

L'implémentation d'application: SecureChat.

MOARRAF YASSINE

(Ce travail fait partie d'un projet de fin d'année impliquant des recherches sur les réseaux peer-to-peer dans le développement d'une application de messagerie peer-to-peer.)

6.1. Front end:

- React components

Pour la réalisation de notre application front-end, nous avons opté pour React, une bibliothèque JavaScript populaire pour la construction d'interfaces utilisateur. React nous permet de créer des composants réutilisables, ce qui facilite la gestion et la maintenance de notre code. Voici un aperçu des principaux composants utilisés dans notre application :

- **App.jsx** : Le composant principal de notre application. Il sert de point d'entrée et intègre les différents composants enfants. Il est responsable de la gestion de l'état global et de la structure de base de l'application.
- **Main.jsx** : Ce composant gère la logique principale de l'application. Il contient la majorité des interactions de l'utilisateur et les affichages dynamiques.

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import './App.css';

ReactDOM.render(<App />, document.getElementById('root'));
```

Main.jsx est le point d'entrée de notre application. Il importe le composant principal **App** et utilise **ReactDOM.render** pour monter ce composant dans un élément DOM avec l'identifiant **root**.

- **Utils.js** : Un fichier utilitaire qui contient des fonctions auxiliaires utilisées dans toute l'application, comme le `friend_request_response`, qui envoie une réponse de demande d'ami via WebSocket, et des styles comme `dark_theme_style_container` pour le thème sombre.

```
import React from 'react';

import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';

import Home from './components/Home';

import About from './components/About';

import './App.css';

function App() {

  return (

    <Router>

      <div className="App">

        <Switch>

          <Route path="/" exact component={Home} />

          <Route path="/about" component={About} />

        </Switch>

      </div>

    </Router>

  );

}

export default App;
```

`App.jsx` gère la navigation de l'application. Il utilise `react-router-dom` pour configurer les routes et définir les composants à afficher pour différentes URLs. Les composants `Home` et `About` sont importés et utilisés pour illustrer les différentes pages de l'application:

- **Home.jsx :**

```
import React from 'react';

function Home() {

  return (
    <div className="Home">
      <h1>Welcome to the Home Page</h1>
    </div>
  );
}

export default Home;
```

`Home . jsx` est un composant fonctionnel qui affiche un message de bienvenue.

- **About.jsx :**

```
import React from 'react';
```

```
function About() {

  return (
    <div className="About">
      <h1>About Us</h1>
    </div>
  );
}
```

```
export default About;
```

`About.js` est un composant fonctionnel qui affiche des informations sur l'application ou l'organisation.

- **App.css** : Le fichier CSS principal pour la mise en style de notre application. Il contient des styles globaux qui s'appliquent à l'ensemble de l'application, assurant une apparence cohérente.

```
.App {  
    text-align: center;  
}  
  
.Home, .About {  
    margin: 20px;  
}
```

`App.css` définit les styles de base pour les composants de l'application, comme l'alignement du texte et les marges.

Les composants React sont structurés pour être modulaires et maintenables, ce qui simplifie l'ajout de nouvelles fonctionnalités et le débogage.

- Websocket

Pour la communication en temps réel, notre application utilise WebSocket. WebSocket est un protocole qui permet des communications bidirectionnelles entre le client et le serveur sur une seule connexion TCP. Cette technologie est essentielle pour les applications qui nécessitent des mises à jour en temps réel, comme les chats en ligne, les jeux multijoueurs, ou les notifications instantanées.

- **WebSocket Initialization** : Dans notre application, nous initialisons WebSocket dans le composant principal et maintenons une connexion persistante avec le serveur. Cela nous permet d'envoyer et de recevoir des messages instantanément.

- **Handling WebSocket Events** : Nous avons défini divers événements WebSocket pour gérer différentes interactions utilisateur. Par exemple, la fonction `friend_request_response` dans `utils.js` envoie un message JSON au serveur pour accepter une demande d'ami. Chaque événement WebSocket est associé à une fonction spécifique qui gère la logique nécessaire.
- **State Management** : La gestion de l'état dans une application WebSocket peut être complexe en raison des mises à jour fréquentes. Nous utilisons les hooks de React, comme `useState` et `useEffect`, pour gérer l'état et les effets de manière propre et efficace.

En conclusion, l'utilisation de React pour les composants front-end, combinée à WebSocket pour les communications en temps réel, nous permet de créer une application réactive, dynamique et maintenable. Les composants sont bien structurés et les communications en temps réel sont gérées de manière efficace, garantissant une expérience utilisateur fluide et réactive.

Description des Images de l'Interface Utilisateur

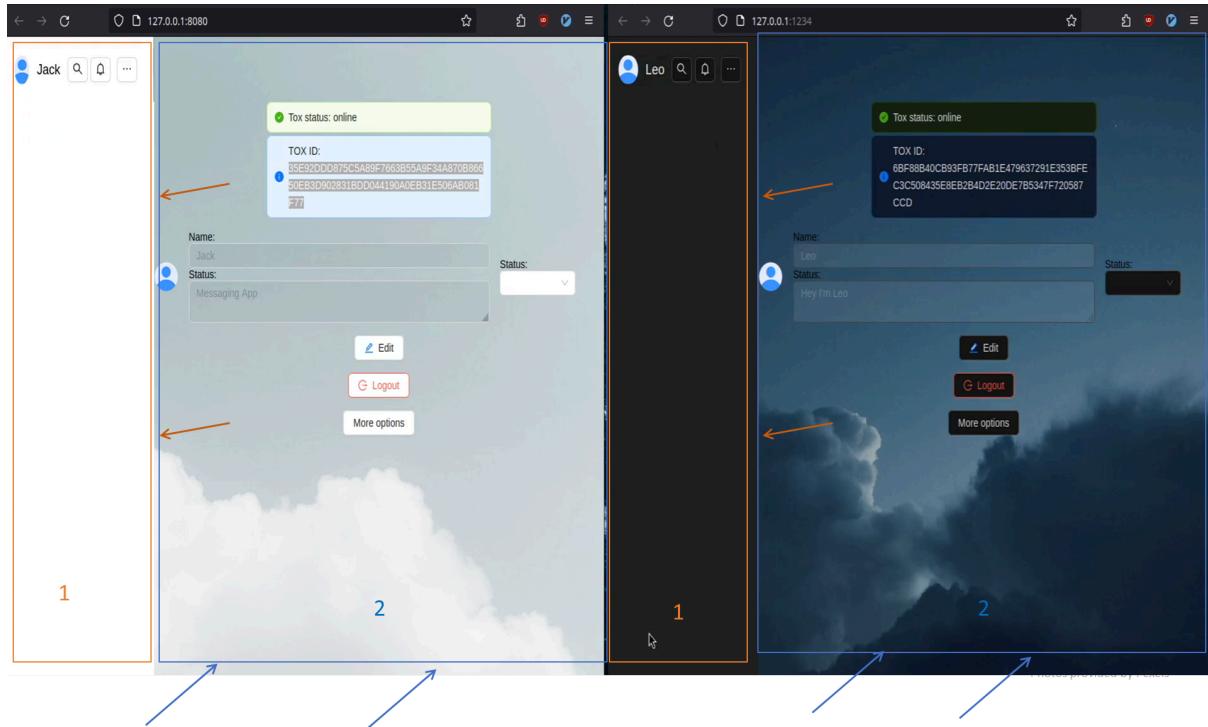
Si vous avez des images spécifiques de l'interface utilisateur, voici comment les intégrer dans la description du rapport :

1. **Page d'accueil (Home)** L'image de la page d'accueil montre un titre accueillant avec un design simple et épuré. Cela met en évidence la convivialité et l'accessibilité de l'application dès l'arrivée de l'utilisateur.
2. **Page 'À propos' (About)** L'image de la page 'À propos' présente des informations clés sur l'application ou l'organisation, permettant aux utilisateurs de mieux comprendre l'objectif et la mission de l'application.

```

294
295
296 return exited == true ? <p>You can now close this tab.</p> : (
297   <ConfigProvider theme={({
298     algorithm: globalStat.dark_theme_enabled ? theme.darkAlgorithm : theme.defaultAlgorithm
299   })>
300     <globalContext.Provider value={{globalStat, dispatch, messageSentBtnHandler, notification, app_exit}}>
301       <SendRequest handler={sendFrReqHandler} />
302       <Row style={{maxHeight: '100vh'}}>
303         <Col span={6} style={{backgroundColor: globalStat.dark_theme_enabled ? "rgb(30, 30, 30)" : 'white'}}> 1
304           <ChatsSidebar />
305         </Col>
306         <Col span={18}>
307           {
308             globalStat.currentFocusedFriend == -1 ?
309               <UserInfoWindow /> :
310               <MessagingWindow messageSentBtnHandler={messageSentBtnHandler}
311                 defaultView={globalStat.currentFocusedFriend == -1}
312                 showSideBarToggle={ () => setRecipientPropertiesSidebar(!showRecipientPropertiesSidebar) }/>
313           }
314         </Col> 2
315       </Row>
316     </globalContext.Provider>
317   </ConfigProvider>
318 )
319
320
321
322
323
324
325 export default App

```



- Persistance avec client side storage: IndexedDB

Introduction

IndexedDB est une base de données NoSQL intégrée aux navigateurs web, idéale pour stocker de grandes quantités de données structurées localement. Cette section explore l'utilisation de IndexedDB pour assurer la persistance des données dans une application de chat P2P utilisant Tox. La gestion efficace des données est cruciale pour garantir une expérience utilisateur fluide et fiable.

Intégration avec Tox dans une Application de Chat P2P

L'intégration de IndexedDB avec Tox dans notre application de chat P2P assure la persistance des messages et des métadonnées. Les informations ci-dessous illustrent les étapes de ce processus, garantissant une gestion efficace et sécurisée des données de l'utilisateur.

Explications:

Vue d'Ensemble de l'Application

- Les différentes méthodes de stockage disponibles sont visibles dans l'interface de chat a savoir : Local Storage, Session Storage, IndexedDB, et Cookies.

Détails de IndexedDB et des Cookies

- Les messages sont stockés dans IndexedDB.
- Les cookies gèrent les sessions utilisateur.

Processus d'Envoi et de Stockage de Messages

Initialisation et Connexion

1. **Génération de Clés :**
 - Chaque utilisateur génère une paire de clés publique/privée à la première utilisation.
 - Ces clés assurent la sécurité des communications.
2. **Connexion Tox :**
 - Une connexion sécurisée au réseau Tox est établie pour découvrir et se connecter à d'autres pairs.
3. **IndexedDB :**
 - IndexedDB est initialisé pour stocker localement les messages et les métadonnées.

Envoi de Message

1. **Création et Stockage Initial :**
 - L'utilisateur compose un message qui est stocké dans IndexedDB avec un statut 'pending'.
2. **Stockage des Métadonnées :**
 - Les métadonnées (horodatage, ID du destinataire) sont stockées dans les cookies pour un accès rapide.

Envoi via Tox

1. **Pousser l'Événement :**
 - Un événement `E_NEW_MESSAGE_SENT` est poussé dans la boucle d'événements.
2. **Traitement de l'Événement :**
 - La fonction `handle_message_sent` récupère le message depuis IndexedDB.
3. **Envoi via Tox :**
 - Le message est envoyé au destinataire via `tox_friend_send_message`.

Confirmation et Mise à Jour

1. **Confirmation d'Envoi :**
 - Après l'envoi, une confirmation est renvoyée à l'application.
2. **Mise à Jour du Statut :**
 - Le statut du message passe de 'pending' à 'sent' dans IndexedDB.
3. **Mise à Jour des Métadonnées :**

- Les métadonnées sont mises à jour dans les cookies.

Réception de Message

- Notification de Réception :**
 - Lorsqu'un message est reçu via Tox, l'application est informée.
- Déchiffrement et Stockage :**
 - Le message est déchiffré et stocké dans IndexedDB avec un statut 'received'.
- Stockage des Métadonnées :**
 - Les métadonnées associées sont stockées dans les cookies.

Autres Méthodes de Stockage Client Side

Local Storage :

- Permet de stocker des paires clé-valeur de manière persistante.
- Les données restent disponibles même après la fermeture du navigateur.

Session Storage :

- Similaire à Local Storage, mais les données sont disponibles uniquement pour la durée de la session de navigation.
- Les données sont supprimées lorsque l'onglet ou la fenêtre du navigateur est fermé.

Cookies :

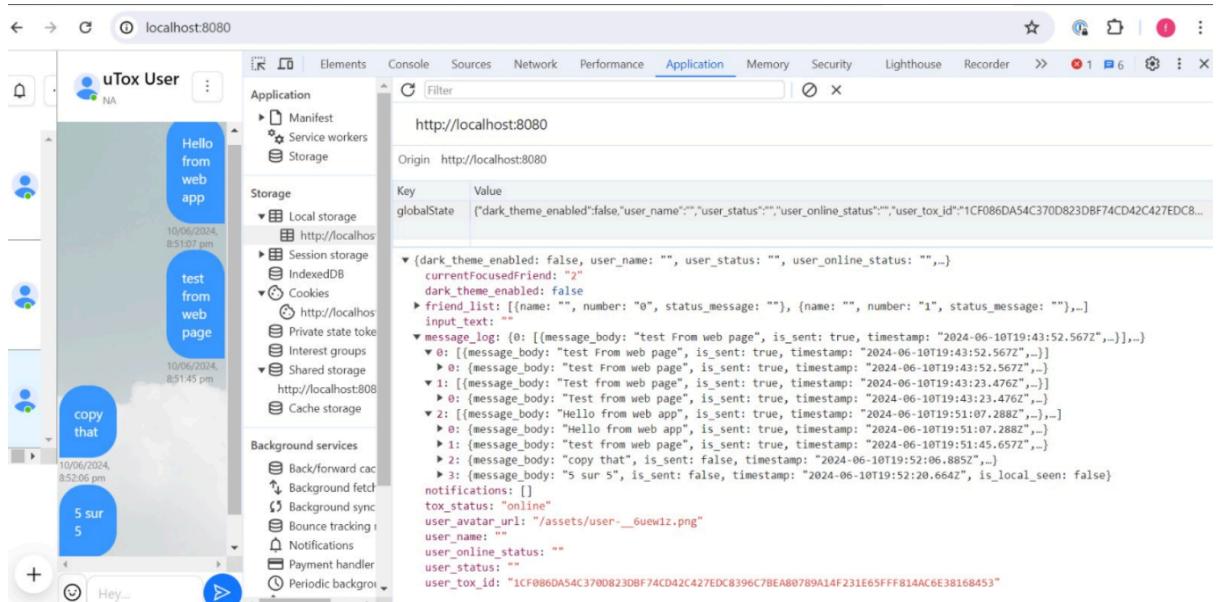
- Utilisés pour stocker des informations de session et des préférences utilisateur.
- Les cookies peuvent avoir une date d'expiration spécifique et sont envoyés avec chaque requête HTTP vers le serveur.

Cache Storage :

- Utilisé principalement pour les Progressive Web Apps (PWAs) pour stocker des ressources réseau, permettant ainsi une meilleure performance et une utilisation hors ligne.

Private State Tokens et Interest Groups :

- Utilisés pour des cas spécifiques comme la gestion des jetons de session privée et les groupes d'intérêt pour la publicité ciblée.



Conclusion

IndexedDB et l'API Web Storage sont des outils essentiels pour assurer la persistance et la fiabilité des communications dans une application de chat P2P. Leur intégration avec Tox permet de garantir que les messages et les métadonnées sont stockés de manière sécurisée et accessible, assurant ainsi une expérience utilisateur fluide et sécurisée. Grâce à ces technologies, il est possible de créer des applications web modernes capables de gérer efficacement de grandes quantités de données tout en maintenant une performance optimale.

6.2. Back end:

6.2.1. Event loop:

Aperçu sur les méthodes de synchronisation entre les threads:

Mutex (Mutual Exclusion Object) : Un mutex est un mécanisme qui permet de verrouiller une ressource afin qu'un seul thread puisse y accéder à la fois. Lorsqu'un thread verrouille un mutex, les autres threads qui tentent de le verrouiller sont bloqués jusqu'à ce que le mutex soit déverrouillé. Cela garantit qu'une section critique du code n'est exécutée que par un seul thread à la fois.

Sémaphores : Les sémaphores sont des variables ou des abstractions plus sophistiquées qui contrôlent l'accès à une ressource partagée. Il en existe deux types : les sémaphores binaires, qui fonctionnent comme des mutex, et les sémaphores comptables, qui permettent de gérer un nombre limité d'accès simultanés à une ressource.

Barrières (Barriers) : Les barrières sont utilisées pour synchroniser un groupe de threads. Elles permettent de s'assurer que tous les threads d'un groupe atteignent un certain point d'exécution avant que l'un d'eux puisse continuer. Cela est utile dans les situations où des phases de traitement parallèles doivent être synchronisées.

Boucles d'événements (Event Loops) : Les boucles d'événements sont utilisées principalement dans les environnements asynchrones pour gérer les opérations I/O de manière efficace. Une boucle d'événements écoute et distribue les événements ou messages dans un programme. Lorsqu'un événement ou un message est détecté, il est placé dans une file d'attente. La boucle d'événements traite alors chaque message en invoquant le gestionnaire d'événements approprié. Cela permet d'éviter le blocage des threads en attente de l'achèvement des opérations I/O, améliorant ainsi la réactivité et la performance des applications.

6.3. Le design pattern "Event loop":

L'approche fondamentale que nous utiliserons, comme indiqué précédemment, est appelée concurrence basée sur les événements. Le concept est assez simple : il suffit d'attendre

qu'un événement se produise ; lorsqu'il survient, vous vérifiez de quel type d'événement il s'agit et vous effectuez la petite quantité de travail requise (ce qui peut inclure l'émission de requêtes d'E/S, ou la programmation d'autres événements pour un traitement ultérieur, etc.).

examinons d'abord à quoi ressemble un serveur basé sur les événements. De telles applications sont construites autour d'une structure simple connue sous le nom de boucle d'événements. Un pseudocode pour une boucle d'événements ressemble à ceci :

```
while (1) {  
    événements = obtenirÉvénements();  
    for (e in événements) traiterÉvénement(e); }
```

La boucle principale attend simplement quelque chose à faire (en appelant `obtenirÉvénements()` dans le code ci-dessus) et ensuite, pour chaque événement retourné, les traite un par un ; le code qui traite chaque événement est connu sous le nom de gestionnaire d'événements. Il est important de noter que lorsque qu'un gestionnaire traite un événement, c'est la seule activité en cours dans le système ; ainsi, décider quel événement traiter ensuite équivaut à une planification. Ce contrôle explicite de la planification est l'un des avantages fondamentaux de l'approche basée sur les événements.

La simplicité de cette approche:

Avec un seul processeur et une application basée sur les événements, les problèmes rencontrés dans les programmes concurrents ne sont plus présents. Plus précisément, parce qu'un seul événement est traité à la fois, il n'est pas nécessaire d'acquérir ou de libérer des verrous ; le serveur basé sur les événements ne peut pas être interrompu par un autre thread car il est résolument monothread. Ainsi, les bogues de concurrence courants dans les programmes multithreads ne se manifestent pas dans l'approche de base basée sur les événements.

6.4. L'intégration L'event loop dans notre application:

L'event loop dans notre projet consiste de deux partie:

- Une file d'attente pour les événement sans réponse et Une file d'attente pour les événement avec réponse.
- Une class qui entoure ces deux fils d'attentats pour récupérer les événement et les traite

Une vue plus détaillée:

Les files d'attente:

L'implémentation des files d'attentats est réalisé par le bibliothèque C++ copper (<https://github.com/atollk/copper>) car il expose:

- La classe `copper::buffered_channel`
- Une famille des méthodes `push/pop` qui sert à bloquer l'exécution si pas d'événement existe dans la file d' attente pour réaliser l'attente des événements.
- `close / is_read_closed` pour fermer ou vérifie si la files d'attente est ferme

Dans notre program il y'a deux files d'attente:

```
copper::buffered_channel<async_event> *main_event_queue;  
copper::buffered_channel<sync_event*> *req_event_queue;
```

Source: src/event_loop.hpp

Comme on a déjà discuté: main_event_queue pour les événement sans réponse et req_event_queue pour les événements avec réponse.

Récupération et traitement des événements:

On a défini deux dictionnaires ou table de hachage (std::map), le clé c'est le type d'événement et la valeur c'est la liste des callback qui écoute cette événement:

```
typedef std::function<void(async_event)> callback_fn;  
typedef std::function<void(sync_event*)> callback_fn_resp;  
std::map<event_type, std::vector<callback_fn*>>  
*callback_list;  
std::map<event_type, std::vector<callback_fn_resp*>>  
*callback_list_resp;
```

Les types événement tout simplement sont définis dans une énumération comme cela:

```
enum event_type {  
  
    E_RESP_GET_FRIEND_STATUS_MSG,  
    E_RESP_GET_FRIEND_NAME,  
    E_RESP_GET_FRIEND_NUMBERS_LIST,  
    E_RESP_GET_USER_ID,  
    E_RESP_SEND_FRIEND_REQ,  
    E_SET_SELF_NAME,  
    E_ACCEPT_FR_REQ,  
    . . .
```

Par exemple, l'événement E_SET_SELF_NAME signifie qu'il y a une demande de changement du nom d'utilisateur au niveau du nœud Tox.

Un événement quand il est envoyé au event loop il vient avec plusieur d'autre information cette information est contenue dans les deux structure:

```
struct async_event {  
    async_event();  
    async_event(event_type _e_type, void * payload);  
    event_type e_type;  
    void * event_payload;
```

```

};

struct sync_event:async_event {
    sync_event();
    sync_event(event_type _e_type, void * payload, uint32_t id);
    uint32_t event_id;
    bool is_request;
};

```

On note principalement que sync_event hérite de async_event donc les deux structure contient le membre event_payload qui consiste d'un pointeur vers les données inclue avec un événement. Dans notre exemple précédent un événement E_SET_SELF_NAME continent comme donnée inclut le nouveau nom, concrètement event_payload dans ce cas point vers un std::string qui contient le nom.

Et finalement la boucle principale qui récupère les événement et réalise le traitement c'est à dire appeler les callback correspondantes:

```

event_loop::main_loop() {
// Tant que la file d'attente n'est pas fermée.
    while (!this->main_event_queue->is_read_closed() ) {
// Récupération d'un événement.
        std::optional<async_event> curr_e_opt =
            this->main_event_queue->pop();
        if (curr_e_opt) {
            async_event curr_e = curr_e_opt.value();
            if (this->callback_list->count(curr_e.e_type) != 0) {
                std::vector<callback_fn> *target_callback_list =
this->callback_list->at(curr_e.e_type);
// Exécution des callbacks
                for (callback_fn cb : *target_callback_list) {
                    cb(curr_e);
                }
                if(curr_e.e_type == event::event_type::SYS_EXIT) {
                    this->main_event_queue->close();
                    this->req_event_queue->close();
                }
            }
        }
    }
}

```

6.2.2. Tox node:

6.2.2.1. Bibliothèque Tox:

La bibliothèque Tox est une bibliothèque de communication sécurisée et décentralisée conçue pour la messagerie instantanée. Elle fournit des fonctionnalités pour envoyer des messages, gérer des contacts, et maintenir des connexions sécurisées entre les utilisateurs. Voici une explication générale de la bibliothèque Tox et de ses fonctionnalités basées sur les informations typiques de `tox.h`.

Fonctionnalités Clés et fonctions de la Bibliothèque Tox :

- Initialisation et Configuration:
 - Fonctions:
 - ❖ `tox_new` : Crée une nouvelle instance Tox.
 - ❖ `tox_options_default` : Initialise les options par défaut pour une instance Tox.
- Gestion des Contacts :
 - Ajouter des amis : La bibliothèque permet d'ajouter des amis à l'aide de leurs clés publiques.
 - Accepter des demandes d'amis : Gérer les demandes d'amis entrantes et les accepter automatiquement ou manuellement.
 - Fonctions:
 - ❖ `tox_friend_add` : Ajoute un nouvel ami à l'aide de sa clé publique.
 - ❖ `tox_friend_delete` : Supprime un ami de la liste de contacts.
 - ❖ `tox_friend_exists` : Vérifie si un ami existe dans la liste de contacts.
- Envoi et Réception de Messages :
 - Envoyer des messages : Envoyer des messages texte ou multimédia à des amis.
 - Recevoir des messages : Recevoir des messages et déclencher des callbacks pour les traiter.
 - Fonctions:
 - ❖ `tox_friend_send_message` : Envoie un message à un ami.

- ❖ `tox_callback_friend_message` : Enregistre un callback pour les messages entrants.
- Gestion de la Connexion :
 - Établir des connexions sécurisées : Utiliser des connexions sécurisées pour la communication entre les utilisateurs.
 - Gérer les statuts de connexion : Suivre les statuts de connexion des amis et de l'utilisateur.
 - Fonctions:
 - ❖ `tox_self_set_status` : Définit le statut de l'utilisateur (en ligne, absent, etc.).
 - ❖ `tox_self_get_connection_status` : Obtient le statut de connexion de l'utilisateur.
- Callbacks :
 - Définir des callbacks : Enregistrer des callbacks pour différents événements comme les messages entrants, les demandes d'amis, les changements de statut de connexion, etc.
 - Fonctions:
 - ❖ `tox_callback_self_connection_status` : Enregistre un callback pour les changements de statut de connexion.
 - ❖ `tox_callback_friend_request` : Enregistre un callback pour les demandes d'amis.

La bibliothèque Tox offre une solution robuste et flexible pour les développeurs soucieux de la sécurité et de la confidentialité de leurs applications de communication. Avec son approche décentralisée et son chiffrement de bout en bout, Tox ouvre la voie à une nouvelle ère de communication en ligne sécurisée.

6.2.2.2. Gestion des noeuds:

Fonctionnement Général “self_node”:

Les fichiers `self_node.cpp` et `self_node.hpp` concernent la gestion d'un nœud Tox dans une application basée sur le protocole Tox, une plateforme de messagerie sécurisée et décentralisée. La classe principale `self_node` initialise, configure et gère les opérations du nœud Tox.

self_node.hpp : Ce fichier d'en-tête définit la classe `self_node` et inclut les bibliothèques nécessaires.

- Les inclusions :

```

EXPLORER      ...
PROJET-WEB-P2P_TOX_TE...
src > tox_node > self_node.hpp > ...
1  #pragma once
2  #include <tox.h>
3  #include <string>
4  #include <list>
5  #include <sodium.h>
6  #include <unistd.h>
7  #include <thread>
8  #include "../event_loop/event_loop.hpp"
9  #include <glog/logging.h>
10
11 namespace tox{
12     struct dht_node {
13         const char *ip;

```

- tox.h : Le fichier d'en-tête principal pour le protocole Tox.
- string, list, thread, unistd.h : Bibliothèques standard C++ pour la manipulation des chaînes, la gestion des listes, le threading et les fonctions standards UNIX.
- sodium.h : Une bibliothèque pour le chiffrement, le déchiffrement, les signatures, le hachage de mots de passe, etc.
- event_loop.hpp : Un en-tête personnalisé, probablement définissant une boucle d'événements pour les opérations asynchrones.
- glog/logging.h : Une bibliothèque pour la journalisation.

- Classe self_node :
- La classe self_node est définie pour gérer les opérations principales d'un nœud Tox.

```

C++ self_node.cpp 2, M      h++ self_node.hpp 5 X      h++ tox_callbacks.hpp      C++ tox_callbacks.cpp 2
src > tox_node > h++ self_node.hpp > ...
11  namespace tox{
12    ,
13    class self_node {
14      public:
15        self_node( event::event_loop *main_event_loop,
16                  std::list<dht_node> *dht_node_list = nullptr,
17                  std::string *serialization_path = nullptr );
18        ~self_node();
19        std::thread spawn();
20        void stop_instance();
21        event::event_loop *main_event_loop;
22      private:
23        friend class self_node_cb;
24        void node_bootstrap();
25        bool setup_options();
26        void set_tox_id();
27        void connect_cb();
28        std::string * serialization_path;
29        std::string * user_name;
30        std::string * user_status;
31        std::string * user_tox_id;
32        std::list<dht_node> *dht_node_list;
33        Tox_Options node_options;
34
35        bool node_first_run;
36        /* node_status */
37        Tox* tox_c_instance;
38        void main_loop();
39        void register_handlers();
40        void register_tox_callbacks();
41        void update_savedata_file();
42        bool auto_accept = false;
43        bool enable_trace = false;
44        const char *savedata_filename = "savedata.tox";
45        const char *savedata_tmp_filename = "savedata.tox.tmp";
46
47    };
48
49
50  };
51

```

- Les attributs :
 - ❖ event::event_loop *main_event_loop : Pointeur vers la boucle principale d'événements.

- ❖ std::list<dht_node> *dht_node_list : Pointeur vers une liste de nœuds DHT (Distributed Hash Table).
 - ❖ std::string *serialization_path : Pointeur vers le chemin de sérialisation pour sauvegarder l'état du nœud.
 - ❖ std::string *user_name : Pointeur vers le nom d'utilisateur.
 - ❖ std::string *user_status : Pointeur vers le statut de l'utilisateur.
 - ❖ std::string *user_tox_id : Pointeur vers l'ID Tox de l'utilisateur.
 - ❖ Tox_Options node_options : Options pour l'instance Tox.
 - ❖ bool node_first_run : Indique s'il s'agit de la première exécution du nœud.
 - ❖ Tox* tox_c_instance : Pointeur vers l'instance Tox.
 - ❖ bool auto_accept : Booléen indiquant s'il faut accepter automatiquement les demandes d'amis.
 - ❖ bool enable_trace : Booléen indiquant s'il faut activer le traçage.
 - ❖ const char *savedata_filename : Nom de fichier pour les données sauvegardées.
 - ❖ const char *savedata_tmp_filename : Nom de fichier temporaire pour les données sauvegardées.
- Les méthodes :
- ❖ Constructeur (self_node) : Initialise l'objet self_node avec une boucle d'événements, une liste de nœuds DHT et un chemin de sérialisation.
 - ❖ Destructeur (~self_node) : Nettoie les ressources.
 - ❖ update_savedata_file() : Récupère la taille des données sauvegardées et les écrit dans un fichier temporaire, puis le renomme en fichier de données sauvegardées.
 - ❖ connect_cb() : Définit le nœud actuel et enregistre les gestionnaires et les callbacks Tox.
 - ❖ set_tox_id() : Obtient l'adresse Tox et la convertit en chaîne hexadécimale, puis enregistre l'ID Tox.
 - ❖ setup_options() : Configure les options pour l'instance Tox, y compris la lecture des données sauvegardées si disponibles.
 - ❖ main_loop() : Exécute la boucle principale, itérant continuellement l'instance Tox et dormant pendant l'intervalle d'itération.
 - ❖ spawn() : Crée un nouveau thread pour exécuter la boucle principale.
 - ❖ stop_instance() : Arrête l'instance Tox en appelant tox_kill.

self_node.cpp : Ce fichier d'implémentation définit les méthodes déclarées dans le fichier d'en-tête `self_node.hpp` .

- Méthodes Implémentées :
- Constructeur (self_node) :
 - ❖ Initialise l'objet `self_node` avec une boucle d'événements, une liste de nœuds DHT et un chemin de sérialisation.
 - ❖ Configure les options Tox par défaut.
 - ❖ Initialise le nœud Tox.
 - ❖ Met à jour le fichier de données sauvegardées.
 - ❖ Définit l'ID Tox de l'utilisateur.
 - ❖ Connecte les callbacks.

```

C++ self_node.cpp 2, M X h++ self_node.hpp 5 h++ tox_callbacks.hpp C++ tox_callbacks.cpp 2
src > tox_node > C++ self_node.cpp > main_loop()
17     self_node::self_node(event::event_loop *main_event_loop,
18     std::list<dht_node> *dht_node_list ,
19     std::string *serialization_path)
20     : main_event_loop(main_event_loop),
21     dht_node_list(dht_node_list),
22     serialization_path(serialization_path)
23 {
24     tox_options_default(&node_options);
25     if (enable_trace) node_options.log_callback = self_node_cb::log;
26
27     setup_options();
28     tox_c_instance = tox_new(&node_options, NULL);
29     /* this->user_name = new std::string("test dev build"); */ //
30     /* this->user_status = new std::string("test dev build"); */
31     node_bootstrap();
32     update_savedata_file();
33     set_tox_id();
34     connect_cb();
35 }

```

- update_savedata_file() :
 - ❖ Récupère la taille des données sauvegardées.
 - ❖ Alloue de la mémoire pour les données.
 - ❖ Écrit les données dans un fichier temporaire.
 - ❖ Renomme le fichier temporaire en fichier de données sauvegardées.

```

C++ self_node.cpp 2, M X h++ self_node.hpp 5 h++ tox_callbacks.hpp C++ tox_callbacks.cpp 2
src > tox_node > C++ self_node.cpp > main_loop()
7     void self_node::update_savedata_file() {
8         size_t size = tox_get_savedata_size(tox_c_instance);
9         uint8_t *savedata = (uint8_t*)malloc(size);
10        tox_get_savedata(tox_c_instance, savedata);
11        FILE *f = fopen(savedata_tmp_filename, "wb");
12        fwrite(savedata, size, 1, f);
13        fclose(f);
14        rename(savedata_tmp_filename, savedata_filename);
15    }

```

- `connect_cb()` :
 - ❖ Définit le nœud actuel et enregistre les gestionnaires et les callbacks Tox.

C++ self_node.cpp 2, M X h++ self_node.hpp 5 h++ tox_callbacks.hpp

```
src > tox_node > C++ self_node.cpp > update_savedata_file()

36
37     void self_node::connect_cb() {
38         self_node_cb::curr_node = this;
39         self_node_cb::register_handlers();
40         self_node_cb::register_tox_callbacks();
41     }
```

- `set_tox_id()` :
 - ❖ Obtient l'adresse Tox.
 - ❖ Convertit l'adresse en chaîne hexadécimale.
 - ❖ Enregistre l'ID Tox.

```
42     void self_node::set_tox_id() {
43         uint8_t tox_id_bin[TOX_ADDRESS_SIZE];
44         tox_self_get_address(this->tox_c_instance, tox_id_bin);
45
46         char tox_id_hex[TOX_ADDRESS_SIZE*2 + 1];
47         sodium_bin2hex(tox_id_hex, sizeof(tox_id_hex), tox_id_bin, sizeof(tox_id_bin));
48
49         for (size_t i = 0; i < sizeof(tox_id_hex)-1; i++) {
50             tox_id_hex[i] = toupper(tox_id_hex[i]);
51         }
52         this->user_tox_id = new std::string(tox_id_hex);
53         LOG(INFO) << "id: " << tox_id_hex << '\n';
54     }
```

- `setup_options()`:
 - ❖ Configure les options pour l'instance Tox.
 - ❖ Lit les données sauvegardées si disponibles.

```

56   bool self_node::setup_options() {
57     FILE *f = fopen(savedata_filename, "rb");
58     node_first_run = f == NULL;
59     if (f) {
60       fseek(f, 0, SEEK_END);
61       long fsize = ftell(f);
62       fseek(f, 0, SEEK_SET);
63       uint8_t *savedata = (uint8_t*)malloc(fsize);
64       fread(savedata, fsize, 1, f);
65       fclose(f);
66       node_options.savedata_type = TOX_SAVEDATA_TYPE_TOX_SAVE;
67       node_options.savedata_data = savedata;
68       node_options.savedata_length = fsize;
69     }
70     return f != NULL;
71 }
```

- `node_bootstrap()`:

La fonction `node_bootstrap()` est une méthode de la classe `self_node` utilisée pour initialiser la connexion d'un nœud Tox avec le réseau Tox. Elle permet au nœud de se connecter aux nœuds DHT (Distributed Hash Table) pour rejoindre le réseau Tox et commencer à communiquer avec d'autres nœuds. Cette fonction est cruciale pour établir la présence du nœud sur le réseau.

```

src > tox_node > C++ self_node.cpp > node_bootstrap()
72 void self_node::node_bootstrap() {
73   dht_node nodes[] =
74   {
75     {"85.143.221.42", 33445, "DA4E4ED4B697F2E9B000EEFE3A34B554ACD3F},
76     /* {"2a04:ac00:1:9f00:5054:ff:fe01:be0d", 33445, "DA4E4ED4B697F2E9B000EEFE3A34B554AC},
77     {"78.46.73.141", 33445, "02807CF4F8BB8FB390CC3794BDF1E8449E9A8},
78     /* {"2a01:4f8:120:4091::3", 33445, "02807CF4F8BB8FB390CC3794BDF1E8449E},
79     {"tox.initramfs.io", 33445, "3F0A45A268367C1BEA652F258C85F4A66DA76},
80     {"tox2.abilinski.com", 33445, "7A6098B590BDC73F9723FC59F82B3F9085A64},
81     {"205.185.115.131", 53, "3091C6BEB2A993F1C6300C16549FABA67098F},
82     {"tox.kurnevsky.net", 33445, "82EF82BA33445A1F91A7DB27189ECFC0C013E},
83   };
84   for (size_t i = 0; i < sizeof(nodes)/sizeof(dht_node); i++) {
85     unsigned char key_bin[TOX_PUBLIC_KEY_SIZE];
86     sodium_hex2bin(key_bin, sizeof(key_bin), nodes[i].key_hex, sizeof(nodes[i].key_hex)-
87     NULL, NULL, NULL);
88     tox_bootstrap(this->tox_c_instance, nodes[i].ip, nodes[i].port, key_bin, NULL);

```

❖ Fonctionnement de node_bootstrap() :

Boucle sur les nœuds DHT :

- o La fonction itère sur chaque nœud DHT dans la liste dht_node_list.
- o Chaque nœud DHT est représenté par une structure dht_node contenant les informations suivantes :
 - o ip : L'adresse IP du nœud DHT.
 - o port : Le port du nœud DHT.
 - o key_hex : La clé publique du nœud DHT en format hexadécimal.

Bootstrap Tox :

- o Pour chaque nœud DHT, la fonction appelle tox_bootstrap() pour connecter l'instance Tox (tox_c_instance) au réseau Tox via ce nœud DHT.
- o tox_bootstrap() prend les paramètres suivants :
 - o tox_c_instance : L'instance Tox.
 - o node.ip : L'adresse IP du nœud DHT.
 - o node.port : Le port du nœud DHT.
 - o node.key_hex : La clé publique du nœud DHT en format hexadécimal.
 - o NULL : Paramètre utilisateur optionnel, ici non utilisé.

- main_loop() :

- ❖ Exécute la boucle principale.
- ❖ Itère continuellement l'instance Tox.
- ❖ Dort pendant l'intervalle d'itération.

```
96     }
97     void self_node::main_loop() {
98         Tox* tox = this->tox_c_instance;
99         while (1) {
100             tox_iterate(tox, NULL);
101
102             usleep(tox_iteration_interval(tox) * 1000);
103         }
104     }
```

- spawn() :

- ❖ Crée un nouveau thread pour exécuter la boucle principale.

- stop_instance() :
 - ❖ Arrête l'instance Tox en appelant `tox_kill`.

```

106     std::thread self_node::spawn() {
107         return std::thread(&self_node::main_loop, this);
108     }
109     void self_node::stop_instance() {
110         tox_kill(tox_c_instance);
111     }
112

```

6.2.2.3. Gestion des callbacks

Fonctionnement Général “tox_callbacks”:

Les fichiers `tox_callbacks.cpp` et `tox_callbacks.hpp` sont utilisés pour définir et implémenter les fonctions de rappel (callbacks) qui gèrent les événements spécifiques du protocole Tox. Ces événements incluent les demandes d'amis, les messages reçus, les changements de statut de connexion, etc. Les callbacks sont enregistrés avec l'instance Tox et sont automatiquement appelés lorsqu'un événement correspondant se produit.

tox_callbacks.hpp : Ce fichier d'en-tête déclare les fonctions de rappel et les méthodes utilisées pour enregistrer ces callbacks.

- Les inclusions :
- tox.h : Le fichier d'en-tête principal pour le protocole Tox.
- string, list : Bibliothèques standard C++ pour la manipulation des chaînes et des listes.
- event_loop.hpp : Un en-tête personnalisé, probablement définissant une boucle d'événements pour les opérations asynchrones.
- glog/logging.h : Une bibliothèque pour la journalisation.

- Namespace tox: l'espace de noms `tox` contient les déclarations des fonctions de rappel et des méthodes pour les enregistrer.

```

EXPLORER ... C++ self_node.cpp 2, M h++ self_node.hpp 5 h++ tox_callbacks.hpp 2 x C++ tox_callbacks.cpp 2
P2P_CHAT_APP src > tox_node > h++ tox_callbacks.hpp > {} tox
> build
> c-toxcore
> docs
> include
> libs
src > back_end
> event_loop
tox_node
  C++ self_no... 2, M
  h++ self_node... 5
  C++ tox_callba... 2
  h++ tox_callba... 2
  C++ main.cpp
> test
> web_front
  $system_dia...
  .gitignore
OUTLINE
24

3
namespace tox{
4
    class self_node_cb {
5
        private:
6
            friend class self_node;
7
            static self_node *curr_node;
8
9
        static void register_handlers();
10
        static void register_tox_callbacks();
11
12
        static void log(Tox *tox, Tox_Log_Level level, const char *file, uint32_t line,
13
                        const char *message, void *user_data);
14
15
        // handlers to handle events from the event loop
16
        static void handle_message_sent(event::async_event e);
17
18
        static void handle_friend_list_req(event::sync_event * e);
19
20
        static void handle_friend_get_name(event::sync_event * e);
21
22
        static void handle_friend_get_status_message(event::sync_event * e);
23
        static void handle_friend_accept(event::async_event e);
        static void handle_get_user_id(event::sync_event *e);
24

25
// tox callbacks: put events in the event loop
26
static void friend_request_cb(Tox *tox, const uint8_t *public_key,
27
                            const uint8_t *message, size_t length,
28
                            void *user_data); // rn we accept all requests
29
30
static void friend_message_cb(Tox *tox, uint32_t friend_number,
31
                            TOX_MESSAGE_TYPE type, const uint8_t *message,
32
                            size_t length, void *user_data);
33
34
static void self_connection_status_cb(Tox *tox, TOX_CONNECTION connection_status,
35
                                      void *user_data);
36
static void friend_name_cb(
37
    Tox *tox, Tox_Friend_Number friend_number,
38
    const uint8_t name[], size_t length, void *user_data);
39
40
static void friend_status_cb(
41
    Tox *tox, Tox_Friend_Number friend_number, Tox_User_Status status, void
42
)
43
44

```

```

> docs
> include
> libs
> src
> back_end
> event_loop
> tox_node
  C++ self_no... 2, M
  h++ self_node.... 5
  C++ tox_callback... 2
45
46
47
48
49
50
51
52
53
54
55
    static void friend_connection_status_cb(
        Tox *tox, Tox_Friend_Number friend_number, Tox_Connection connection_st
    );
    static void friend_typing_cb(
        Tox *tox, Tox_Friend_Number friend_number, bool typing, void *user_data)
    static void friend_read_receipt_cb(
        Tox *tox, Tox_Friend_Number friend_number, Tox_Friend_Message_Id message
);
}

```

tox_callbacks.cpp : Ce fichier d'implémentation définit les fonctions de rappel et les méthodes pour enregistrer ces callbacks.

- Méthodes Implémentées
 - Register_handlers() :
 - ❖ Enregistre les gestionnaires d'événements pour traiter divers événements de la boucle d'événements.

```

void self_node_cb::register_handlers() {
    LOG(INFO) << "registering handlers\n";
    curr_node->main_event_loop->subscribe_event(event::event_type::E_NEW_MESSAGE_SENT, self_node_cb::handle_message);
    curr_node->main_event_loop->subscribe_event(event::event_type::E_ACCEPT_FR_REQ, self_node_cb::handle_friend_accepted);

    curr_node->main_event_loop->subscribe_event_resp(event::event_type::E_RESP_GET_FRIEND_NUMBERS_LIST, self_node_cb::handle_get_friend_numbers_list);
    curr_node->main_event_loop->subscribe_event_resp(event::event_type::E_RESP_GET_FRIEND_NAME, self_node_cb::handle_get_friend_name);
    curr_node->main_event_loop->subscribe_event_resp(event::event_type::E_RESP_GET_FRIEND_STATUS_MSG, self_node_cb::handle_get_friend_status_msg);
    curr_node->main_event_loop->subscribe_event_resp(event::event_type::E_RESP_GET_USER_ID, self_node_cb::handle_get_user_id);
}

```

- Register_tox_callbacks() :
 - ❖ Enregistre les callbacks Tox avec l'instance Tox pour gérer les événements tels que les demandes d'amis, les messages, les changements de statut de connexion, etc.

```

void self_node_cb::register_tox_callbacks() {
    Tox* tox = curr_node->tox_c_instance;
    tox_callback_friend_request(tox, self_node_cb::friend_request_cb);
    tox_callback_friend_message(tox, self_node_cb::friend_message_cb);
    tox_callback_self_connection_status(tox, self_node_cb::self_connection_status_cb);
    tox_callback_friend_name(tox, self_node_cb::friend_name_cb);
    tox_callback_friend_status(tox, self_node_cb::friend_status_cb);
    tox_callback_friend_connection_status(tox, self_node_cb::friend_connection_status_cb);
    tox_callback_friend_typing(tox, self_node_cb::friend_typing_cb);
    tox_callback_friend_read_receipt(tox, self_node_cb::friend_read_receipt_cb);
}

```

- Fonctions de Callback Tox :

- friend_request_cb() :
 - ❖ Gère les demandes d'amis entrantes.

```

44 void self_node_cb::friend_request_cb(Tox *tox, const uint8_t *public_key, const uint8_t *message, size_t length,
45                                     void *user_data)
46 {
47     if (curr_node->auto_accept)
48     {
49         LOG(INFO) << "[TOX CALLBACK accepting fr req]\n";
50         tox_friend_add_norequest(tox, public_key, NULL);
51         curr_node->update_savedata_file();
52     }
53     else {
54         char tox_id_hex[TOX_PUBLIC_KEY_SIZE*2 + 1] = {0};
55         sodium_bin2hex(tox_id_hex, sizeof(tox_id_hex), public_key, TOX_PUBLIC_KEY_SIZE);
56         auto x = new std::pair<std::string*, std::string*>(
57             new std::string((char*)message), new std::string(tox_id_hex)
58         );
59         LOG(INFO) << "new friend request from: " << *(x->second)
60             << " message: " << *(x->first) << '\n';
61         SEND_ASYNC_EV(
62             E_NEW_FR_REQ, x
63         );
64     }
}

```

- friend_message_cb() :
 - ❖ Gère les messages entrants des amis.

```

65     self_node_cb::friend_message_cb(Tox *tox, uint32_t friend_number, TOX_MESSAGE_TYPE type, const uint8_t *message,
66                                     size_t length, void *user_data)
67     {
68         event::message_event *e = new event::message_event();
69         e->message = message;
70         e->length = length;
71         e->friend_number = friend_number;
72         e->type = type;
73
74         SEND_ASYNC_EV(E_NEW_MESSAGE_RECV, e)
75         /* e
76         | * event::message_event *e = new event::message_event(); */
77         /* e->message = message; */
78         /* e->length = length; */
79         /* e->friend_number = friend_number; */
80         /* e->type = type; */
81         /* event::async_event ev(event::event_type::E_NEW_MESSAGE_RECV, e); */
82         /* curr_node->main_event_loop->push_event(ev); */
83     }

```

- self_connection_status_cb() :
 - ❖ Gère les changements de statut de connexion de l'instance Tox.

```

84     self_node_cb::self_connection_status_cb(Tox *tox, TOX_CONNECTION connection_status, void *user_data)
85     {
86
87         SEND_ASYNC_EV(E_CONN_STATUS, new std::string(tox_connection_to_string(connection_status)));
88         switch (connection_status) {
89             case TOX_CONNECTION_NONE:
90                 LOG(INFO) << "Offline\n";
91                 break;
92             case TOX_CONNECTION_TCP:
93                 LOG(INFO) << "Online, using TCP\n";
94                 break;
95             case TOX_CONNECTION_UDP:
96                 LOG(INFO) << "Online, using UDP\n";
97                 break;
98         }
99     }

```

- friend_name_cb():
 - ❖ Gère les changements de nom d'un ami.

```

152 void self_node_cb::friend_name_cb(
153     Tox *tox, Tox_Friend_Number friend_number,
154     const uint8_t name[], size_t length, void *user_data) {
155     if (name == NULL) {
156         printf("[TOX CALLBACK] cant do crap name is null\n");
157         return;
158     }
159     auto r = new std::pair<uint32_t, std::string>(friend_number, new std::string((char*)name));
160     SEND_ASYNC_EV(E_FR_CHANGE_NAME, r);
161 }
162

```

- friend_status_cb() :
 - ❖ Gère les changements de statut d'un ami.

```

163 void self_node_cb::friend_status_cb(
164     Tox *tox, Tox_Friend_Number friend_number, Tox_User_Status status, void *user_data) {
165     auto r = new std::pair<uint32_t, Tox_User_Status>(friend_number, status);
166     SEND_ASYNC_EV(E_FR_CHANGE_NAME, r);
167 }

```

- friend_connection_status_cb() :
 - ❖ Gère les changements de statut de connexion d'un ami.

```

170 void self_node_cb::friend_connection_status_cb(
171     Tox *tox, Tox_Friend_Number friend_number, Tox_Connection connection_status, void *user_data) {
172     auto r = new std::pair<uint32_t, Tox_Connection>(friend_number, connection_status);
173     SEND_ASYNC_EV(E_FR_CHANGE_NAME, r);
174 }

```

- friend_typing_cb() :
 - ❖ Gère les notifications de saisie des amis.

```

175 void self_node_cb::friend_typing_cb(
176     Tox *tox, Tox_Friend_Number friend_number, bool typing, void *user_data) {
177     auto r = new std::pair<uint32_t, bool>(friend_number, typing);
178     SEND_ASYNC_EV(E_FR_CHANGE_NAME, r);
179 }

```

- friend_read_receipt_cb() :
 - ❖ Gère les accusés de réception des messages.

```

180 void self_node_cb::friend_read_receipt_cb(
181     Tox *tox, Tox_Friend_Number friend_number, Tox_Friend_Message_Id message_id, void *user_data) {
182     auto r = new std::pair<uint32_t, uint32_t>(friend_number, message_id);
183     SEND_ASYNC_EV(E_FR_CHANGE_NAME, r);
184 }

```

6.2.2.4. Diagramme de séquence

Le diagramme de séquence illustre comment un message est envoyé dans l'application en passant par différents composants :

- Le back-end de l'application web (`web_back_end`) détecte l'envoi d'un message et pousse un événement dans la boucle d'événements (`event_loop`).
- La boucle d'événements s'abonne à cet événement avec un gestionnaire spécifique (`handle_message_sent`).
- Lorsque l'événement est détecté, le gestionnaire de message est appelé avec les données du message.
- Le gestionnaire utilise la bibliothèque Tox (`tox_lib`) pour envoyer le message à l'ami spécifié.

Cela montre le flux asynchrone et la façon dont les événements sont gérés et traités pour envoyer un message dans l'application Tox.



Le diagramme de séquence fournit montré le flux d'interactions pour l'envoi d'un message dans une application utilisant Tox. Les principaux composants impliqués sont `web_back_end`, `event_loop`, `tox_callback`, et `tox_lib`. Voici une explication détaillée du diagramme :

- `web_back_end` : La partie back-end de l'application web, responsable de l'interface utilisateur et de la logique métier côté serveur.
- `event_loop` : La boucle d'événements qui gère les événements asynchrones dans l'application.
- `tox_callback` : La partie du code qui gère les callbacks spécifiques à Tox.
- `tox_lib` : La bibliothèque Tox, fournissant les fonctions pour les opérations de messagerie sécurisée.

Flux d'Interactions :

- `push_event(E_NEW_MESSAGE_SENT, data)` :
 - `web_back_end` envoie un événement `E_NEW_MESSAGE_SENT` avec les données du message à `event_loop`.
 - Cela signifie que le back-end de l'application web a détecté qu'un nouveau message a été envoyé et pousse cet événement dans la boucle d'événements pour traitement.

- subscribe_event(E_NEW_MESSAGE_SENT, handle_message_sent) :
 - event_loop s'abonne à l'événement E_NEW_MESSAGE_SENT et associe le gestionnaire handle_message_sent pour traiter cet événement.
 - Cela signifie que la boucle d'événements sait maintenant quel gestionnaire appeler lorsque cet événement spécifique se produit.
- handle_message_sent(data) :
 - Lorsque l'événement E_NEW_MESSAGE_SENT se produit, event_loop appelle le gestionnaire handle_message_sent avec les données du message.
 - Le gestionnaire handle_message_sent est responsable de traiter le message envoyé.
- tox_friend_send_message(data) :
 - handle_message_sent dans tox_callback appelle la fonction tox_friend_send_message dans tox_lib avec les données du message.
 - Cela signifie que le gestionnaire de message utilise la bibliothèque Tox pour envoyer réellement le message à l'ami spécifié.

6.2.3. Back end web :

❖ Fichier d'implémentation ‘http_server.cpp’ :

Ce fichier configure et lance le serveur HTTP en utilisant la bibliothèque Drogo.

```
proj > src > back_end > http_server.cpp > ...
1 #include "./http_server.hpp"
2 using namespace back_end;
3 event::event_loop *back_end_server::main_event_loop;
4 back_end_server::back_end_server(event::event_loop* event_loop)
5 {
6     // server app() configuration
7     back_end_server::main_event_loop = event_loop;
8     std::printf("main_event_lopp backend server init = %p\n", back_end_server::main_event_loop);
9     drogon::app().loadConfigFile("./config.json");
10 }
11 void back_end_server::main_loop() {
12     drogon::app().run();
13 }
14 std::thread back_end_server::spawn_thread() {
15     return std::thread(back_end_server::main_loop);
16 }
```

1.Initialisation de la boucle d'événements:

Le constructeur de `back_end_server` initialise la boucle d'événements principale et charge la configuration du serveur depuis un fichier JSON.

2.Fonction `main_loop` :

Cette fonction démarre le serveur HTTP en appelant `drogon::app().run()`, qui lance la boucle d'événements de Drogo pour gérer les requêtes entrantes.

3.Fonction `spawn_thread` :

Crée et retourne un thread pour exécuter la fonction ‘`main_loop`’, permettant au serveur de fonctionner de manière asynchrone.

❖ Fichier d'En-tête ‘http_server.hpp’ :

```
src > back_end > http_server.hpp > ...
1  #pragma once
2  #include <drogon/dragon.h>
3  #include <thread>
4  #include "../event_loop/event_loop.hpp"
5  #include <glob.h>
6  namespace back_end {
7    class back_end_server {
8      public:
9        back_end_server(event::event_loop* event_loop);
10       ~back_end_server();
11       std::thread spawn_thread();
12       static event::event_loop* main_event_loop;
13     private:
14       static void main_loop();
15    };
16  };
```

Ce fichier déclare la classe ‘`back_end_server`’, ses constructeurs, destructeurs, et méthodes, ainsi que le pointeur statique `main_event_loop` pour la boucle d'événements. Il inclut également les en-têtes nécessaires pour utiliser Drogo et la gestion des threads.

❖ Fichier d'En-tête ‘web_socket.hpp’ :

```
src > back_end > controller > websocket.hpp > EchoWebsock > handleNewMessage(const WebSocketConnectionPtr &, std::string &&, const WebSocketMessageType &)
1  #pragma once
2  #include <drogon/WebSocketController.h>
3  #include "../event_loop/event_loop.hpp"
4  #include "../http_server.hpp"
5  #include "json_helper.hpp"
6  #include "tox.h"
7  #include "sodium.h"
8  #include <glob.h>
9  using namespace drogon;
10 class EchoWebsock:public drogon::WebSocketController<EchoWebsock>
11 {
12 public:
13   static WebSocketConnectionPtr front_conn;
14   static bool subscribed_event;
15   virtual void handleNewMessage(const WebSocketConnectionPtr&,
16                                 std::string &&,
17                                 const WebSocketMessageType &)override;
18   virtual void handleNewConnection(const HttpRequestPtr &,
19                                   const WebSocketConnectionPtr&)override;
20   virtual void handleConnectionClosed(const WebSocketConnectionPtr&)override;
21
22   static void handle_new_msg(event::async_event e);
23   static void handle_new_friend_request(event::async_event e);
24   static void handle_tox_status(event::async_event e);
25   static void handle_friend_name_change(event::async_event e);
26   WS_PATH_LIST_BEGIN
27   //list path definitions here;
28   WS_PATH_ADD("/echo");
29   WS_PATH_LIST_END
30 };
```

clarification du code :

Variables Membres Statics:

- `WebSocketConnectionPtr front_conn` : Pointeur vers la connexion WebSocket courante.
- `bool subscribed_event` : Indique si les événements ont été abonnés.

Méthodes Principales:

- `handleNewMessage` : Gère les nouveaux messages reçus via WebSocket.
- `handleNewConnection` : Gère les nouvelles connexions WebSocket.
- `handleConnectionClosed` : Gère la fermeture de connexion.

Méthodes Statics de Gestion des Événements:

- `handle_new_msg` : Envoie un nouveau message à l'interface frontale via WebSocket.
- `handle_new_friend_request` : Envoie une nouvelle demande d'ami à l'interface frontale via WebSocket.
- `handle_tox_status` : Envoie le statut de Tox à l'interface frontale via WebSocket.
- `handle_friend_name_change` : Envoie le changement de nom d'ami à l'interface frontale via WebSocket.

❖ Fichier d'Implémentation ‘web_socket.cpp’ :

- **handleNewMessage** :

Parse le message JSON reçu via WebSocket et envoie l'événement approprié à la boucle d'événements backend.

- **handleNewConnection** :

Gère une nouvelle connexion WebSocket et abonne les événements nécessaires.

- **handle_new_msg, handle_new_friend_request, handle_tox_status, handle_friend_name_change** :

Envoie les messages d'événement appropriés à l'interface frontale via WebSocket.

❖ Fichier d'En-tête ‘rest_api.hpp’ :

```
src > back_end > controller > rest_api.hpp > ...
1  #pragma once
2  ✓ #include <drogon/HttpController.h>
3  ✓ #include <json/json.h>
4  ✓ namespace api {
5      using namespace drogon;
6  ✓ class friends: public HttpController<friends> {
7      public:
8          METHOD_LIST_BEGIN
9              ADD_METHOD_TO(friends::get_friends_list, "/get_friends_list", Get);
10             ADD_METHOD_TO(friends::get_tox_id, "/get_user_id", Get);
11             METHOD_LIST_END
12             void get_friends_list(const HttpRequestPtr &req,
13                 std::function<void (const HttpResponsePtr &)> &&callback);
14             void get_tox_id(const HttpRequestPtr &req,
15                 std::function<void (const HttpResponsePtr &)> &&callback);
16         };
17 }
```

Clarification :

- METHOD_LIST_BEGIN, METHOD_LIST_END : Liste des méthodes API enregistrées pour ce contrôleur.
- METHOD_ADD : Ajoute les méthodes API `get_friends_list` et `get_user_id`.

❖ Fichier d'Implémentation ‘rest_api.cpp’ :

• Fonction `get_tox_id` :

```
void friends::get_tox_id(const HttpRequestPtr &req,
    std::function<void (const HttpResponsePtr &)> &&callback) {
    LOG(INFO) << "GOT REQUEST USER TOX ID\n";
    event::sync_event* e = new event::sync_event();
    e->e_type = event::event_type::E_RESP_GET_USER_ID;
    e->is_request = true;
    e->event_payload = nullptr;
    e = back_end::back_end_server::main_event_loop->push_wait(e);
    Json::Value resp;
    resp["id"] = *(std::string*)e->event_payload;
    auto http_resp = HttpResponse::newHttpJsonResponse(resp);
    http_resp->addHeader("Access-Control-Allow-Origin", "*");
    http_resp->addHeader("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE, OPTIONS");
    callback(http_resp);
}
```

cette fonction récupère l'ID Tox de l'utilisateur en envoyant un événement synchrone et retourne un JSON avec cet ID.

- **Fonction get_friends_list :**

Obtenir la liste des amis: Crée un événement synchronisé pour obtenir le nombre d'amis, envoie cet événement à la boucle d'événements et attend la réponse.

Réponse JSON: Formate la réponse en JSON et l'envoie au client via une fonction de rappel (`callback`).

❖ **Bibliothèque Drogo :**

Drogo est une bibliothèque C++ conçue pour le développement de serveurs HTTP et WebSocket performants. Elle offre un modèle de programmation asynchrone, ce qui est essentiel pour construire des applications nécessitant une gestion efficace des E/S, comme les serveurs de messagerie P2P. Drogo permet de créer des API REST et des services WebSocket facilement, en tirant parti des fonctionnalités modernes du langage C++.

Fonctionnalités Clés de Drogo :

1. **Support HTTP et WebSocket:** Drogo permet de gérer des requêtes HTTP et WebSocket de manière transparente, ce qui est crucial pour les applications nécessitant une interaction en temps réel entre le client et le serveur.
2. **Modèle Asynchrone:** Grâce à son architecture asynchrone, Drogo peut gérer un grand nombre de connexions simultanées sans bloquer, ce qui améliore les performances et la réactivité de l'application.
3. **Gestion de la Concurrence:** Drogo utilise des techniques de gestion de la concurrence pour optimiser l'utilisation des ressources du système, garantissant ainsi une haute disponibilité et une faible latence.
4. **Configuration Flexible:** La bibliothèque permet de configurer facilement le serveur à l'aide de fichiers JSON, simplifiant ainsi le déploiement et la gestion des configurations.
5. **Extensibilité:** Drogo est conçu pour être extensible, permettant aux développeurs de créer des contrôleurs personnalisés pour gérer des routes spécifiques et des événements.

❖ **Utilisation de Drogo dans notre Projet:**

Dans notre application de messagerie P2P, Drogo est utilisé pour :

1. Lancer le Serveur HTTP:

Drgo configure et lance le serveur HTTP à partir d'un fichier de configuration JSON, comme montré dans la classe `back_end_server`.

2. Gérer les Connexions WebSocket:

Les connexions WebSocket sont gérées par la classe `EchoWebsock`, qui utilise Drogo pour recevoir et envoyer des messages en temps réel entre le serveur et les clients.

3. Créer des API REST:

La classe `friends` illustre comment Drogo peut être utilisé pour créer des API REST qui permettent aux clients de récupérer des informations, comme la liste des amis, via des requêtes HTTP.

4. Gérer les Événements Asynchrones:

Drgo facilite la gestion des événements asynchrones, comme les nouveaux messages envoyés ou reçus, en intégrant ces événements dans la boucle d'événements principale de l'application.