```cpp
#include "server.h"
#include <iostream>
#include "events.h"
#include "gt.hpp"
#include "proton/hash.hpp"
#include "proton/rtparam.hpp"
#include "utils.h"

void server::handle_outgoing() {
    ENetEvent evt;
    while (enet_host_service(m_proxy_server, &evt, 0) > 0) {
        m_gt_peer = evt.peer;

        switch (evt.type) {
            case ENET_EVENT_TYPE_CONNECT: {
                if (!this->connect())
                    return;
            } break;
            case ENET_EVENT_TYPE_RECEIVE: {
                int packet_type = get_packet_type(evt.packet);
                switch (packet_type) {
                    case NET_MESSAGE_GENERIC_TEXT:
                        if (events::out::generictext(utils::get_text(evt.packet))) {
                            enet_packet_destroy(evt.packet);
                            return;
                        }
                        break;
                    case NET_MESSAGE_GAME_MESSAGE:
                        if (events::out::gamemessage(utils::get_text(evt.packet))) {
                            enet_packet_destroy(evt.packet);
                            return;
                        }
                        break;
                    case NET_MESSAGE_GAME_PACKET: {
                        auto packet = utils::get_struct(evt.packet);
                        if (!packet)
                            break;

                        switch (packet->m_type) {
                            case PACKET_STATE:
                                if (events::out::state(packet)) {
                                    enet_packet_destroy(evt.packet);
                                    return;
                                }
                                break;
                            case PACKET_CALL_FUNCTION:
                                if (events::out::variantlist(packet)) {
                                    enet_packet_destroy(evt.packet);
                                    return;
                                }
                                break;

                            case PACKET_PING_REPLY:
                                if (events::out::pingreply(packet)) {
                                    enet_packet_destroy(evt.packet);
                                    return;
                                }
                                break;
                            case PACKET_DISCONNECT:
                            case PACKET_APP_INTEGRITY_FAIL:
                                if (gt::in_game)
                                    return;
                                break;

                            default: PRINTS("gamepacket type: %d\n", packet->m_type);
                        }
                    } break;
                    case NET_MESSAGE_TRACK: //track one should never be used, but its not bad to have it in case.
                    case NET_MESSAGE_CLIENT_LOG_RESPONSE: return;

                    default: PRINTS("Got unknown packet of type %d.\n", packet_type); break;
                }

                if (!m_server_peer || !m_real_server)
                    return;
                enet_peer_send(m_server_peer, 0, evt.packet);
                enet_host_flush(m_real_server);
            } break;
            case ENET_EVENT_TYPE_DISCONNECT: {
                if (gt::in_game)
                    return;
                if (gt::connecting) {
                    this->disconnect(false);
                    gt::connecting = false;
                    return;
                }

            } break;
            default: PRINTS("UNHANDLED\n"); break;
        }
    }
}

void server::handle_incoming() {
    ENetEvent event;

    while (enet_host_service(m_real_server, &event, 0) > 0) {
        switch (event.type) {
            case ENET_EVENT_TYPE_CONNECT: PRINTC("connection event\n"); break;
            case ENET_EVENT_TYPE_DISCONNECT: this->disconnect(true); return;
            case ENET_EVENT_TYPE_RECEIVE: {
                if (event.packet->data) {
                    int packet_type = get_packet_type(event.packet);
                    switch (packet_type) {
                        case NET_MESSAGE_GENERIC_TEXT:
                            if (events::in::generictext(utils::get_text(event.packet))) {
                                enet_packet_destroy(event.packet);
                                return;
                            }
                            break;
                        case NET_MESSAGE_GAME_MESSAGE:
                            if (events::in::gamemessage(utils::get_text(event.packet))) {
                                enet_packet_destroy(event.packet);
                                return;
                            }
                            break;
                        case NET_MESSAGE_GAME_PACKET: {
                            auto packet = utils::get_struct(event.packet);
                            if (!packet)
                                break;

                            switch (packet->m_type) {
                    case 8: {
                        if (!packet->m_int_data) {
                            std::string dice_roll = std::to_string(packet->m_count + 1);
                            gt::send_log("`bThe dice `bwill roll a `#" + dice_roll);
                        }
                    }break;
                                case PACKET_CALL_FUNCTION:
                                    if (events::in::variantlist(packet)) {
                                        enet_packet_destroy(event.packet);
                                        return;
                                    }
                                    break;

                                case PACKET_SEND_MAP_DATA:
                                    if (events::in::sendmapdata(packet)) {
                                        enet_packet_destroy(event.packet);
                                        return;
                                    }
                                    break;

                                case PACKET_STATE:
                                    if (events::in::state(packet)) {
                                        enet_packet_destroy(event.packet);
                                        return;
                                    }
                                    break;
                                //no need to print this for handled packet types such as func call, because we know its 1
                                default: PRINTC("gamepacket type: %d\n", packet->m_type); break;
                            }
                        } break;

                        //ignore tracking packet, and request of client crash log
                        case NET_MESSAGE_TRACK:
                            if (events::in::tracking(utils::get_text(event.packet))) {
                                enet_packet_destroy(event.packet);
                                return;
                            }
                            break;
                        case NET_MESSAGE_CLIENT_LOG_REQUEST: return;

                        default: PRINTS("Got unknown packet of type %d.\n", packet_type); break;
                    }
                }

                if (!m_gt_peer || !m_proxy_server)
                    return;
                enet_peer_send(m_gt_peer, 0, event.packet);
                enet_host_flush(m_proxy_server);

            } break;

            default: PRINTC("UNKNOWN event: %d\n", event.type); break;
        }
    }
}

void server::poll() {
    //outgoing packets going to real server that are intercepted by our proxy server
    this->handle_outgoing();

    if (!m_real_server)
        return;

    //ingoing packets coming to gt client intercepted by our proxy client
    this->handle_incoming();
}

bool server::start() {
    ENetAddress address;
    enet_address_set_host(&address, "0.0.0.0");
    address.port = m_proxyport;
    m_proxy_server = enet_host_create(&address, 1024, 10, 0, 0);
    m_proxy_server->usingNewPacket = false;

    if (!m_proxy_server) {
        PRINTS("failed to start the proxy server!\n");
        return false;
    }
    m_proxy_server->checksum = enet_crc32;
    auto code = enet_host_compress_with_range_coder(m_proxy_server);
    if (code != 0)
        PRINTS("enet host compressing failed\n");
    PRINTS("started the enet server.\n");
    return setup_client();
}

void server::quit() {
    gt::in_game = false;
    this->disconnect(true);
```

```cpp
}

bool server::setup_client() {
    m_real_server = enet_host_create(0, 1, 2, 0, 0);
    m_real_server->usingNewPacket = true;
    if (!m_real_server) {
        PRINTC("failed to start the client\n");
        return false;
    }
    m_real_server->checksum = enet_crc32;
    auto code = enet_host_compress_with_range_coder(m_real_server);
    if (code != 0)
        PRINTC("enet host compressing failed\n");
    enet_host_flush(m_real_server);
    PRINTC("Started enet client\n");
    return true;
}

void server::redirect_server(variantlist_t& varlist) {
    m_port = varlist[1].get_uint32();
    m_token = varlist[2].get_uint32();
    m_user = varlist[3].get_uint32();
    auto str = varlist[4].get_string();

    auto doorid = str.substr(str.find("|"));
    m_server = str.erase(str.find("|")); //remove | and doorid from end
    PRINTC("port: %d token %d user %d server %s doorid %s\n", m_port, m_token,
    m_user, m_server.c_str(), doorid.c_str());
    varlist[1] = m_proxyport;
    varlist[4] = "127.0.0.1" + doorid;

    gt::connecting = true;
    send(true, varlist);
    if (m_real_server) {
        enet_host_destroy(m_real_server);
        m_real_server = nullptr;
    }
}

void server::disconnect(bool reset) {
    m_world.connected = false;
    m_world.local = {};
    m_world.players.clear();
    if (m_server_peer) {
        enet_peer_disconnect(m_server_peer, 0);
        m_server_peer = nullptr;
        enet_host_destroy(m_real_server);
        m_real_server = nullptr;
    }
    if (reset) {
        m_user = 0;
        m_token = 0;
        m_server = "213.179.209.168";
        m_port = 17198;
    }
}

bool server::connect() {
    PRINTS("Connecting to server.\n");
    ENetAddress address;
    enet_address_set_host(&address, m_server.c_str());
    address.port = m_port;
    PRINTS("port is %d and server is %s\n", m_port, m_server.c_str());
    if (!this->setup_client()) {
        PRINTS("Failed to setup client when trying to connect to server!\n");
        return false;
    }
    m_server_peer = enet_host_connect(m_real_server, &address, 2, 0);
    if (!m_server_peer) {
        PRINTS("Failed to connect to real server.\n");
        return false;
    }
    return true;
```

```cpp
}

//bool client: true - sends to growtopia client    false - sends to gt server
void server::send(bool client, int32_t type, uint8_t* data, int32_t len) {
    auto peer = client ? m_gt_peer : m_server_peer;
    auto host = client ? m_proxy_server : m_real_server;

    if (!peer || !host)
        return;
    auto packet = enet_packet_create(0, len + 5, ENET_PACKET_FLAG_RELIABLE);
    auto game_packet = (gametextpacket_t*)packet->data;
    game_packet->m_type = type;
    if (data)
        memcpy(&game_packet->m_data, data, len);

    memset(&game_packet->m_data + len, 0, 1);
    int code = enet_peer_send(peer, 0, packet);
    if (code != 0)
        PRINTS("Error sending packet! code: %d\n", code);
    enet_host_flush(host);
}

//bool client: true - sends to growtopia client    false - sends to gt server
void server::send(bool client, variantlist_t& list, int32_t netid, int32_t delay) {
    auto peer = client ? m_gt_peer : m_server_peer;
    auto host = client ? m_proxy_server : m_real_server;

    if (!peer || !host)
        return;

    uint32_t data_size = 0;
    void* data = list.serialize_to_mem(&data_size, nullptr);

    //optionally we wouldnt allocate this much but i dont want to bother looking into it
    auto update_packet = MALLOC(gameupdatepacket_t, +data_size);
    auto game_packet = MALLOC(gametextpacket_t, +sizeof(gameupdatepacket_t) + data_size);

    if (!game_packet || !update_packet)
        return;

    memset(update_packet, 0, sizeof(gameupdatepacket_t) + data_size);
    memset(game_packet, 0, sizeof(gametextpacket_t) + sizeof(gameupdatepacket_t) + data_size);
    game_packet->m_type = NET_MESSAGE_GAME_PACKET;

    update_packet->m_type = PACKET_CALL_FUNCTION;
    update_packet->m_player_flags = netid;
    update_packet->m_packet_flags |= 8;
    update_packet->m_int_data = delay;
    memcpy(&update_packet->m_data, data, data_size);
    update_packet->m_data_size = data_size;
    memcpy(&game_packet->m_data, update_packet, sizeof(gameupdatepacket_t) + data_size);
    free(update_packet);

    auto packet = enet_packet_create(game_packet, data_size + sizeof(gameupdatepacket_t), ENET_PACKET_FLAG_RELIABLE);
    enet_peer_send(peer, 0, packet);
    enet_host_flush(host);
    free(game_packet);
}

//bool client: true - sends to growtopia client    false - sends to gt server
void server::send(bool client, std::string text, int32_t type) {
    auto peer = client ? m_gt_peer : m_server_peer;
    auto host = client ? m_proxy_server : m_real_server;

    if (!peer || !host)
        return;
    auto packet = enet_packet_create(0, text.length() + 5, ENET_PACKET_FLAG_RELIABLE);
    auto game_packet = (gametextpacket_t*)packet->data;
    game_packet->m_type = type;
    memcpy(&game_packet->m_data, text.c_str(), text.length());

    memset(&game_packet->m_data + text.length(), 0, 1);
    int code = enet_peer_send(peer, 0, packet);
    if (code != 0)
        PRINTS("Error sending packet! code: %d\n", code);
    enet_host_flush(host);
}
```