Data Structures II
CS/CE 201/372

Final Project Report

Collaborative Text Editor

Instructor: Maria Samad

Taha Zahid - tz09220
Fowzi Ali - fa09193
Ayaan Merchant - am08853
Abdullah Shaikh - as09245

9th May 2025

Contents

Main Functionalities

0.1 Implementation of Data Structure

This section describes the implementation details of the core data structure used in the project: the Replicated Growable Array (RGA). The RGA model is a Conflict-free Replicated Data Type (CRDT) used for collaborative text editing. The following are the key functions of the data structure used in the project:

Struct: RGA_Node

`RGA_Node(std::string id, string value, bool is_deleted, std::string prev_id, std::map<char, int> version_vector)`

This struct represents a single character (or element) in the document. It stores the unique ID of the character, the character value itself, a tombstone flag `is_deleted` for deletion, the ID of the previous character, and the version vector.

Simple Functions

- `bool is_empty()`
  Returns true if the document has no characters; otherwise, false.
- `int getNodeIndex(const RGA_Node& r1)`
  Returns the index of a node `r1` using its ID from the `id_to_index` map.
- `std::vector<RGA_Node> getNodes()`
  Returns the list of all nodes currently in the RGA.
- `std::map<std::string, size_t> get_id_to_index()`
  Returns the mapping from node IDs to their indices in the document.
- `std::map<char,int> getVersionVector()`
  Returns the current version vector of the local document.
- `std::map<char,int> getNodeVersionVector(const std::string id)`
  Returns the version vector for the node with the given ID.
- `bool is_concurrent(const RGA_Node& a, const RGA_Node& b)`
  Returns true if nodes `a` and `b` are concurrent, i.e., neither dominates the other.
- `RGA_Node* searchNode(const std::string& id)`
  Returns a pointer to the node with the given ID.
- `std::optional<std::string> search(const std::string& id) const`
  Returns the character value of the node with the given ID, if it exists and is not deleted.
- `std::string print_document()`
  Returns the current text of the document, omitting deleted characters.
- `void initializeFromContent(const std::string& content, char clientId)`
  Initializes the RGA with the provided string `content`, assigning characters unique IDs based on the `clientId`.

- `std::string serializeState() const`
  Converts the document state (including version vector and all nodes) into a serialized string.
- `void deserializeState(const std::string& state)`
  Reconstructs the RGA's state from a serialized string representation.

Detailed Function Explanations

`bool isDominate(const std::map<char, int>& a, const std::map<char, int>& b)` This function checks if version vector `a` dominates version vector `b`. Domination means that for every client, `a`'s version is greater than or equal to `b`'s version, and for at least one client it is strictly greater. It returns true if `a` dominates `b`, otherwise false.

`void insert(const std::string& id, const std::string& value, const std::map<char, int>& version_vector_pass, const std::string& prev_id = "")` This function inserts a new character into the document. It creates a new node with a unique ID, the character value, its version vector, and the ID of the character that should come before it.

If the previous ID (`prev_id`) exists, the function finds the appropriate index for insertion by checking for concurrent inserts or dominance with other nodes that have the same previous ID.

Once the correct index is determined, the node is inserted, and the `id_to_index` map is updated. The indices of subsequent nodes are adjusted accordingly. This function handles complex cases like concurrent insertions by recursively calling itself after adjusting the `prev_id`.

`void remove(const std::string& id)` Marks a character as deleted by setting its `is_deleted` flag. If other nodes pointed to this node using `prev_id`, they are updated to point to the nearest preceding undeleted node. The function also updates the version vector to reflect the delete operation and adjusts the indices of the subsequent nodes.

`void merge(RGA_Node& other_node)` Merges a node from another client into the local document. First, it checks if the node with `other_node.id` already exists. If not, it resolves conflicts with existing nodes that share the same `prev_id`. If a conflict is found (e.g., concurrent insertions), it uses version vector comparison to decide the correct insertion point. Then, it calls `insert()` to add the node properly into the structure.

0.2 WebSocket Implementation

We implemented WebSocket communication to enable real-time collaboration between multiple users. Our solution uses a Python-based WebSocket server and Qt's WebSocket client implementation.

0.2.1 Architecture Overview

- **Protocol**: `ws://` for local development with prepared `wss://` support
- **Server**: Python 3.7+ using `websockets` library (supports up to 26 concurrent clients)
- **Client**: Qt's `QWebSocket` implementation with JSON message formatting

0.2.2 Key Components

Server Implementation The Python server handles client connections and message routing. Here is a small excerpt:

```python
import asyncio
import websockets
import string

connected_clients = set()
message_history = []

assigned_ids = {}
available_ids = list(string.ascii_uppercase)  # ['A', 'B', ..., 'Z']
```

```

async def echo(websocket):
    if available_ids:
        client_id = available_ids.pop(0)
    else:
        client_id = '?'
    assigned_ids[websocket] = client_id
    await websocket.send(f"[Server] You are Client ID: {client_id}")

    async for message in websocket:
        message_history.append(message)
        for client in connected_clients:
            if client != websocket:
                await client.send(message)
```

Listing 1: Server Connection Handling

Client Implementation The Qt client manages WebSocket connections and message processing. Here is a small excerpt showing how we are sending and receiving messages:

```
void MainWindow::sendTextMessage() {
    for(QJsonObject &obj : allOperations) {
        if (!obj.contains("cursor_pos")) {
            obj["cursor_pos"] = textEdit->textCursor().position();
        }
        if (isConnected) {
            webSocket.sendTextMessage(QJsonDocument(obj).toJson());
        }
    }
    if(isConnected) {
        allOperations.clear();
    }
}

void MainWindow::onMessageReceived(QString message) {
    QJsonObject obj = doc.object();
    QString type = obj["type"].toString();

    isRemoteChange = true;
    if (type == "insert") {
        string id = obj["id"].toString().toStdString();
        if (r1.searchNode(id) == nullptr) {
            string value = obj["value"].toString().toStdString();
            string prev_id = obj["prev_id"].toString().toStdString();
            QJsonObject vvJson = obj["version"].toObject();
            map<char, int> node_version_vector;
            for (const auto& key : vvJson.keys()) {
                char client = key[0].toLatin1();
                int seq = vvJson[key].toInt();
                node_version_vector[client] = seq;
            }

            RGA_Node newNode(id, value, node_version_vector, prev_id);
            r1.merge(newNode);
            charAdded += 1;
        }
    }
    else if (type == "delete") {
        string id = obj["id"].toString().toStdString();
        if(r1.searchNode(id) != nullptr && !(r1.searchNode(id)->is_deleted)){
            string id = obj["id"].toString().toStdString();
            r1.remove(id);
        }
    }
    // ... (remaining logic)
}
```

Listing 2: Client Message Handling (C++/Qt)

0.2.3 Features & Optimizations

- **Message Debouncing**: Waits 3 seconds after user's last input before sending any messages. Batches continuous typing together to send over WebSocket.
- **State Synchronization**: Full message history replay on reconnect.
- **Connection Monitoring**: 5-second ping/pong heartbeat to detect that connection holds.

0.2.4 Limitations & Future Work

- Current limitations:
  - Limited to 26 concurrent clients
  - Can't handle psuedo-instructions (bold, italic etc) or images
  - Client IDs are not recycled properly on disconnect.
- Future Work:
  - TLS support for production deployments
  - Handling of same id assignment on reconnect
  - WebSocket compression of messages
  - Handling psuedo-instructions, images.

0.3 The Screen

We used the `QTextEdit` widget on top of a `QMainWindow` as the main part of the screen along with an intuitive toolbar providing some basic text editing features.

0.3.1 Key Components

- **Text Editing**: `QTextEdit` for rich text editing.
- **Toolbar**: Basic text formatting options (bold, italic, underline).
- **Status Indication**: Displays connection status.
- **Text Cursor**: Tracks cursor position and selection.

Screen Setup This part of the code initializes the basic parts of the screen, including the toolbar and the text editing area. The stylesheet is also included to to make the screen look more appealing. Lastly the `createIconButton` function is a universal function used to create all the buttons you see on screen.

```
MainWindow::MainWindow(QWidget *parent)
: QMainWindow(parent), currentFile(""), clientId('?'), LastKnownText(""), charAdded(0)
{
    QWidget *central = new QWidget(this);
    QVBoxLayout *mainLayout = new QVBoxLayout;
    this->setFixedSize(800, 600);

    QToolBar *toolbar = new QToolBar(this);
    toolbar->setMovable(false);
    toolbar->setFixedSize(780,50);

    toolbar->setStyleSheet(
        "QToolBar {"
        "   background: light gray;"
        "   spacing: 10px;"
        "   padding: 5px;"
        "   border: 2px solid #4a90e2;"
        "   border-radius: 5px;"
        "}"
        "QToolButton {"
        "   padding: 4px;"
```

```cpp
        "   icon-size: 24px;"
        "   color: white"
        "   background: transparent"
        "}"
        );

    auto createIconButton = [&](const QString &iconPath, void (MainWindow::*slot)(), bool checkable = false) {
        QToolButton *btn = new QToolButton;
        btn->setIcon(QIcon(iconPath));
        btn->setIconSize(QSize(24, 24));
        btn->setCheckable(checkable);
        connect(btn, &QToolButton::clicked, this, slot);
        toolbar->addWidget(btn);
        return btn;
    };
    createIconButton(":/icons/icons/rotate-ccw.svg", &MainWindow::onUndo);
    createIconButton(":/icons/icons/rotate-cw.svg", &MainWindow::onRedo);
    createIconButton(":/icons/icons/printer.svg", &MainWindow::onPrint);
    btnBold = createIconButton(":/icons/icons/bold.svg", &MainWindow::onBold, true);
    btnItalic = createIconButton(":/icons/icons/italic.svg", &MainWindow::onItalic, true);
    btnUnderline = createIconButton(":/icons/icons/underline.svg", &MainWindow::onUnderline, true);
    btnAlignLeft = createIconButton(":/icons/icons/align-left.svg", &MainWindow::onAlignLeft, true);
    btnAlignCenter = createIconButton(":/icons/icons/align-center.svg", &MainWindow::onAlignCenter, true);
    btnAlignRight = createIconButton(":/icons/icons/align-right.svg", &MainWindow::onAlignRight, true);
}
```

Listing 3: Screen Setup (C++/Qt)

ToolBar and TextEdit After the buttons and toolbar is created, we add the buttons to the toolbar and set the layout of the main window. The align buttons are grouped together using a `QButtonGroup` to ensure only one can be selected at a time. The text editing area is initialized as a `QTextEdit` widget, which is then added to the main layout.

```cpp
QButtonGroup *alignGroup = new QButtonGroup(this);
alignGroup->setExclusive(true);
alignGroup->addButton(btnAlignLeft);
alignGroup->addButton(btnAlignCenter);
alignGroup->addButton(btnAlignRight);

textEdit = new QTextEdit;

mainLayout->addWidget(toolbar);
mainLayout->addWidget(textEdit);

QHBoxLayout *buttonLayout = new QHBoxLayout;
QPushButton *btnConnect = new QPushButton("Connect");
QPushButton *btnSave = new QPushButton("Save");
connect(btnConnect, &QPushButton::clicked, this, &MainWindow::onConnect);
connect(btnSave, &QPushButton::clicked, this, &MainWindow::onSave);
buttonLayout->addStretch();
buttonLayout->addWidget(btnConnect);
buttonLayout->addWidget(btnSave);

mainLayout->addLayout(buttonLayout);
central->setLayout(mainLayout);
setCentralWidget(central);
```

Listing 4: ToolBar and TextEdit (C++/Qt)

Status and Cursor Control The status is displayed using a `QStatusBar` widget, which is updated based on the connection status and is connected to the websocket functions. The cursor is controlled using the `QTextCursor` class, which allows us to track the current position and selection of the text cursor the functionality of which is integrated into our websocket and CRDT implementation.

0.3.2 Limitations

- **Basic text formatting**: Currently the screen is limited to only bold, italic and underline functions and align features. The changes made through these is entirely local and do not reflect on the other users' devices as we were unable to implement psudo-instructions in the CRDT.
- **Additional features**: Without the implementation of psudo-instructions in the CRDT we also couldn't add additional features like font size, color, and other text formatting options.
- **Better UI**: The current screen is purely made through code and does not use any QML or other UI libraries which didn't leave us with much room for more advanced customizations.

Complete Workflow

This section details the complete workflow of the collaborative text editor, covering both client-side and server-side operations.

0.4 Client-Side Core Functions

The client implementation revolves around several key functions that handle text editing, synchronization, and communication with the server.

0.4.1 Message Handling

- **onMessageReceived**: This is the central function for processing incoming WebSocket messages. It handles:
  - Client ID assignment from the server
  - Insert/delete operations from other clients
  - File operations (loading/saving documents)
  - CRDT state synchronization
- **Key Operations**:

```cpp
void MainWindow::onMessageReceived(QString message) {
    // Handle client ID assignment
    if (message.startsWith("[Server] You are Client ID: ")) {
        QString idStr = message.section(':', -1).trimmed();
        clientId = idStr[0].toLatin1();
        return;
    }

    // Process CRDT operations
    QJsonObject obj = QJsonDocument::fromJson(message.toUtf8()).object();
    QString type = obj["type"].toString();

    if (type == "insert") {
        // Merge insert operation into local CRDT
        r1.merge(newNode);
    }
    else if (type == "delete") {
        // Apply delete operation to local CRDT
        r1.remove(id);
    }

    // Update UI while preventing recursive textChanged signals
    isRemoteChange = true;
    textEdit->setPlainText(QString::fromStdString(r1.print_document()));
    isRemoteChange = false;
}
```

0.4.2 Text Editing

- **onTextChanged**: Triggered on every text modification, this function:
  - Detects whether the change was an insertion or deletion

  - Generates corresponding CRDT operations
  - Batches operations for network transmission
  - Maintains local CRDT state

- **Key Operations**:

```
void MainWindow::onTextChanged() {
    if (isRemoteChange) return; // Ignore changes from network

    QString currentText = textEdit->toPlainText();
    int cursorPos = textEdit->textCursor().position();

    // Handle insertions
    if (currentText.length() > LastKnownText.length()) {
        string id = clientId + to_string(charAdded++);
        string value = currentText[cursorPos-1].toStdString();
        r1.insert(id, value, versionVector, prev_id);

        // Queue operation for network transmission
        QJsonObject op;
        op["type"] = "insert";
        op["id"] = QString::fromStdString(id);
        allOperations.push_back(op);
    }
    // Handle deletions
    else if (currentText.length() < LastKnownText.length()) {
        r1.remove(node.id);

        // Queue operation for network transmission
        QJsonObject op;
        op["type"] = "delete";
        allOperations.push_back(op);
    }

    LastKnownText = currentText;
    debounceTimer.start(3000); // Batch operations for 3 seconds
}
```

0.4.3 Connection Management

- **onConnected/onDisconnected**: Handle WebSocket connection state changes
- **checkDisconnect**: Implements ping/pong heartbeat (5 second interval)
- **onConnect**: Manual reconnection logic with clean state reset

0.4.4 Client ID Assignment

The server assigns each client a unique single-character ID (A-Z) during connection:

- IDs are recycled when clients disconnect
- The ID becomes part of each operation's unique identifier
- Used for version vector tracking in the CRDT

```
// Client receives ID assignment
if (message.startsWith("[Server] You are Client ID: ")) {
    clientId = message.section(':', -1).trimmed()[0].toLatin1();
}
```

0.5 Server Implementation

The Python WebSocket server manages client connections and message routing with the following key features:

0.5.1 Core Functionality

```python
async def echo(websocket):
    # Assign client ID (A-Z)
    if available_ids:
        client_id = available_ids.pop(0)
    else:
        client_id = '?'
    assigned_ids[websocket] = client_id

    # Send assignment message
    await websocket.send(f"[Server] You are Client ID: {client_id}")

    # Replay message history for new clients
    for old_message in message_history:
        await websocket.send(old_message)

    # Message handling loop
    async for message in websocket:
        message_history.append(message)
        # Broadcast to all other clients
        for client in connected_clients:
            if client != websocket:
                await client.send(message)
```

Listing 5: WebSocket Server Implementation

0.5.2 Key Features

- **Client Management**:
  - Tracks connected clients in a set
  - Maintains available IDs (A-Z)
  - Recycles IDs on client disconnect
- **Message Handling**:
  - Maintains complete message history
  - Replays history to new clients
  - Broadcasts messages to all clients except sender
- **State Management**:
  - Clears message history when last client disconnects
  - Uses asynchronous I/O for high concurrency

0.5.3 Limitations

- Maximum 26 concurrent clients (A-Z IDs)
- No message persistence beyond current session
- Basic broadcast strategy (no targeted delivery)

0.6 Data Flow

The complete workflow follows this sequence:

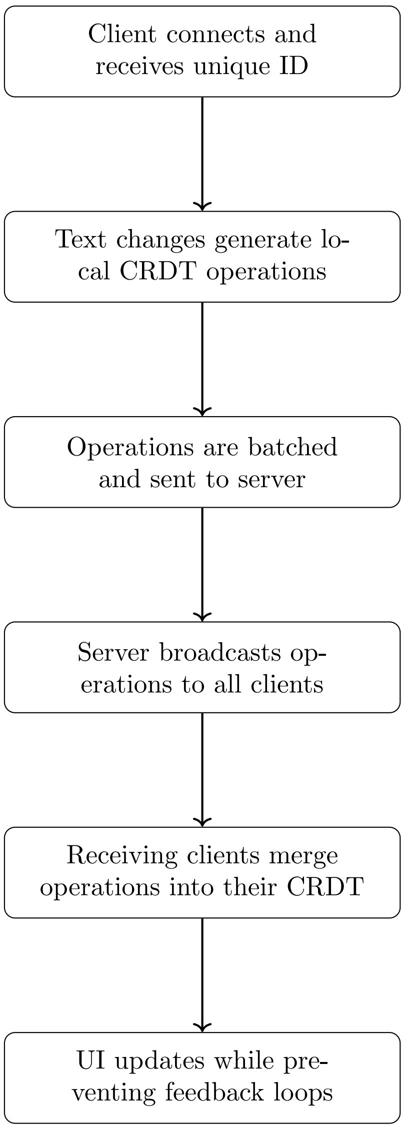
Client connects and receives unique ID
Text changes generate local CRDT operations
Operations are batched and sent to server
Server broadcasts operations to all clients
Receiving clients merge operations into their CRDT
UI updates while preventing feedback loops

Figure 1: Real-Time Collaborative Editing Flow

Additional Features

0.7 Offline Support & Automatic Reconnection

Our implementation provides robust offline support through several key mechanisms:

0.7.1 Local CRDT Operations

All edits are first applied to the local CRDT replica and added to **allOperations**

```
// Client-side text change handling
void MainWindow::onTextChanged() {
    // ... (remaining logic)

    if(currentText.length() > LastKnownText.length()){
        // ... (remaining logic)

        QJsonObject op;
        op["type"] = "insert";
        op["id"] = QString::fromStdString(id);
        op["value"] = inserted;
        op["prev_id"] = QString::fromStdString(prev_id);
        map<char, int> temp_vector = r1.getNodeVersionVector(id);
        QJsonObject versionVec;
        for (const auto& [client, seq] : temp_vector) {
            versionVec[QString(client)] = seq;
        }
        op["version"] = versionVec;
        charAdded += 1;
        allOperations.push_back(op);
    }

    // ... (remaining logic)

    LastKnownText = currentText;
    debounceTimer.start(3000);
}
```

0.7.2 Reconnect Logic

If the websocket server is unable to automatically connect the client to the server, our manual reconnection logic is used

```
void MainWindow::onConnect() {
    webSocket.abort();
    QCoreApplication::processEvents();
    QTimer::singleShot(100, [this]() {
        // webSocket.open(QUrl("ws://192.168.213.150:12345"));
        webSocket.open(QUrl("wss://5be6-111-88-46-136.ngrok-free.app"));
        statusBar()->showMessage("Reconnecting...");
    });
    connect(&webSocket, &QWebSocket::connected, this, [this]() {
        isConnected = true;
        statusBar()->showMessage("Connected");
        sendTextMessage();
    });
}
```

0.7.3 State Synchronization

Full CRDT state transfer on reconnect, and user's local operations sent as the user calls **sendTextMessage()** in **onConnect()**

```
# Server-side message history
message_history = []
async def echo(websocket):
    for old_message in message_history:
        await websocket.send(old_message)
```

Feature	Implementation
Connection Monitoring	5-second ping/pong heartbeat
Message Debouncing	3-second local batch window
Offline Edits	CRDT with version vectors

0.7.4 Reconnection Behavior

- **Brief Disconnects** ($< 5s$): Automatic recovery.
- **Extended Disconnects** ($> 5s$): Manual reconnect required.
- **5-second Ping/Pong**: Matches server/client timeout settings to ensure correct operations execution.

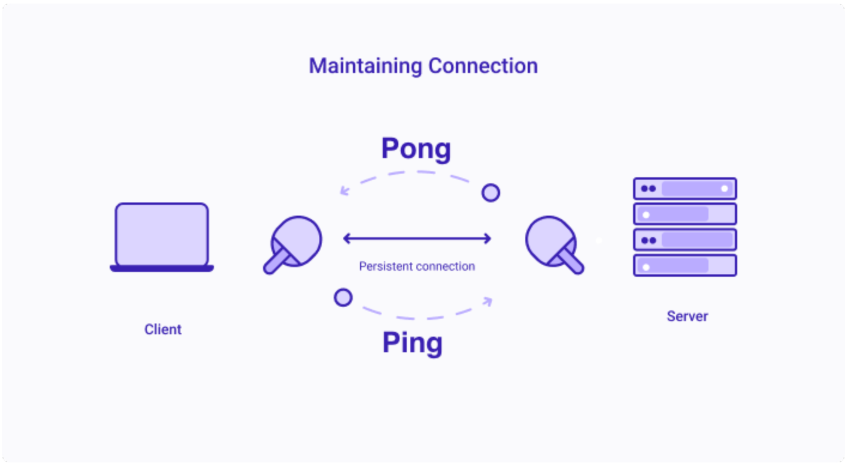

Figure 2: Client reconnection state diagram

0.8 Public Domain Access via Ngrok

We enabled Internet access through ngrok to overcome local network limitations:

- **Architecture**:
  1. Local WS server at `ws://localhost:12345`
  2. Ngrok HTTP tunnel: `ngrok http 12345`
  3. Public WSS endpoint: `wss://<subdomain>.ngrok-free.app`
- **Client Configuration**:

```
// Switch between local and public endpoints
webSocket.open(QUrl("ws://192.168.213.150:12345")); // Local
// webSocket.open(QUrl("wss://52fb-103-125-241-66.ngrok-free.app"));
```

- **Security Upgrade**:
  - Local: Unencrypted `ws://`

  - Public: TLS-encrypted `wss://`
  - Automatic certificate handling by ngrok

Key Benefits:

- No router configuration required
- Supports testing across different networks
- Simplified deployment workflow

Limitations:

- Ngrok free tier restrictions (40 connections/min)
- Random public URLs require reconfiguration
- Potential latency increase

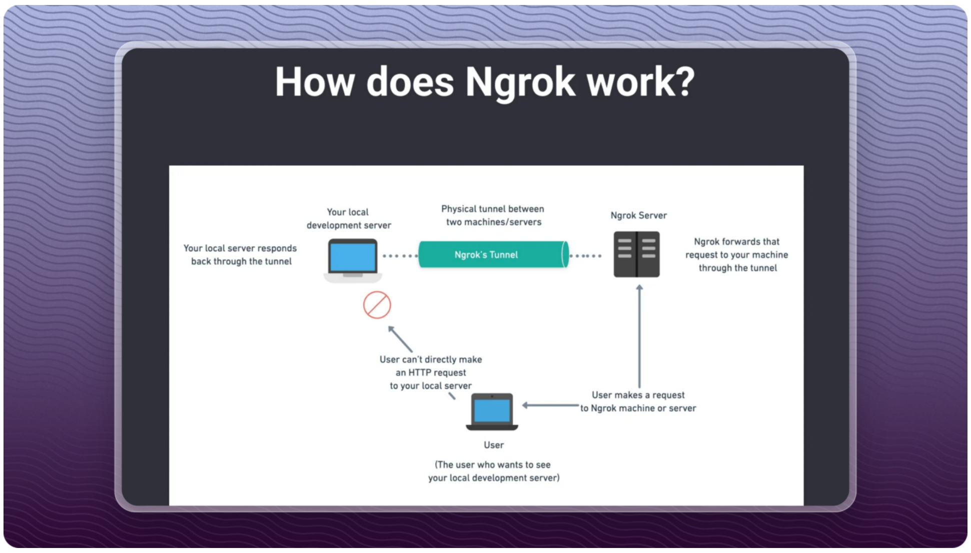
How does Ngrok work?
Your local development server
Physical tunnel between two machines/servers
Ngrok Server
Your local server responds back through the tunnel
Ngrok's Tunnel
Ngrok forwards that request to your machine through the tunnel
User can't directly make an HTTP request to your local server
User makes a request to Ngrok machine or server
User
(The user who wants to see your local development server)

Figure 3: Ngrok HTTP architecture

0.9 File System Integration

The editor includes server-based file management capabilities that enable users to save and load documents from a central repository. This system maintains both the document content and the CRDT state for collaborative editing.

0.9.1 Key Functions

- **File Operations**:
  - `loadFileFromServer`: Requests a file from the server by filename
  - `saveFileToServer`: Saves both content and CRDT state to server
  - `requestFileList`: Retrieves available files from server
- **Message Handling**:

```
void MainWindow::handleFileMessage(const QJsonObject &obj) {
    QString type = obj["type"].toString();

    if (type == "file_content") {
        // Load content and CRDT state
        QString content = obj["content"].toString();
        r1.deserializeState(obj["crdt_state"].toString().toStdString());
        textEdit->setPlainText(content);
    }
    else if (type == "file_list") {
        // Show available files dialog
        QStringList fileNames = /* parse JSON array */;
        QString selectedFile = QInputDialog::getItem(/* file list */);
        loadFileFromServer(selectedFile);
    }
}
```

- **State Preservation**:
  - Saves complete CRDT state with version vectors
  - Maintains character sequence counter (`charAdded`)
  - Preserves current filename context

```
void MainWindow::saveFileToServer(const QString& filename) {
    QJsonObject message;
    message["type"] = "save_file";
    message["filename"] = filename;
    message["content"] = textEdit->toPlainText();
    message["crdt_state"] = QString::fromStdString(r1.serializeState());
    message["char_added"] = charAdded;
    webSocket.sendTextMessage(QJsonDocument(message).toJson());
}
```

0.9.2 Workflow

1. User requests file list via `requestFileList()`
2. Server responds with available filenames
3. User selects file, triggering `loadFileFromServer()`
4. Server sends file content and CRDT state
5. Client reconstructs editor state including:
   - Document text
   - CRDT node structure
   - Version vectors
   - Character sequence counter
6. Subsequent edits synchronize through normal CRDT operations

0.9.3 Limitations and Proposed Solutions

- **No File Deletion**:
  - *Current Limitation*:
    - System lacks functionality to remove files from server
  - *Proposed Solution*:
    - Implement `deleteFile` command in server protocol
    - Add client-side UI option for file deletion
    - Example implementation:

```
# Server-side deletion handler
async def handle_delete(filename):
    if filename in file_storage:
        del file_storage[filename]
        await broadcast({"type": "file_deleted", "filename": filename})
```

- **No File Distinction**:
  - *Current Limitation*:
    - All clients receive all operations regardless of active file
    - Cross-file contamination (File2 edits appear in File1)
    - No document-level isolation
  - *Proposed Solution*:
    - Extend RGA nodes with file identifier field
    - Modify CRDT operations to include file context:

```
struct RGA_Node {
    string id;
    string value;
    string file_id; // New field for file association
    map<char, int> version_vector;
    string prev_id;
};
```

    - Filter operations by current file ID on client side
    - Add file context to all WebSocket messages:

```
{
    "type": "insert",
    "file_id": "doc123",
    "id": "A15",
    "value": "x",
    ...
}
```

Limitation	**Impact**	**Solution**
No file deletion	Server storage bloat	Add delete command + UI
No file isolation	Cross-file contamination	File IDs in CRDT nodes

References

- Ngrok - Secure introspectable tunnels to localhost