CS351 - Midpoint Progress Report

Zachary Kaplan - ztk4

April 19, 2019

This midpoint progress report will provide an update for where I am in completing the assignment to build and end-to-end encrypted (E2EE) messaging service. First, I will give an overview of the system architecture I've chosen. Then I will discuss the design choices and progress made so far in each of three sections: Server Authentication, Client Authentication / Key Storage, and E2EE Message Protocol.

Additionally, all existing code for this project is currently live at `https://github.com/ztk4/NJIT/tree/master/cs351/proj`.

1 Architecture

I have decided to go with a NodeJS server, and web browser clients. The server will serve an HTML/CSS/JavaScript client application to each user that navigates to the app's domain (currently just localhost on my laptop). The server will also expose several remote API endpoints over HTTPS that the client-side JavaScript will call into asynchronously (AJAX), using JSON as the communication format.

On the server, I'm using many packages from the Node Package Manager (`npm`) to serve web pages, query the database, etc. The most crucial of which are `express`, `https`, and `babel`. `express` is a web framework, allowing easy construction of a web-server with JavaScript functions sitting at each URL. `https` is a package for creating an HTTPS server, which I am using to create a TLSv1.2 server. The `express` framework listens on the incoming connections of this server. Finally, `babel` is used to compile ES6 compliant JavaScript to ES5 compliant JavaScript since NodeJS currently requires ES5.

On the clients, I'm mainly using native browser JavaScript. For easier GUI work, I have include a recent build of jQuery – a library that massively simplifies access of the DOM. I'm also using the `npm` package `idb` – served from `https://unpkg.com` – which wraps the native browser API for IndexedDB with a promise-based API. I'm using IndexedDB to store client key pairs after a client has closed their browser, since its storage is persistent between browser sessions. Finally, I'm using the native browser WebCrypto API to generate, derive, and wrap keys, as well as for encrypting and decrypting messages on the client. This is of course the most critical component of the project, which is what allows me to user a browser as a client instead of building my own native desktop client from scratch.

2 Server Authentication

Ideally, for a client to authenticate the server they would verify the server's advertised public key by getting a certificate from a trusted third-party Certificate Authority (CA). Unfortunately, since I don't have a public domain for the server, I don't have a good way to get a third-party certificate to sign my server's public key. As such, my plan is to use a self-signed public key. This is not great for the clients, because the certificate verifying the server's public identity is also created by the server itself. This is not an ideal solution, and I'm still thinking about how to improve this. See Fig 4 for information about how I currently generate my server's self-signed key pair.

3 Client Authentication

Since any client has to have a key pair to identify themselves to other clients, we can actually use this key to authenticate any client trying to login. For example, if Alice wants to ask the server if she has any new messages, she need only sign her request for new messages with the private key of her identity key pair. Since the server has her public identity key stored from her initial account creation (see protocol below), the server can just verify the signature on the message to be sure that it was Alice who sent it. Therefore, **passwords never have to be sent over the network, and never have to be stored on the server**. I will, however, encode each client's public key pairs with a PBKDF2 password derived key before storing the keys locally. This would require a password to be typed in by a client on login, despite not sending the password to the server.

I have already implemented a JavaScript class for storing to and reading from IndexedDB. This will be used for key storage and message history storage. A screenshot of some of this code is attached in the appendix as 4

4 Message Protocol

I decided to base my message protocol off of WhatsApp's implementation of the Signal messaging protocol. You can find the WhatsApp whitepaper at
`https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf`, and info about Signal at `https://signal.org`. I won't go into too much detail about my protocol since the first several figures in the Appendix below detail it pretty well. I will mention that the keys referenced as `I`, `S`, and `O` are Identity, Signed (by the Identity key), and One-Time key pairs respectively. Whatsapp uses Curve21159 for these keys, but WebCrypto doesn't offer that so I will probably use a different elliptic curve suitable for Diffie-Hellman key exchange. Additionally, I don't plan to implement every detail of the protocol. For a project of this scope, I intend to omit key rotation of the Signed keys, and possibly some advanced features of the double-ratchet (such as support for out-of-order or delayed messages). Also, I **highly** recommend zooming in on this PDF to read some of the pictures in the Appendix.

Appendix

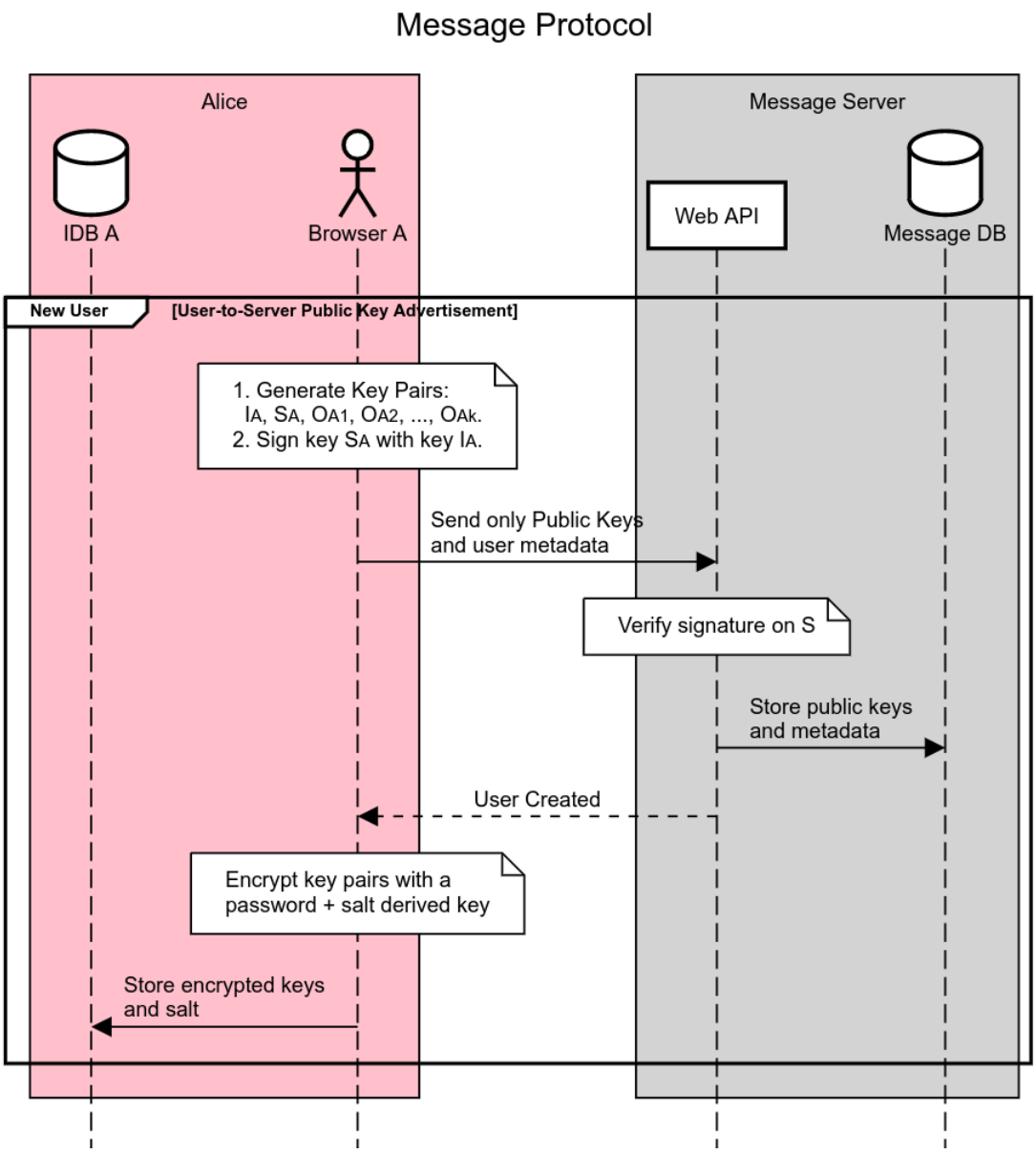

Figure 1: Flow for adding a new User.

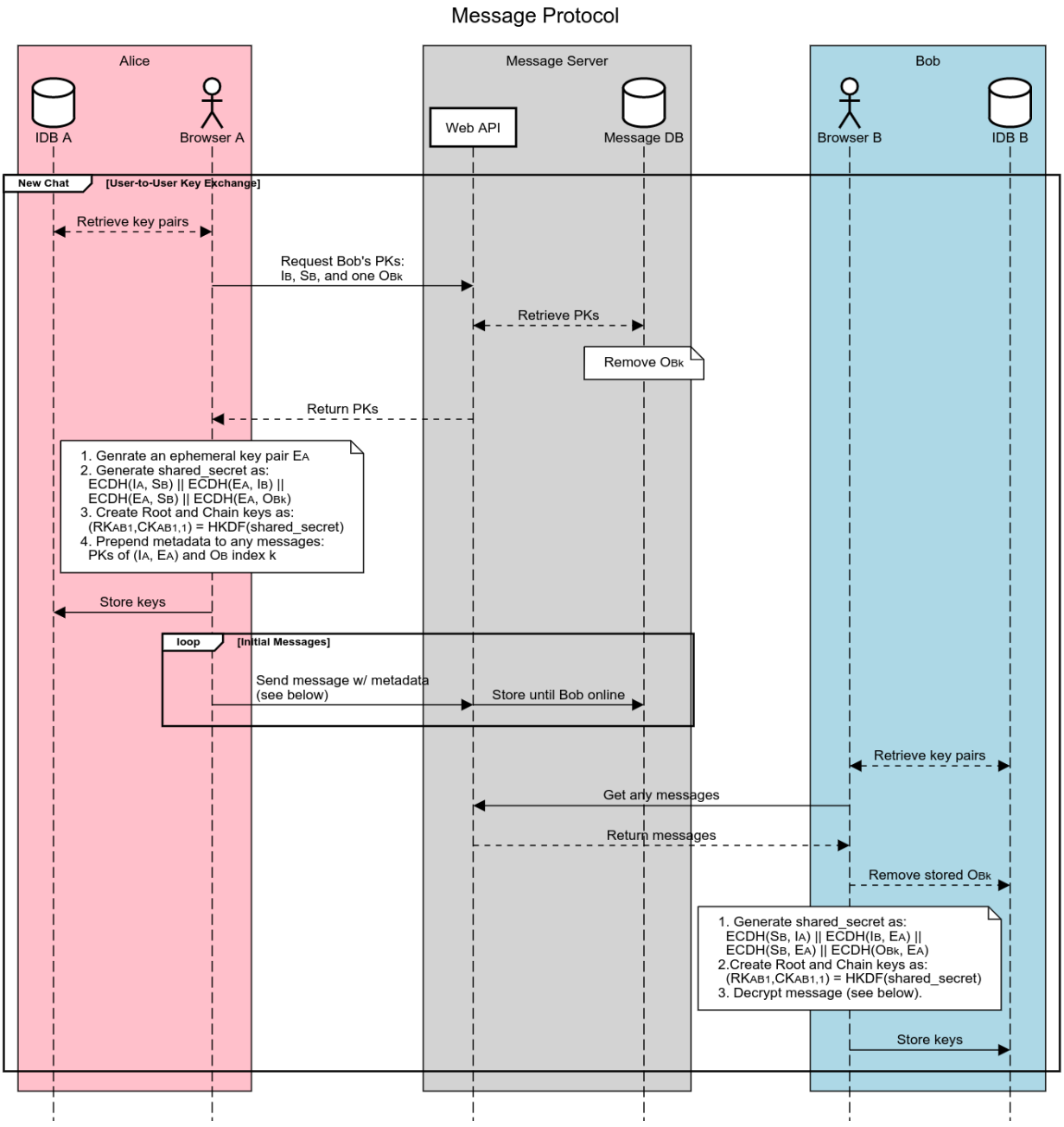

Figure 2: Flow for initiating a chat session with another user.

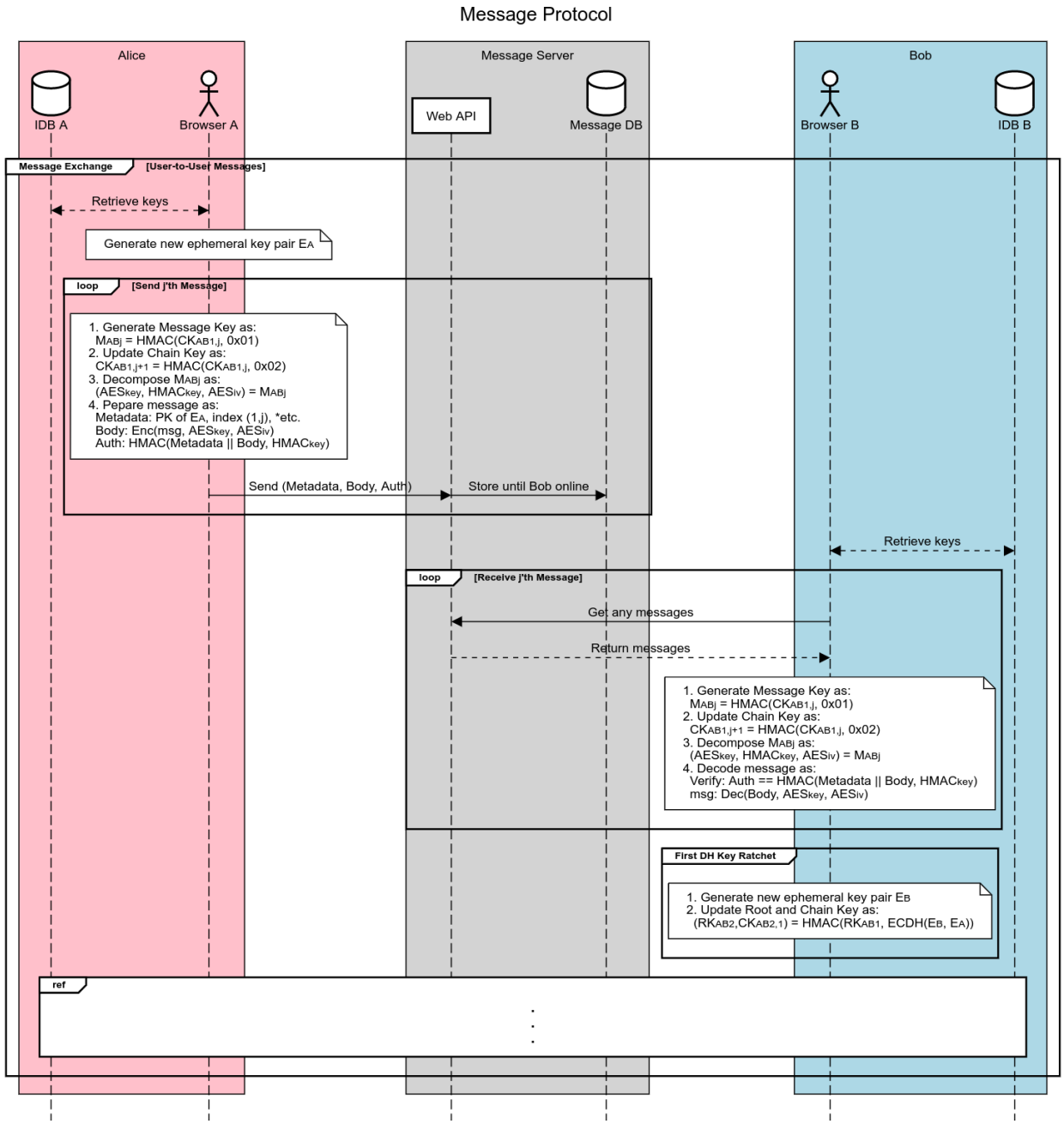

Figure 3: Flow for exchanging messages with another user.

```
# Create a new key pair according to our policy:
# RSA 4096 + SHA-256, no DES, self-signed (localhost), live for 1 year.
openssl req -x509 -newkey rsa:4096 -sha256 \
  -keyout .certs/key.pem -out .certs/cert.pem -days 365 \
  -nodes -subj='/CN=localhost'
```

Figure 4: Command for generating the self-signed key pair and certificate on the server.

Figure 5: Code snippet for IndexedDB storage on the client.