

```

1  #include "server.h"
2  #include <iostream>
3  #include "events.h"
4  #include "gt.hpp"
5  #include "proton/hash.hpp"
6  #include "proton/rtparam.hpp"
7  #include "utils.h"
8
9  void server::handle_outgoing() {
10     ENetEvent evt;
11     while (enet_host_service(m_proxy_server, &evt, 0) > 0) {
12         m_gt_peer = evt.peer;
13
14         switch (evt.type) {
15             case ENET_EVENT_TYPE_CONNECT: {
16                 if (!this->connect())
17                     return;
18             } break;
19             case ENET_EVENT_TYPE_RECEIVE: {
20                 int packet_type = get_packet_type(evt.packet);
21                 switch (packet_type) {
22                     case NET_MESSAGE_GENERIC_TEXT:
23                         if (events::out::generic_text(utils::get_text(evt.packet))) {
24                             enet_packet_destroy(evt.packet);
25                             return;
26                         }
27                     break;
28                     case NET_MESSAGE_GAME_MESSAGE:
29                         if (events::out::game_message(utils::get_text(evt.packet))) {
30                             enet_packet_destroy(evt.packet);
31                             return;
32                         }
33                     break;
34                     case NET_MESSAGE_GAME_PACKET: {
35                         auto packet = utils::get_struct(evt.packet);
36                         if (!packet)
37                             break;
38
39                         switch (packet->m_type) {
40                             case PACKET_STATE:
41                                 if (events::out::state(packet)) {
42                                     enet_packet_destroy(evt.packet);
43                                     return;
44                                 }
45                             break;
46                             case PACKET_CALL_FUNCTION:
47                                 if (events::out::variant_list(packet)) {
48                                     enet_packet_destroy(evt.packet);
49                                     return;
50                                 }
51                             break;
52
53                             case PACKET_PING_REPLY:
54                                 if (events::out::ping_reply(packet)) {
55                                     enet_packet_destroy(evt.packet);
56                                     return;
57                                 }
58                             break;
59                             case PACKET_DISCONNECT:
60                             case PACKET_APP_INTEGRITY_FAIL:
61                                 if (gt::in_game)
62                                     return;
63                             break;
64
65                             default: PRINTS("gamepacket type: %d\n", packet->m_type);
66                         }
67                     } break;
68                     case NET_MESSAGE_TRACK: //track one should never be used, but
69                                         //its not bad to have it in case.
70                     case NET_MESSAGE_CLIENT_LOG_RESPONSE: return;
71
72                     default: PRINTS("Got unknown packet of type %d.\n",
73                                     packet_type); break;
74                 }
75             }
76         }
77     }
78 }

```

```

72         }
73
74         if (!m_server_peer || !m_real_server)
75             return;
76         enet_peer_send(m_server_peer, 0, evt.packet);
77         enet_host_flush(m_real_server);
78     } break;
79     case ENET_EVENT_TYPE_DISCONNECT: {
80         if (gt::in_game)
81             return;
82         if (gt::connecting) {
83             this->disconnect(false);
84             gt::connecting = false;
85             return;
86         }
87
88     } break;
89     default: PRINTS("UNHANDLED\n"); break;
90 }
91 }
92 }
93
94 void server::handle_incoming() {
95     ENetEvent event;
96
97     while (enet_host_service(m_real_server, &event, 0) > 0) {
98         switch (event.type) {
99             case ENET_EVENT_TYPE_CONNECT: PRINTC("connection event\n"); break;
100            case ENET_EVENT_TYPE_DISCONNECT: this->disconnect(true); return;
101            case ENET_EVENT_TYPE_RECEIVE: {
102                if (event.packet->data) {
103                    int packet_type = get_packet_type(event.packet);
104                    switch (packet_type) {
105                        case NET_MESSAGE_GENERIC_TEXT:
106                            if
107                                (events::in::generictext(utils::get_text(event.packet))) {
108                                enet_packet_destroy(event.packet);
109                                return;
110                            }
111                            break;
112                        case NET_MESSAGE_GAME_MESSAGE:
113                            if
114                                (events::in::gamemessage(utils::get_text(event.packet))) {
115                                enet_packet_destroy(event.packet);
116                                return;
117                            }
118                            break;
119                        case NET_MESSAGE_GAME_PACKET: {
120                            auto packet = utils::get_struct(event.packet);
121                            if (!packet)
122                                break;
123
124                            switch (packet->m_type) {
125                                case 8: {
126                                    if (!packet->m_int_data) {
127                                        std::string dice_roll = std::to_string(packet->m_count +
128                                            1);
129                                        gt::send_log("`bThe dice `bwill roll a `#" + dice_roll);
130                                    }
131                                } break;
132                                case PACKET_CALL_FUNCTION:
133                                    if (events::in::variantlist(packet)) {
134                                        enet_packet_destroy(event.packet);
135                                        return;
136                                    }
137                                    break;
138                                case PACKET_SEND_MAP_DATA:
139                                    if (events::in::sendmapdata(packet)) {
140                                        enet_packet_destroy(event.packet);
141                                        return;
142                                    }
143                                    break;

```

```

142
143         case PACKET_STATE:
144             if (events::in::state(packet)) {
145                 enet_packet_destroy(event.packet);
146                 return;
147             }
148             break;
149         //no need to print this for handled packet types
150         //such as func call, because we know its 1
151         default: PRINTC("gamepacket type: %d\n",
152             packet->m_type); break;
153     }
154     } break;
155
156     //ignore tracking packet, and request of client crash log
157     case NET_MESSAGE_TRACK:
158         if (events::in::tracking(utils::get_text(event.packet))) {
159             enet_packet_destroy(event.packet);
160             return;
161         }
162         break;
163     case NET_MESSAGE_CLIENT_LOG_REQUEST: return;
164
165     default: PRINTS("Got unknown packet of type %d.\n",
166         packet_type); break;
167 }
168
169     if (!m_gt_peer || !m_proxy_server)
170         return;
171     enet_peer_send(m_gt_peer, 0, event.packet);
172     enet_host_flush(m_proxy_server);
173
174     } break;
175
176     default: PRINTC("UNKNOWN event: %d\n", event.type); break;
177 }
178
179 void server::poll() {
180     //outgoing packets going to real server that are intercepted by our proxy server
181     this->handle_outgoing();
182
183     if (!m_real_server)
184         return;
185
186     //ingoing packets coming to gt client intercepted by our proxy client
187     this->handle_incoming();
188 }
189
190 bool server::start() {
191     ENetAddress address;
192     enet_address_set_host(&address, "0.0.0.0");
193     address.port = m_proxyport;
194     m_proxy_server = enet_host_create(&address, 1024, 10, 0, 0);
195     m_proxy_server->usingNewPacket = false;
196
197     if (!m_proxy_server) {
198         PRINTS("failed to start the proxy server!\n");
199         return false;
200     }
201     m_proxy_server->checksum = enet_crc32;
202     auto code = enet_host_compress_with_range_coder(m_proxy_server);
203     if (code != 0)
204         PRINTS("enet host compressing failed\n");
205     PRINTS("started the enet server.\n");
206     return setup_client();
207 }
208
209 void server::quit() {
210     gt::in_game = false;
211     this->disconnect(true);

```

```

212 }
213
214 bool server::setup_client() {
215     m_real_server = enet_host_create(0, 1, 2, 0, 0);
216     m_real_server->usingNewPacket = true;
217     if (!m_real_server) {
218         PRINTC("failed to start the client\n");
219         return false;
220     }
221     m_real_server->checksum = enet_crc32;
222     auto code = enet_host_compress_with_range_coder(m_real_server);
223     if (code != 0)
224         PRINTC("enet host compressing failed\n");
225     enet_host_flush(m_real_server);
226     PRINTC("Started enet client\n");
227     return true;
228 }
229
230 void server::redirect_server(variantlist_t& varlist) {
231     m_port = varlist[1].get_uint32();
232     m_token = varlist[2].get_uint32();
233     m_user = varlist[3].get_uint32();
234     auto str = varlist[4].get_string();
235
236     auto doorid = str.substr(str.find("|"));
237     m_server = str.erase(str.find("|")); //remove | and doorid from end
238     PRINTC("port: %d token %d user %d server %s doorid %s\n", m_port, m_token,
239     m_user, m_server.c_str(), doorid.c_str());
240     varlist[1] = m_proxyport;
241     varlist[4] = "127.0.0.1" + doorid;
242
243     gt::connecting = true;
244     send(true, varlist);
245     if (m_real_server) {
246         enet_host_destroy(m_real_server);
247         m_real_server = nullptr;
248     }
249
250 void server::disconnect(bool reset) {
251     m_world.connected = false;
252     m_world.local = {};
253     m_world.players.clear();
254     if (m_server_peer) {
255         enet_peer_disconnect(m_server_peer, 0);
256         m_server_peer = nullptr;
257         enet_host_destroy(m_real_server);
258         m_real_server = nullptr;
259     }
260     if (reset) {
261         m_user = 0;
262         m_token = 0;
263         m_server = "213.179.209.168";
264         m_port = 17198;
265     }
266 }
267
268 bool server::connect() {
269     PRINTS("Connecting to server.\n");
270     ENetAddress address;
271     enet_address_set_host(&address, m_server.c_str());
272     address.port = m_port;
273     PRINTS("port is %d and server is %s\n", m_port, m_server.c_str());
274     if (!this->setup_client()) {
275         PRINTS("Failed to setup client when trying to connect to server!\n");
276         return false;
277     }
278     m_server_peer = enet_host_connect(m_real_server, &address, 2, 0);
279     if (!m_server_peer) {
280         PRINTS("Failed to connect to real server.\n");
281         return false;
282     }
283     return true;

```

```

284 }
285
286 //bool client: true - sends to growtopia client false - sends to gt server
287 void server::send(bool client, int32_t type, uint8_t* data, int32_t len) {
288     auto peer = client ? m_gt_peer : m_server_peer;
289     auto host = client ? m_proxy_server : m_real_server;
290
291     if (!peer || !host)
292         return;
293     auto packet = enet_packet_create(0, len + 5, ENET_PACKET_FLAG_RELIABLE);
294     auto game_packet = (gametextpacket_t*)packet->data;
295     game_packet->m_type = type;
296     if (data)
297         memcpy(&game_packet->m_data, data, len);
298
299     memset(&game_packet->m_data + len, 0, 1);
300     int code = enet_peer_send(peer, 0, packet);
301     if (code != 0)
302         PRINTS("Error sending packet! code: %d\n", code);
303     enet_host_flush(host);
304 }
305
306 //bool client: true - sends to growtopia client false - sends to gt server
307 void server::send(bool client, variantlist_t& list, int32_t netid, int32_t delay) {
308     auto peer = client ? m_gt_peer : m_server_peer;
309     auto host = client ? m_proxy_server : m_real_server;
310
311     if (!peer || !host)
312         return;
313
314     uint32_t data_size = 0;
315     void* data = list.serialize_to_mem(&data_size, nullptr);
316
317     //optionally we wouldnt allocate this much but i dont want to bother looking
    into it
318     auto update_packet = MALLOC(gameupdatepacket_t, +data_size);
319     auto game_packet = MALLOC(gametextpacket_t, +sizeof(gameupdatepacket_t) +
    data_size);
320
321     if (!game_packet || !update_packet)
322         return;
323
324     memset(update_packet, 0, sizeof(gameupdatepacket_t) + data_size);
325     memset(game_packet, 0, sizeof(gametextpacket_t) + sizeof(gameupdatepacket_t) +
    data_size);
326     game_packet->m_type = NET_MESSAGE_GAME_PACKET;
327
328     update_packet->m_type = PACKET_CALL_FUNCTION;
329     update_packet->m_player_flags = netid;
330     update_packet->m_packet_flags |= 8;
331     update_packet->m_int_data = delay;
332     memcpy(&update_packet->m_data, data, data_size);
333     update_packet->m_data_size = data_size;
334     memcpy(&game_packet->m_data, update_packet, sizeof(gameupdatepacket_t) +
    data_size);
335     free(update_packet);
336
337     auto packet = enet_packet_create(game_packet, data_size +
    sizeof(gameupdatepacket_t), ENET_PACKET_FLAG_RELIABLE);
338     enet_peer_send(peer, 0, packet);
339     enet_host_flush(host);
340     free(game_packet);
341 }
342
343 //bool client: true - sends to growtopia client false - sends to gt server
344 void server::send(bool client, std::string text, int32_t type) {
345     auto peer = client ? m_gt_peer : m_server_peer;
346     auto host = client ? m_proxy_server : m_real_server;
347
348     if (!peer || !host)
349         return;
350     auto packet = enet_packet_create(0, text.length() + 5, ENET_PACKET_FLAG_RELIABLE);
351     auto game_packet = (gametextpacket_t*)packet->data;

```

```
352     game_packet->m_type = type;
353     memcpy(&game_packet->m_data, text.c_str(), text.length());
354
355     memset(&game_packet->m_data + text.length(), 0, 1);
356     int code = enet_peer_send(peer, 0, packet);
357     if (code != 0)
358         PRINTS("Error sending packet! code: %d\n", code);
359     enet_host_flush(host);
360 }
361
```