Collabypto: Real-Time Collaborative Text-Editing with Signal Protocol

Aviv Ples, Eran Jordan, Yuval Alaluf

Abstract

Real-time collaborative text-editing software such as Google Docs have become very in-demand in recent years. However, they have a significant drawback: all documents are stored on the (potentially untrusted) server as plaintext and therefore anyone with access to the server (e.g., system administrators, governments, hackers) may read and modify these documents.

Collabypto aims to create a real-time collaborative text-editor with end-to-end encryption by using Operational Transformation and the Signal Protocol. This allows users to create and edit documents in real-time while protecting users against an honest-but-curious servers. The source code of Collabypto is published on GitHub as an open-source project: https://github.com/avivples/collabypto

1 Project Goals

The goal of the project was to create real-time collaborative text editor with end-to-end encryption using the Signal Protocol. This allows users to create and edit documents in real-time while protecting users against an honest-but-curious servers. This is achieved by combining Operational Transformation (OT) with the Signal Protocol.
Our proposed system encrypts each edit operation separately on the client side and sends them to the server that distributes these edits to all other clients editing the same document. Each client then reconstructs and updates the document from these encrypted operations using OT.
Using Signal Protocol also provides users with Forward Secrecy and Future Secrecy due to the Double Ratchet Algorithm and the Diffie-Hellman Key Exchange Protocol (X3DH), respectively.

2 Operational Transformation (OT)

During real-time collaboration, network latency between multiple clients may result in the possibility of version conflicts. This is where Operational Transformation (OT) will be used.
As a motivating example, consider a Client A and a Client B who currently share the state (document) containing the string `ABCD`.
Now, say Client A enters `X` in between `C` and `D`. The operation will be of the form `insert(X,3)` (insert X at position 3). Assume that at the same time, client B deletes the letter `B` (i.e., performs `delete(1,1)` (delete all characters between positions 1 to 1).
What we want to occur is for both Client A and Client B to end up with the string `ACXD`. However, the operations performed will result in the following states:

Client A	**Client B**
Starting State: `ABCD`	Starting State: `ABCD`
`insert(X,3) [local]` $\Longrightarrow$ `ABCXD`	`delete(1,1) [local]` $\Longrightarrow$ `ACD`
`delete(1,1) [remote]` $\Longrightarrow$ `ACXD`	`insert(X,3) [remote]` $\Longrightarrow$ `ACDX`

And the documents are out of sync! To fix this issue, we rely on the OT algorithm.

In short, the OT algorithm transforms edit operations such that they can be applied to documents whose states have diverged, bringing them back to the same desired state. This is done as follows:

- Every edit operation (insert/delete) is represented as an **operation,** which is applied to to current working document and results in a new document state.
- To handle remote operations, OT uses **transform functions** which takes two operations applied on the same document by different clients and transforms the second operation so that the intention of the first operation is preserved.

Back to our motivating example, we if apply OT, we will now see:

Client A	**Client B**
Starting State: `ABCD`	Starting State: `ABCD`
`insert(X,3) [local]` $\Longrightarrow$ `ABCXD`	`delete(1,1) [local]` $\Longrightarrow$ `ACD`
`delete(1,1) [remote]` $\Longrightarrow$ `ACXD`	`// Transform the offset to 3 - 1 = 2`
	`insert(X,2) [remote]` $\Longrightarrow$ `ACXD`

And the documents are in sync!

There are two types of Transformation functions:

1. **Inclusion Transformations ($IT(O_A, O_B)$):** transforms O_A against O_B such that the impact of O_B is included.
2. **Exclusion Transformations ($ET(O_A, O_B)$):** transforms O_A against O_B such that the impact of O_B is excluded.

These functions must satisfy the following two convergence properties:

1. **CP1/TP1:** For two concurrent operations, the final result must be equivalent on all clients.

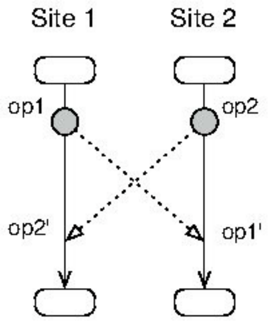


Figure 1: TP1 property

2. **CP2/TP2:** For three concurrent operations, the transformed operations should not be dependent on the order executed. Further, all three should converge to the same document.

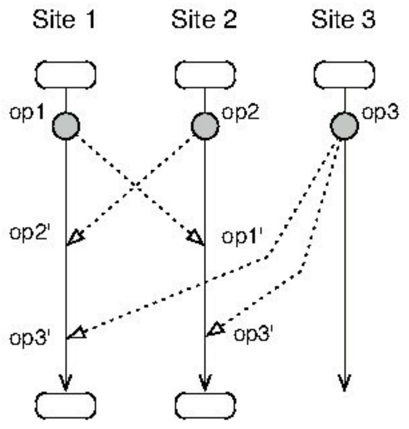


Figure 2: TP2 property

We also have the following three consistency properties:

- **Causality preservation:** ensures that the execution order of dependent operations are in a cause-effect relation. That is, all operations will be received in the order they were made.
- **Convergence:** ensures that all local copies of the shared document are identical.
- **Intention preservation:** ensures that the effect of executing an operation on the document is the same as what the client intended.

2.1 Our Usage of OT

By using Operational Transformation, our system provides each user with their own local copy of the documents, wherein each client operates on his/her own local copy. This allows for extremely fast local edits, even under network latency. In additional, multiple users are able to simultaneously edit any part of the document with no locking at any time (since they work on different local copies).

Changes made by a client are propagated to the rest of the clients, allowing for high responsiveness. When a client receives an update operation from another client, the client first transforms the changes before executing the update on his/her own local copy.

We solve inconsistency by enforcing a total ordering on messages sent to the clients. This is done using the central server which time-stamps each message before sending it to the clients. [1] Using a global ordering will ensure both convergence and causality. However, we need to handle intention. This will be done by transforming independent operations with respect to each other. This is handled by modifying the offset with respect to each other as explained above.

[1]In future works, we wish to expand our assumption of an "honest, but curious server" to a "malicious" server. To do so, we need to remove the use of global time-stamps and replace it with an topological ordering of the operations. That way, each user knows the correct ordering of the operations without relying on the server.

3 Signal Protocol

The Signal Protocol is an end-to-end encryption protocol originally created for instant messaging. It is described by its creators, Whisper Systems, as “a ratcheting forward secrecy protocol that works in synchronous and asynchronous messaging environments.”[2]

As a high-level overview, Signal improves security by using true end-to-end encryption with something called **forward secrecy**, which means the encryption keys used to communicate between users can't be learned by a server, and no single key gives access to past messages. So even if your private keys are stolen, your history will be safe.
Signal Protocol also uses a **Double Ratchet Algorithm** – which on every message, changes the shared key – minimizing the amount of information that can be decrypted if one of the keys were to be compromised by an adversary.
Signal also provides **future secrecy** which means that when an adversary compromises a key, he/she cannot compromise future keys.
Putting these together, if an adversary comprises a message then he/she can only learn that message and no other message past or future.

To go into more details, the Signal Protocol uses the following algorithms and protocols: the Extended Triple Diffie-Hellman (X3DH) Key Agreement Protocol, Double Ratchet Algorithm, Curve25519, AES-256, and HMAC-SHA256, as well as some cryptographic primitives.
The above protocols/algorithms are used in the following steps of the process:

- X3DH is used to create a master secret between two clients.
- Curve25519 generates both public keys out of private keys, and master secret keys out of one client's private key and another's public key.
- AES-256 is used as the encryption algorithm, using the shared master key.
- HMAC-SHA256 is used for verification and authentication of messages.

Assume we have two users Alice and Bob and a central server. We explain the Signal Protocol in more detail below.

Key Generation and Registration:
The first step in creating end-to-end encryption with Signal is creating long-term identity keys, a medium-term signed prekey pair, and ephemeral prekey pairs. These keys are generated by the client and stored locally. Next, these public keys and the client's ID are used to create a *key bundle,* which is registered with the server.
When Alice wants to send messages to Bob, she first registers with the server and requests Bob's key bundle from the server. This is done so that Alice can learn Bob's ID and public keys to start a session with Bob.

Sessions:
Now that Alice has Bob's key bundle, she can create a master shared secret key by using her identity key and medium-term private key with Bob's private keys. This master key is used to create a session with Bob. After the master key is created and validated by Bob, the two can start sending encrypted messages back and forth securely.

[2]libsignal-protocol-java. GitHub repository, commit hash 94fd1e38. 2016. URL: github.com/WhisperSystems/libsignalprotocol-java (visited on 07/2016).

Sending Messages:
When Bob and Alice exchange messages with each other, Alice encrypts her messages using the shared master key generated in the X3DH protocol (explained below) along with Bob's ephemeral keys. Each message that is sent generates a new set of one-time session/ephemeral keys that are used to encrypt/decrypt future messages. These one-time keys are generated using the Double Ratchet Algorithm explained below.

Let's break down the **X3DH Key Agreement Protocol** further:
X3DH starts things off by generating all the necessary keys between the two parties that wish to communicate. In particular, we create a shared secret key used by the parties. X3DH has three main stages:

1. Bob registers his identity key and pre-keys to the server.
2. Alice obtains Bob's prekey bundle from the server and uses it to start a session with Bob. Alice then sends an initial message to Bob.
3. Bob receives and decrypts Alice's message.

Now, Alice and Bob have the shared secret key and can begin sending messages back and forth.

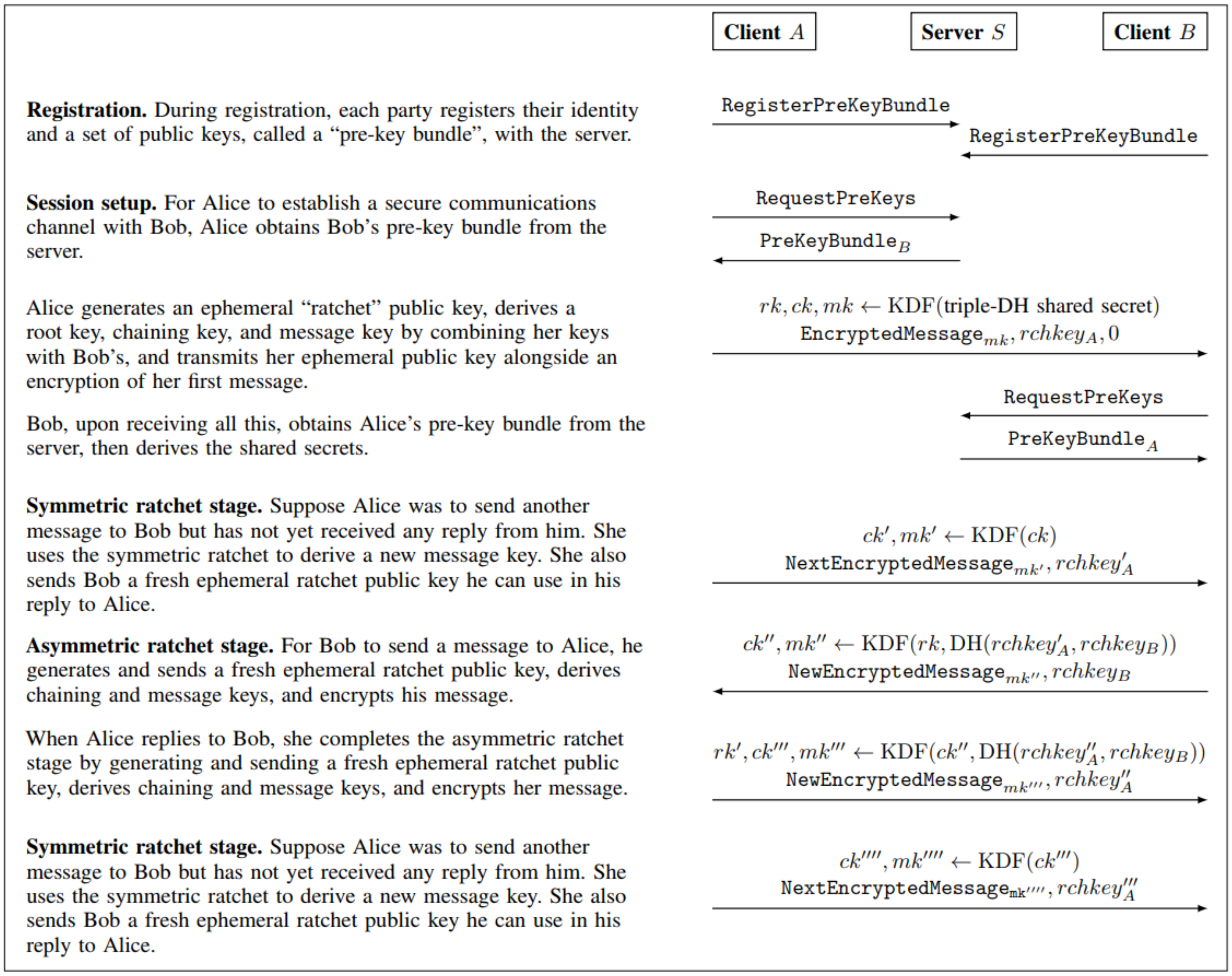
The Double Ratchet algorithm is a bit more involved. After both parties have agreed on a shared secret key via X3DH, they use the Double Ratchet Algorithm to send and receive the encrypted messages.

- X3DH outputs a master secrecy key which is used to derive two symmetric keys called the *root key* and *sending chain key.*
- As messages are sent/received, keys that are used are constantly changed via the *Key Derivation Chain (KDF).*
- When Alice encrypts her messages, she advances her sending chain one step, deriving the next sending chain key along with a new message encryption key.
- When Alice receives a message from Bob, she advances her receiving chain one step to derive the next decryption key.
- When a session is first created, the root change is advanced, generated a *ratchet key (ephemeral key).*
  - This is attached to messages that Alice sends so that each message is attached a new ephemeral key, making it impossible for adversaries to decrypt past or future messages.

The protocol can be divided to four stages: registration, session setup, symmetric ratcheting, and asymmetric ratcheting.

- The first two stages are done once: registration is done once per client, and session setup is done once per session, of which there can be an arbitrary amount.
- The clients switch between the last two stages periodically.

The process is summed up in the following picture:



Registration. During registration, each party registers their identity and a set of public keys, called a "pre-key bundle", with the server.

Session setup. For Alice to establish a secure communications channel with Bob, Alice obtains Bob's pre-key bundle from the server.

Alice generates an ephemeral "ratchet" public key, derives a root key, chaining key, and message key by combining her keys with Bob's, and transmits her ephemeral public key alongside an encryption of her first message.

Bob, upon receiving all this, obtains Alice's pre-key bundle from the server, then derives the shared secrets.

Symmetric ratchet stage. Suppose Alice was to send another message to Bob but has not yet received any reply from him. She uses the symmetric ratchet to derive a new message key. She also sends Bob a fresh ephemeral ratchet public key he can use in his reply to Alice.

Asymmetric ratchet stage. For Bob to send a message to Alice, he generates and sends a fresh ephemeral ratchet public key, derives chaining and message keys, and encrypts his message.

When Alice replies to Bob, she completes the asymmetric ratchet stage by generating and sending a fresh ephemeral ratchet public key, derives chaining and message keys, and encrypts her message.

Symmetric ratchet stage. Suppose Alice was to send another message to Bob but has not yet received any reply from him. She uses the symmetric ratchet to derive a new message key. She also sends Bob a fresh ephemeral ratchet public key he can use in his reply to Alice.

Client A
Server S
Client B
RegisterPreKeyBundle
RegisterPreKeyBundle
RequestPreKeys
$\texttt{PreKeyBundle}_B$
$rk, ck, mk \leftarrow \text{KDF(triple-DH shared secret)}$
$\texttt{EncryptedMessage}_{mk}, rchkey_A, 0$
RequestPreKeys
$\texttt{PreKeyBundle}_A$
$ck', mk' \leftarrow \text{KDF}(ck)$
$\texttt{NextEncryptedMessage}_{mk'}, rchkey'_A$
$ck'', mk'' \leftarrow \text{KDF}(rk, \text{DH}(rchkey'_A, rchkey_B))$
$\texttt{NewEncryptedMessage}_{mk''}, rchkey_B$
$rk', ck''', mk''' \leftarrow \text{KDF}(ck'', \text{DH}(rchkey''_A, rchkey_B))$
$\texttt{NewEncryptedMessage}_{mk'''}, rchkey''_A$
$ck'''', mk'''' \leftarrow \text{KDF}(ck''')$
$\texttt{NextEncryptedMessage}_{\text{mk}''''}, rchkey'''_A$

Figure 3: Signal Protocol Process, https://eprint.iacr.org/2016/1013.pdf

3.1 Our Usage of the Signal Protocol

In our system, we use the Signal Protocol, implemented using Whisper System's Java Library, to hide clients' messages from the server. By implementing the Signal Protocol, the server becomes unable to gain any information from the messages it receives, as they are all encrypted pairwise according to some session between two clients.

The client registration is done upon connecting to the server with a valid authentication token and sessions are built between each two clients in the document upon creation of a new document. When a client enters the document, he/she exchanges key information with other clients and starts sessions between each of the clients. From this point on, clients can send messages (insert/delete edit operations) encrypted using their session ciphers to one another without the server learning anything about the contents of the edits.

The main difference between our usage of the protocol and the standard usage is how the information is sent. In its basic form, the Signal Protocol is created between each two people individually (for one-on-one conversations, e.g., in WhatsApp). However, our main form of communication in this application is group messaging. Therefore, instead of sending an encrypted message to one person, a client encrypts as many messages as there are users in the documents (pairwise encrypted for each one) and relays them to the server. The server then relays each message to its correct recipient.

4 System Architecture

We will use OT to maintain concurrency between users participating the same session. To allow for more robustness, we will adopt a replicated architecture for the storage of shared documents. In particular, each user will locally store a copy of the current working document. When remote editing operations arrive at the local site, the user applies OT operations to update his current local version of the document. This replicated architecture provides the added benefit of parallel processing and reduced latencies when multiple users collaborate on a single document at the same time.
In order to avoid a total of $O(n^2)$ pairwise connections between clients, we will have a central server, which acts as a relay by forwarding operations from each user to all the other users in the network. Using a central server will allow for only $O(n)$ connections, one connection for each client.

4.1 The Honest-But-Curious Server

The server can be viewed as a central entity whose main responsibility is to receive and forward messages from and to clients.
The server allocates one thread for each connection and keeps track of the following for each client:

- The socket for the connection.
- The input and output streams.
- The public information of the client (which documents the user has access to, public keys, etc.).
- The document the client is editing.
- The encrypted history of the document the user was editing (so the user is able to retrieve the correct document when he/she returns).

Each time a new client connects, the server sends the sessions and document history.
When a client sends an edit operation, the server is able to read relevant information about who sent the message and to whom to forward the message to. However, the actual contents of the message are unknown to the server (as they are encrypted using the Signal Protocol).

4.1.1 Identifying Clients

To avoid excessive latency, we limit the maximum number of concurrent users for a single document. We will keep a list of the sockets between the server and each client. We also have a list of usernames and documents for which the user has access to in order to update the display showing all connected clients.
When the client disconnects, we flag the client as inactive in the list of sockets so that operations are not sent to that client. Further, the server appropriately closes all sockets and streams to that client and updates the list of active users.
Note: If the server disconnects, this will disconnect all clients as all socket connections will have been broken.

4.2 The Client

The client represents a single "user" who will be editing a collaborative document using his GUI. To communicate with the server, the client will open a TCP connection with the server.

Abstractly, the clients each keep their own local copy of the document that is being edited. Changes to the local copy creates an *Operation* object which is sent to the server and the other session clients. Each client will then use the OT algorithm to update his/her own local copy. This will ensure that all clients viewing the same document converge to the same state.

4.3 Work Flow

4.3.1 Registration

Each client will first have to register with the server in order to create/join documents.
The process is as follows:

1. The server receives the client's username. If the server does not recognize the name, he requests a token.
2. If the token is invalid, the server rejects the user. Else, it continues to registration.
3. The client creates the key pairs needed for the Signal Protocol: Identity key, Signed pre-key, and many unsigned pre-keys (the standard is 100). Then, the client sends the server the PreKeyBundles, which contain the public keys as well as one unsigned public prekey.
4. The server saves the client's public information in its database.

4.3.2 Document Creation

When a client wants to create a document, the following process occurs:

1. The client chooses to edit a document. He selects which (registered) users he wants to invite.
2. The client sends the server the document name and the list of clients in the document.
3. For each client, the server creates a list of sessions out of every other client's public information.
4. When a client first enters the document, the server sends them the list of sessions, and the client builds their sessions with the other clients.
5. The client can now edit the document securely.

4.3.3 Document Updating

Each client will have a queue in which all changes that need to be made are kept. Any change made by a client is added to the queue according to the time-stamp the request is sent. The client will go through the queue to make these changes in order.
When a change is created, this change is sent to the server and then added to the queue of all other clients. Therefore, an edit made by one client will become edits in all other clients.

Concurrent edits on the same document will be handled as follows:

1. The client relays to the server what document he wishes to edit.
2. The server sends the client the history, which is an (encrypted) list of operations that happened since he last joined. The client uses these operations to become up to date with the current state of the document.
3. When the server receives an update operation from a client, it checks which document is being updated.

4. The server then sends the update operation to all clients in the document. The clients will decrypt the message and receive the operation. They will then use the OT algorithm to update their own local copy. If they are not online, the operation is added to their history.

Since each client only accesses his/her own local copy of the operations buffer, multiple threads will not be accessing the same buffer at the same time. All changes will be done by the OT algorithm in the client's copy itself, which is sequential.

4.4 Message Passing

The main type of message sent between clients and the server are *Operation*-type (encrypted) messages which specify the relevant updates to the document, along with the information on the document being updated and which client to send this update to. These messages are either *insert* or *delete* operations generated by the client, which specify the order, position, and character(s) to insert/delete from the document.

Each *operation* is associated with a *context vector*, which corresponds to a document state on which the operation is defined. The use of these context vectors ensures that the effect of the operation is correctly interpreted by the OT algorithm to allow for correct updates and convergence.

In addition, the server may send other messages to the clients such as an updated list of users in the session, documents for which the user has access to, and other information passing between the server and client.

5 Security Assumptions

Our main security assumption is that the server is honest-but-curious. He is allowed to look at the messages being sent, but he will send the messages *unedited* to the correct client(s).
If this assumption is broken, and the server is malicious, a few problems may arise:

1. The server changes the contents of messages:

   In this case, the server cannot transform the message to any other meaningful message, as the contents are encrypted and he has no way of knowing the keys.

   The message he will send will be random data that does not correspond to a valid operation (except with negligible probability.) This will cause an error for the client this is being sent to.

2. The server sends the message to someone else:

   In this case, the client receiving the message will not be able to decrypt it, as he does not have the correct session cipher for this message. This will cause an error for the client.

3. The server does not send the message:

   In this case, when the server does send a message to the client he withheld a message from, he will be unable to decrypt it due to his ratcheting not corresponding with the received message. This will cause an error.

   If he gets a message from a different client after a message was withheld from him, he will be able to decrypt it, but the context vector of the operation will be wrong. This will also cause an error.

4. The server creates its own sessions with the clients:

   In this case, the server creates its own keys needed for the Signal Protocol. When a client intends to create sessions with other clients, the server can build a session between the clients and itself as well.

The server would therefore be able to decrypt messages sent by the clients, and act as a client itself. (i.e., sending his own messages)

However, he would not be able to change the messages sent by other clients. This is because sessions are created pairwise between each two clients, so he does not have the information to meaningfully change the contents of these messages.

5. The server changes the order of the messages:

   The server is the one that decides the order of messages. If the server was not honest about the order, he could give the messages in a different order, even between clients.

   This will cause problems with the OT algorithm, as the context vector will be wrong and the client would not be able to make sense of the operation. This will cause an error.

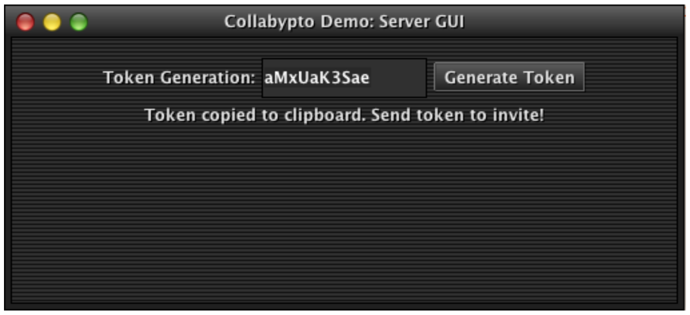
In summary, a malicious server could cause the program to stop working, and even see the contents of the document and add to it. However, he would still not be able to change messages that other clients send to each other.

Another assumption we made is about the unsigned prekeys. Currently (for demo purposes), we assume users will never run out of prekeys. If a user is in enough sessions that they run out of prekeys, being invited to new documents will cause an error for the inviter.
This can be fixed in the short term by increasing the amount of prekeys generated, and in the long term by clients generating more prekeys and updating the server when their supply is low.

6 Features/Tutorial

6.1 Server

1. **Token Generation:** The server is responsible for generating authentication tokens that can be sent to users who wish to register with the server. A token is a random 10-character string.



aMxUaK3Sae

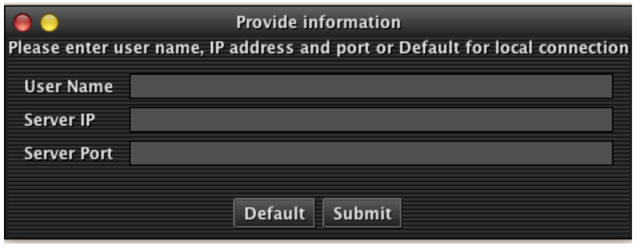
Token copied to clipboard. Send token to invite!

6.2 Client

1. **Login Page:** At the login page, the user (whether new or returning) will be prompted to enter a username. Then, the client may input a specific server IP and port number and click "Submit" to connect to a specific running server or may click "Default" to connect to IP 127.0.0.1 at port 4444.

   If the entered username is already taken, the user will be prompted to enter a new username. [3]

   If a user has information saved about this server, but the server has no information about the user, the user will receive a "Missing information in server" error and the program will close. [4]



Provide information

Please enter user name, IP address and port or Default for local connection

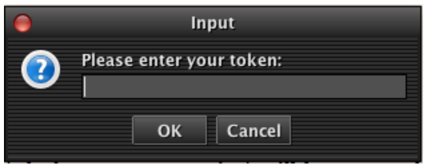
User Name

Server IP

Server Port

2. **Enter Token:** Now, the user will be prompted to enter the invitation token he received from the server (or another client). If the user enters an invalid token or an already used token, he will get an error message "Invalid Token" and the program will exit.

   Note: A returning user will *not* be prompted to enter a token.



Input

Please enter your token:

3. **Document Selection Page:** Here, the user will have several options:

   (a) If the user has been invited to collaborate on a specific document, this document will appear in the left panel. Clicking over the document name and clicking "Select Document" will take the user to the "Client GUI" page with that document.

   (b) The user may press "Refresh List" to refresh the documents listed in the left panel (in the case where he was invited to edit a document after reaching this page).

   (c) In the right panel, the user may create a new document by entering the document name and clicking "New Document". This will prompt the user to a page where he/she can invite other users to collaborate on the new document.

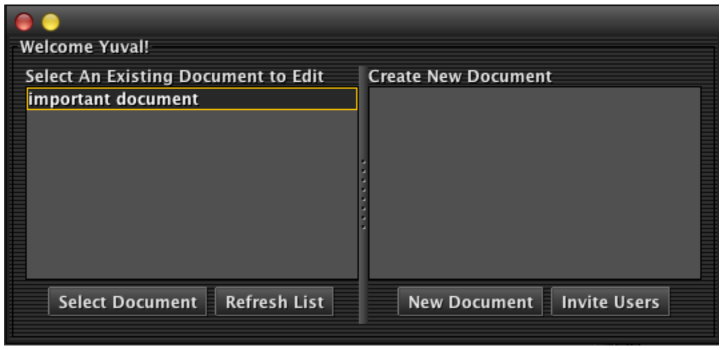
      i. See Invite Users To Edit below.

   (d) If the user wishes to invite users to collaborate on a new document, he should first invite these users to join the server by clicking the "Invite Users" button.

      i. See Invite Users To Server below.

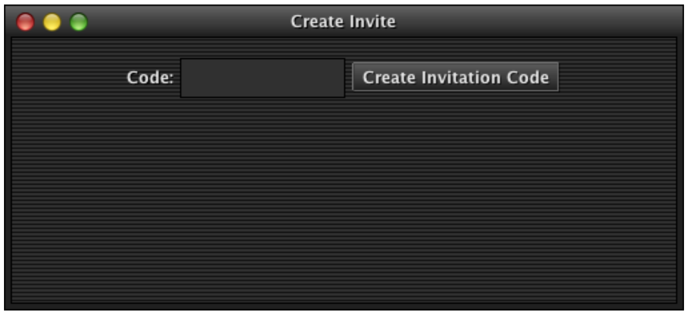
[3]Currently, the program will exit if an already in use username is entered and the user will need to rerun the program. In a future release, we will return the user to the login page rather than exit the program.

[4]In a future release, we will return the user to the login page rather than exit the program.



4. **Invite Users to Server:** Here, the user can generate invites (tokens) to users who he/she wants to invite to register with the server. This page works exactly like the "Token Generation" page of the server with more user-friendly terminology (e.g., create invite vs. create token).

   The user should create and send a token to each of the individuals he wishes to invite to the server. Once finished generating invites, the user can close the window and return to the "Document Selection Page".

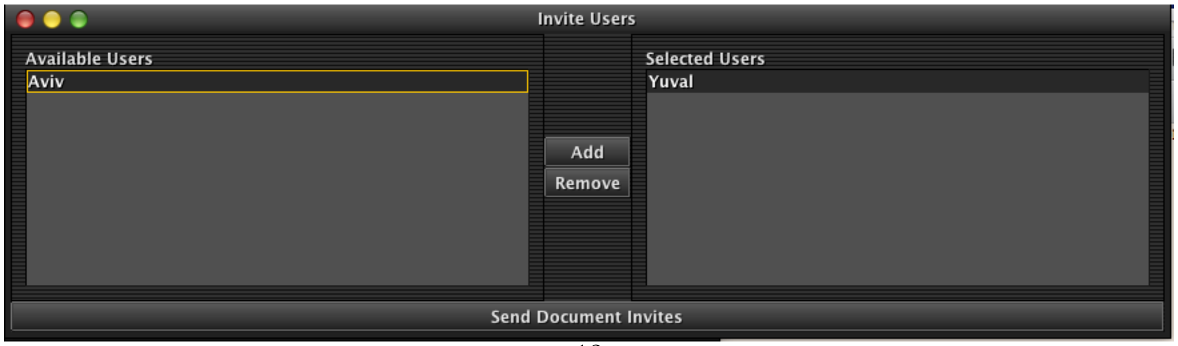


5. **Invite Users To Edit:** After entering the name of a new document, the user will be prompted to this page where he/she can invite specific users to collaborate on the document. On the left side, the user will see a list of all currently registered users and on the right side the user will see a list of users who have currently been selected to invite. The user may freely add/remove users from the "Selected User" list.

   After the users have been selected, pressing "Send Document Invites" will give the selected users access to the document and will take the user to the Client GUI page.

   If the user does not wish to invite any users, he/she may press "Send Document Invites" directly without selecting any users.

   Note that a user will only be able to invite already-registered users. Therefore, before document creation all parties should be registered with their usernames.

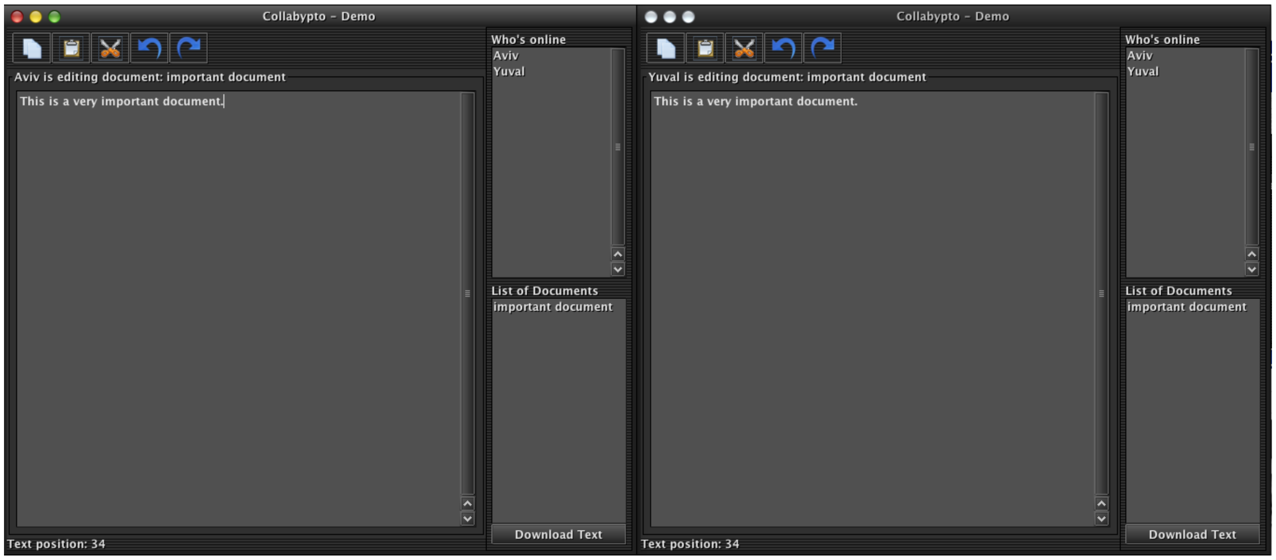


6. **Client GUI:** At this page, the collaborative editing can begin! In the main panel the user will see the main text of the document and on the right side he/she will find the users who are currently online and the list of documents to which the user has access to.

   The user can use the icons in the top to copy, paste, cut, redo, and undo actions (although keyboard shortcuts are also available).

   At the bottom right of the window, the user can download a .txt file with the current working document.

   Note: Currently, the user is unable to return to the Document Selection Page (e.g., to switch between documents). To do so, the user must exit the program and log-in again. [5]



7 Current Issues and Future Work

The main issue that needs to be fixed in the future is synchronization:
Currently, when a client joins or leaves a document, another client writing at that time will cause issues. In particular, when joining, the clients will crash and the document will go out of sync. This is caused because when the client joins, he still needs to update the document state from the history buffer so at this time, no new edits should be made. And when leaving, although the clients will not crash, the client that left will be out of sync. This is caused by operations during joining/leaving taking some time. [6] In particular, when writing the current state of the document to a file, the client does not read new operations that other clients have just made. As the client is still active, these messages will not be saved to history. Therefore when he returns, those messages will not appear for him and he will be out of sync.
These issues may be fixed by a consensus algorithm. For example, only when all clients agree to stop, a new user can start updating from history/writing to file, then when that client is done, the other clients can agree to start writing again. In the meantime (which will be relatively short and seamless for users) users will not be able to change the document.

Another issue is with how documents are stored in the server: document names are stored as the key to a hashmap containing various information about the document. This means that multiple documents cannot have the same name. Clearly, different clients need to be able to make documents with identical names.

[5] This feature will of course be added to the next release to improve the user experience.
[6] Joining: updating from history; Leaving: writing information to files

This can be fixed by appending some unique information to the document name in the server (such as client name or client ID) and hiding it from the users. Then the users can see the name they chose, but the server will differentiate between different documents with the same name.