```cpp
#include "TerminalServer.h"

void _user_login(ClientSocket sock);
void _next_departure(ClientSocket sock, int terminal_id);
void _manage_begin_loading(ClientSocket sock, int terminal_id);
void _manage_end_loading(ClientSocket sock, int terminal_id, s_time loading);
void _manage_leaving(ClientSocket sock, int terminal_id);
void _manage_ask_for_ferry(ClientSocket sock, int terminal_id);
void _manage_docking(ClientSocket sock, int terminal_id);

// Serveur assurant la gestion des terminaux.
void *terminal_server(void* arg)
{
    IniParser properties("terminal_server.ini");

    int port = atoi(properties.get_value("port").c_str());
    int n_clients = atoi(properties.get_value("n_clients").c_str());

    with_server_socket(port, n_clients, _user_login);
    return NULL;
}

// Gère la connexion d'un utilisateur distant sur l'un des terminaux.
// Si les informations de connexion ne sont pas valides, termine la connexion.
void _user_login(ClientSocket sock)
{
    terminal_protocol packet;
    printf("Nouveau client\n");

    sock.receive<terminal_protocol>(&packet);
    if (packet.type == terminal_protocol::LOGIN) {
        try {
            IniParser agents("agents.ini");

            const char* pass = agents.get_value(
                string(packet.content.login.user)
            ).c_str();

            // Vérifie si le terminal n'est pas geré par un autre client
            pthread_mutex_lock(&mutex_connected);
            list<int>::iterator it;
            it = find(
                connected_clients.begin(), connected_clients.end(),
                packet.content.login.terminal_id
            );

            if (it == connected_clients.end()
                && strcmp(pass, packet.content.login.password) == 0) {
                // Connexion réussie
                connected_clients.push_front(packet.content.login.terminal_id);
                pthread_mutex_unlock(&mutex_connected);

                send_flag_packet(sock, terminal_protocol::ACK);
                printf(
                    "Utilisateur connecte en tant que %s sur le terminal %d\n",
                    packet.content.login.user,
                    packet.content.login.terminal_id
                );
                _next_departure(
                    sock,
                    packet.content.login.terminal_id - 1
                );

                // Session terminée, supprime le client
                printf(
                    "Utilisateur deconnecte (%s)\n", packet.content.login.user
                );
                pthread_mutex_lock(&mutex_connected);
                connected_clients.remove(packet.content.login.terminal_id);
                pthread_mutex_unlock(&mutex_connected);
            } else {
                // Terminal occupé par un autre client ou mauvais mot de passe
                pthread_mutex_unlock(&mutex_connected);
                send_flag_packet(sock, terminal_protocol::FAIL);
            }
        } catch (Exception e) { // Utilisateur inexistant
            send_flag_packet(sock, terminal_protocol::FAIL);
        }
    }
    sock.close();
}

// Donne l'heure de départ du ferry depuis le fichier derpartures.ini
s_time _ferry_departure(int ferry_id)
{
    IniParser departures("departures.ini");
    char ferry_id_str[16];
    sprintf(ferry_id_str, "%d", ferry_id);

    return str_to_time(departures.get_value(string(ferry_id_str)).c_str());
}

// Gère la demande du prochain départ d'un ferry.
// Indique si un ferry sort du terminal ce jour, s'il n'y a pas de ferry à ce
// terminal
void _next_departure(ClientSocket sock, int terminal_id)
{
    terminal_protocol packet;
    sock.receive<terminal_protocol>(&packet);

    if (packet.type == terminal_protocol::NEXT_DEPARTURE) {
        if (docked_ferries[terminal_id] != 0) {
            try { // Départ connu
                packet.type = terminal_protocol::DEPARTURE_KNOWN;
                packet.content.departure_known = _ferry_departure(
                    docked_ferries[terminal_id]
                );
                sock.send<terminal_protocol>(&packet);

                // Gère le chargement du ferry
                return _manage_begin_loading(sock, terminal_id);
            } catch (Exception e) { // Pas de départ trouvé
                send_flag_packet(sock, terminal_protocol::DEPARTURE_UNKNOWN);
                return _next_departure(sock, terminal_id); // Se remet en attente
            }
        } else { // Pas de ferry sur ce terminal
            send_flag_packet(sock, terminal_protocol::NO_FERRY);
            return _manage_ask_for_ferry(sock, terminal_id); // Gère l'accostage
        }
    }
}

// Gère la demande de commencement de l'embarquement
void _manage_begin_loading(ClientSocket sock, int terminal_id)
{
    terminal_protocol packet;
    sock.receive<terminal_protocol>(&packet);

    if (packet.type == terminal_protocol::BEGIN_LOADING) {
        // Demande le début du chargement
        s_time departure = _ferry_departure(docked_ferries[terminal_id]);
        s_time loading = packet.content.begin_loading;

        if (time_span(loading, departure) <= 45) {
            // Le chargement ne peut commencer que 45 minutes avant le départ
            send_flag_packet(sock, terminal_protocol::ACK);
            return _manage_end_loading(sock, terminal_id, loading);
        } else {
            send_flag_packet(sock, terminal_protocol::FAIL);
            return _manage_begin_loading(sock, terminal_id);
        }
    }
}

// Gère la notification de fin d'embarquement
void _manage_end_loading(ClientSocket sock, int terminal_id, s_time loading)
{
    terminal_protocol packet;
    sock.receive<terminal_protocol>(&packet);

    if (packet.type == terminal_protocol::END_LOADING) {
        // Confirme la fin du chargement
        s_time end_loading = packet.content.end_loading;

        if (time_span(loading, end_loading) >= 1) { // TODO: 1->15
            // Le chargement dure au minimum 15 minutes
            send_flag_packet(sock, terminal_protocol::ACK);
            return _manage_leaving(sock, terminal_id);
        } else {
            send_flag_packet(sock, terminal_protocol::FAIL);
            return _manage_end_loading(sock, terminal_id, loading);
        }
    }
}

// Gère la notification de départ
void _manage_leaving(ClientSocket sock, int terminal_id)
{
    terminal_protocol packet;
    sock.receive<terminal_protocol>(&packet);

    if (packet.type == terminal_protocol::FERRY_LEAVING) {
        // Ajoute le ferry aux ferry en cours de départ
        pthread_mutex_lock(&mutex_leaving);
        leaving_ferries.push_front(docked_ferries[terminal_id]);
        pthread_mutex_unlock(&mutex_leaving);

        // Supprime le ferry du terminal
        docked_ferries[terminal_id] = 0;

        return _manage_ask_for_ferry(sock, terminal_id); // Terminal vide.
    }
}

// Gère les opérations de dockage d'un ferry lorsque le terminal est vide
void _manage_ask_for_ferry(ClientSocket sock, int terminal_id)
{
    terminal_protocol packet;
    sock.receive<terminal_protocol>(&packet);

    if (packet.type == terminal_protocol::ASK_FOR_FERRY) {
        // Attribue un nouveau ferry au terminal
        pthread_mutex_lock(&mutex_waiting);
        if (waiting_ferries.empty()) { // Pas de ferry en attente
            pthread_mutex_unlock(&mutex_waiting);

            send_flag_packet(sock, terminal_protocol::FAIL);
            return _manage_ask_for_ferry(sock, terminal_id);
        } else { // Un ferry en attente
            int ferry_id = waiting_ferries.front();
            waiting_ferries.pop();
            pthread_mutex_unlock(&mutex_waiting);

            docked_ferries[terminal_id] = ferry_id;

            packet.type = terminal_protocol::FERRY_RESERVED;
            packet.content.ferry_reserved = ferry_id;
            sock.send<terminal_protocol>(&packet);
            return _manage_docking(sock, terminal_id);
        }
    }
}

void _manage_docking(ClientSocket sock, int terminal_id)
{
    terminal_protocol packet;
    sock.receive<terminal_protocol>(&packet);

    if (packet.type == terminal_protocol::FERRY_ARRIVING) {
        int remote_ferry_id = packet.content.ferry_arriving.ferry_id;
        if (docked_ferries[terminal_id] == remote_ferry_id) {
            // Le ferry en attente est bien arrivé au terminale
            send_flag_packet(sock, terminal_protocol::ACK);

            return _next_departure(sock, terminal_id);
        } else {
            // Le ferry n'est pas celui qui a réservé le terminal
            send_flag_packet(sock, terminal_protocol::FAIL);

            return _manage_docking(sock, terminal_id);
        }
    }
}
```