.

[5] Egevang K., Francis P., “The IP network Address Translator (NAT)” RFC 1631, Cray Communications, NTT, May 1994.