FLAME CHAT REPORT

Brought to you by PixelCollider

Jerry Fan
nano@pixelcollider.net

1 Introduction

This report details the process and implementation of a prototype secure chat and social platform aimed for business. The client (company CEO) has requested that a secure communication system is developed for the purposes of protecting confidential company material when executives communicate with each other over the system from remote terminals.

A proposal for a peer-to-peer social network was made by the manager and accepted by the client.

2 Requirements

The minimum requirements for the system have been met with the product. Users of the application are able to securely log in and see a list of online users. They may choose a user from the list to send a message or file to, and receive replies back from the user. The user is able to change their profile page as needed, and view the profiles of other people on the list.

Additionally, inter-application encryption is implemented between applications. Hashing is also implemented. This provides additional security and integrity checking on top of the P2P model.

Several usability enhancements were made in addition to the basic requirements. Features such as search, embedded media, user statuses etc. have been added.

3 Features

As discussed briefly previously, several features were developed to enhance user interactivity and data transmission functionalities. These are categorized and listed below.

Security/Integrity	**User Interface**	**Other**
Message/file encryption	Live conversation feed	Database models
Message/file hashing	Profile/message search	Multithreading on requests
User blacklisting	User status indicator	Legacy support (fallbacks)
Rate limiting	Page routing with templates	Offline messaging
	Progress and loading indicators	

4 Developed System

4.1 Authentication

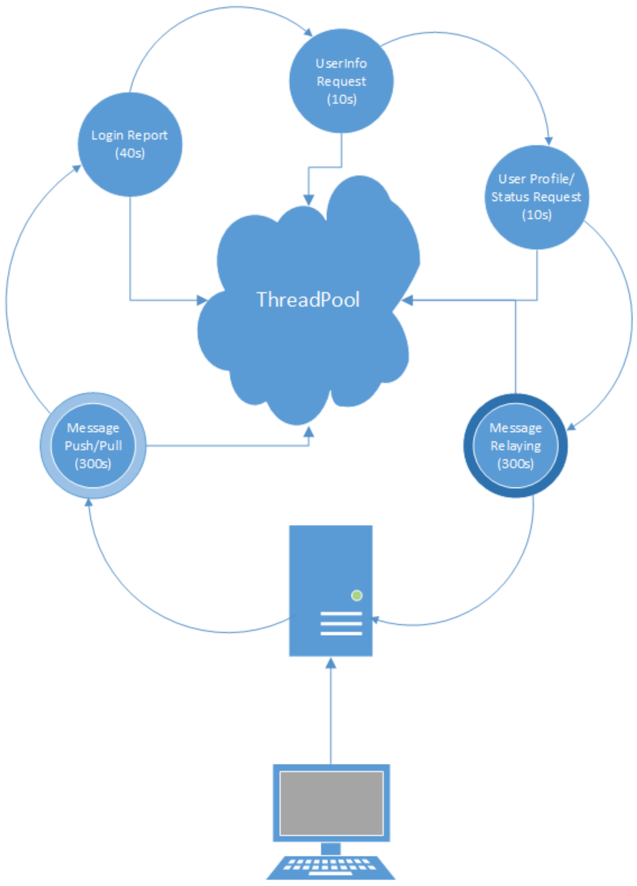
A local/remote hybrid authentication system was developed to take advantage of database caching so that in the event of a login server outage, the application could still function at reduced capacity.

The application periodically checks for the presence of the login server along with the login report. If the login server is detected to be offline, a flag will be raised that switches the application to use the cached user list built from previous login server requests. Using this user list is inefficient as it requires every known user to be tested for an active status, and no new users can be added during this time.

A local authentication mechanism is also available using cached usernames and password hashes. Other P2P functions will remain the same as they do not use the login server. Authentication is verified via a session system provided by cherrypy.

4.2 Local Relay API

All local API functions treat the python server as a relay server for multi-user support. The server does not belong to any one user, and as such, is not designed to keep a connection to the login server on behalf of a user. Instead, logged-in users are kept in sync with remote peer servers and the login servers via a server sent event stream initiated by client-side JavaScript. This connection indicates that a user is currently accessing the system, and drives an internal upkeep loop on the python server based on either shared or session timers.



Internally, the application follows a Model – Controller architecture, with API endpoints mapped to separate controllers for each model. Client side JS is driven by polling.

4.3 Remote API

The remote API is standardized as per application specifications. All incoming request data are mapped to data models as defined by the remote API endpoint parameters, with metadata stored as metadata models that accompany each data model. This ensures that transactions with remote servers are as quick as possible, and keeps most of the processing in the local API to improve overall network responsiveness.

5 Challenges

5.1 Concurrency

Due to the large amount of web requests required for server synchronization, the use of concurrent threaded HTTP requests was required to produce a responsive server application. The initial issue with concurrency came from using cherrypy sessions. If a request is sent to the server, it would automatically trigger a session lock, preventing any further requests from being processed before the current one ended. Manually unlocking the session before any real processing was done solved this issues on processing heavy endpoints such as the stream endpoint.

Further issues were introduced with concurrent programming in the upkeep cycle. All the upkeep cycle workers run thread pools to request data from external sources or to process internal data. However, when the upkeep workers were threaded, this produced many concurrency errors from desynchronized object access. This prevented the upkeep cycle from not blocking further upkeep tasks if a previous one takes longer than expected. This problem with mitigated with reduced timeout values on requests (which were multithreaded themselves, so they only took as long as the timeout in many cases).

5.2 Multi-User Relay System

Designing a multi user system proved to be challenging. Because the server did not belong to any one user, the server had to be developed in a way that any user's data could be stored and retrieved by remote and local users. Messaging and file sharing were the hardest functions to build with this in mind. To properly decrypt received messages and files required the server to know what messages it owned. Without this ownership, the server cannot determine whether the recipient is on that machine and whether it has the private key to decrypt the message should it be encrypted.

To resolve this issue, the server treats itself as a relay server, with all messages sent in being marked as 'send'. If the server finds the user at another IP address, it will forward the message to that IP. Otherwise, if the user has the same IP as the login server, the message is untouched. When the message is retrieved client side, the server marks inbound messages as 'store'. When a message is sent from the client side, the server marks the message as 'broadcast'. 'Broadcast' messages are intended to be sent to all users for relaying unless the recipient is currently reachable on the network. Sent broadcast messages are downgraded to 'send' unless they have been directly sent. In this system, the only distinction between a remotely created message and a locally created message is it sends action, bypassing the issue of ownership. However, the downside is that if sending fails, no errors can be returned to the original user. This problem is mitigated by the upkeep cycle, where extensive checks take place, to ensure that messages are sent in a deliverable form.

6 Tools Used

6.1 Python & CherryPY

Whilst python is a suitable high-level language for implementing a server side web application, cherrypy lacks in basic built-in functionality compared to other libraries that are able to implement a RESTful API, to better synchronize data models and associated actions with the API endpoints. The lack of a built-in router means that python objects must be used to provide endpoints, which may result in more objects being created than needed, and redirecting to a different endpoint requires an object reference. Overall, cherrypy is suitable for building a prototype web application, but a more extensive library should be used to make the application easier to maintain and provide more functionality.

6.2 Client-Side HTML/CSS/JS

Client side options for web applications are limited to these, so there are no alternatives to choose from if browser support is needed. Overall JS is just as flexible as Python in terms of extensibility and currently has a large number of external libraries to provide additional functionality. This has been heavily leveraged in the developed client-side solution, which uses angular and several angular plugins to provide functionality such as the status indicator and sticky chat scrolling. Maintainability on static sites is the biggest issue, which can be resolved by using a build system such as Webpack.

7 Design Considerations

7.1 P2P Methods

A peer to peer system is required if end to end encryption is desired. The client's goal is to prevent third parties from intercepting transmissions. This risk is greatly reduced if direct peer to peer communication is used. Furthermore, a peer to peer system can potentially provide better data integrity by hash confirmations between multiple peers. However, the P2P model introduces greater network complexity compared to more traditional central server networking, which may hinder the speed of communications across users.

7.2 Protocol

The prototype protocol lacks in the utilization of existing standards such as HTTP methods and HTTPS. The protocol at the moment does not serve as a good protocol to take into production, as it requires more standardization, especially in terms of the encryption and hashing standards used. If single user peer servers are assumed, the system performs adequately, and communication is fairly straightforward. However, if multi-user peer servers are assumed instead, the protocol does not support a way of reliably relaying messages within the application itself, as the message does not have enough metadata to figure out whether the message was bound to a certain machine, and whether that machine has the public key, without a user logged in.

7.3 Architecture

Architectures used were the MVC model (without the view) in the python server, and the MVVM model in the client side JS. This produced an application that had good bindings between the data on client and server side. The use of these architectures also provided good maintainability benefits to the resulting application, by increasing code modularity. The resulting application was suitable for use in P2P communications.

8 Future Improvements

Future improvements involve more extensive unit testing, along with potentially using a new server framework such that RESTful APIs can be developed that utilize the HTTP protocol more extensively. The protocol should propose an API that is bound to data models used by the network, such as profiles, users, and messages. This can potentially reduce the number of APIs to be maintained down to one, as remote and local users can use the same API, but with different restrictions based on the HTTP method used (for example, remote users shouldn't be able to use the DELETE method). Several application usability improvements, such as confirmation of message receipt have yet to be implemented, which can be done in the future.