Distributed Instant Messaging System

Lauren Arpin
Matthew Shea
Yi-Chin Sun

Vanderbilt University
Spring 2012

Table of Contents

Description

The Distributed Instant Messaging project is an attempt to provide a secure and decentralised method for communication. With recent changes in attitude towards the Internet and data networks in general, creating a system that protects its users from interference is becoming more and more important. Several chat systems exist already, but none provide a combination of decentralisation and public key encryption, which DIM is attempting to provide. Furthermore, many of these systems are inherently difficult to set up, making them unappealing to a general audience. The primary goal of DIM is to provide secure communications in a way that is completely hidden from the user.

The system operates by using the extended distributed hash table library TomP2P. By using a distributed hash table, users can store their data in the network with inherent resistance to shutdown and censorship. When a user connects to the network, they update their entry in the distributed hash table with their public key, online status, and various other information. When someone wishes to communicate with this user, they simply retrieve their information from the network and send an encrypted message.

Components

Client

Purpose:

The client serves as the end-point of the chat system and peer in the network

Responsibilities:

Connection to Tracking Network
Connection to Other Clients
Message Encryption
Registration of User Accounts

Well-Known Peer

Purpose:

The well-known peer serves as a gateway into the network. Any node can be used to connect to the network, but the well-known peer provides a way to easily spread your network.

Responsibilities:

Maintain a distributed hash table
Accept Connections

Client Structure

User Interface

The User Interface displays messages from other users and provides a method for sending new messages to members of the network. The User Interface uses the Contact Manager as the model to render.

Contact Manager

The Contact Manager maintains all data regarding each contact: active conversation, network status, rendering order, etc. When a message is sent from the User Interface, it passes through the Contact Manager which determines the recipient, instantiates a DIMMessage class, and populates it with the proper data. Once the message is created, it is passed to the Network Service.

Network Service

The Network Service consists of two primary components: the Sender Thread and the Receiver Thread. When TomP2P notifies the Network Service of a new message, it is placed on the Receiver Thread's queue for passing to the Contact Manager. When the Network Service receives a message to be sent, it accesses the KeyManager and passes the DIMMessage to it, where it is encrypted and returned to the Network Manager. Once the message is encrypted, it is serialized into the JSON Instant Messaging Protocol and sent to the TomP2P library for delivery.

Key Manager

The Key Manager manages all encryption keys. It contains a cache of keys which have been fetched from the network to make message sending faster. If the destination user's key is not found in the network, it will make a request to the Network Service to return the user's data. Once the destination user's public key is obtained, an AES key is generated. The message is encrypted with the AES key, and the AES key is encrypted with the destination user's public key. These are then passed back to the Network Service for delivery.

TomP2P Architecture and Integration

Description

TomP2P is an extended distributed hash table library for Java. The system automates the creation of a distributed hash table in an application by usin Java NIO to handle multiple concurrent connections cleanly. The library supports multiple values for each key, which is perfect for this project since a key can be a user's identity, and the values can be such things as the user's public key, status, and PeerAddress.

Each peer in the TomP2P network is assigned a PeerAddress value upon connecting to the network. This address is used to uniquely identify a peer and send directed messages through the network. TomP2P operates on Futures, which are objects that represent a future state of the network. Each Future has an `awaitUninterruptibly()` method which can be used when the program requires the network to be in that state.

Integration

The TomP2P library will only be accessible by the NetworkManager class, and there are a few basic functions that need to be performed.

- Connect to the Network

```
InetSocketAddress inetSock = new
InetSocketAddress(InetAddress.getByName("192.168.1.20"), 4000);
FutureBootstrap fb = peer.bootstrap(inetSock);
fb.awaitUninterruptibly();
```

- Send a Directed Message

```
FutureData futureData=p1.send(p2.getPeerAddress(), "hello");
futureData.awaitUninterruptibly();
System.out.println("reply ["+futureData.getObject()+"]");
```

JSON Instant Messaging Protocol

Specification

All messages are serialized as a JSON string. The String has several required fields and some optional fields. Below is the current draft of the protocol specification.

There are plans to add a great deal of functionality including file sharing to the network. With this, users would be able to directly pass encrypted data to one another, such as a picture. The current draft of this system would store data in the network with a hash key and an expiration date. When a file is sent, it is encrypted with an AES key and stored in the network. This key is then sent to the destination user along with the hash key, which allows for data retrieval.

Required Fields

- `[int]       type      | The type of the message`
- `[string] to    | The destination user name`
- `[string] from  | The sender user name`
- `[string] message   | The message encrypted with AES and serialized in Base64`
- `[string] aeskey| The AES key encrypted with the receiver's public key`

Testing Plan

Since much of the networking is handled by TomP2P, the only problems should be proper port forwarding and access. At least three machines will be used to test the system which all dual-boot Windows and Linux. Testing for Mac OSX is desired, but the availability of machines could be a hindrance for the time being.

<u>Scenarios</u>

- All clients on the same local network
- Some clients behind a router and some not.

<u>Specific Checks</u>

- Packet monitoring with Wireshark
- Encryption checks
- Directed messaging
- Network updates
- Network load testing
    - Rapid sending of messages
    - Sending of large messages

<u>Results</u>

- All encryption works properly
- Directed messages arrive
- Keys are properly stored in the network
- No network load testing was done
    - Other uses of TomP2P promise good scalability

Appendix I

Required Libraries

Dependencies

- Google GSON 2.0
- Commons Code 1.5
- TomP2P 4.0.3
- Netty 3.2.7
- Simple Logging Facade for Java API 1.6.4
- Simple Logging Facade for Java Simple 1.6.4
- Google Guava 10.0.1

Platforms

- Linux
  - Operates Properly
- Windows
  - Untested
- Mac OSX
  - Untested