```cpp
#include "Coordinator.h"
#include "Database.h"

std::mutex mtx;

bool Coordinator::run()
{
    m_userConnectionsThread = new std::thread
      (&Coordinator::userConnectionsThread,this);
    m_serverConnectionsThread = new std::thread
      (&Coordinator::gameServerConnectionsThread, this);

    m_matchmakingThread = new std::thread(&Coordinator::matchmakingThread, this);

    m_userConnectionsThread->join();
    m_serverConnectionsThread->join();

    m_matchmakingThread->join();

    delete m_userConnectionsThread, m_serverConnectionsThread,
      m_matchmakingThread;

    return true;
}

Coordinator::Coordinator(Database* db)
{
    m_db = db;
}

bool Coordinator::init()
{
    WSADATA wsaData;

    // Initialize Winsock
    int iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        printf("WSAStartup failed: %d\n\n", iResult);
        return false;
    }
    return true;
}

bool Coordinator::startServer()
{
    struct addrinfo* result = NULL, * ptr = NULL, hints;

    ZeroMemory(&hints, sizeof(hints));
    hints.ai_family = AF_INET6;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;
    hints.ai_flags = AI_PASSIVE;
```

```
50
51      // Resolve the local address and port to be used by the server
52      int iResult = getaddrinfo(NULL, USER_PORT, &hints, &result);
53      if (iResult != 0) {
54          printf("client getaddrinfo failed: %d\n\n", iResult);
55          WSACleanup();
56          return false;
57      }
58
59      m_UserListenSocket = INVALID_SOCKET;
60
61      // Create a SOCKET for the server to listen for client connections
62
63      m_UserListenSocket = socket(result->ai_family, result->ai_socktype, result-
          >ai_protocol);
64
65      if (m_UserListenSocket == INVALID_SOCKET) {
66          printf("Error at client socket(): %ld\n\n", WSAGetLastError());
67          freeaddrinfo(result);
68          WSACleanup();
69          return false;
70      }
71
72      // Setup the TCP listening socket
73      iResult = bind(m_UserListenSocket, result->ai_addr, (int)result->ai_addrlen);
74      if (iResult == SOCKET_ERROR) {
75          printf("client bind failed with error: %d\n\n", WSAGetLastError());
76          freeaddrinfo(result);
77          closesocket(m_UserListenSocket);
78          WSACleanup();
79          return false;
80      }
81
82      freeaddrinfo(result);
83
84      // Game Server Socket
85
86      // None of the settings stored by the addrinfo object need to be changed
87
88      // Resolve the local address and port to be used by the server
89      iResult = getaddrinfo(NULL, GAMESERVER_PORT, &hints, &result);
90      if (iResult != 0) {
91          printf("getaddrinfo failed: %d\n\n", iResult);
92          WSACleanup();
93          return false;
94      }
95
96      m_GameServerListenSocket = INVALID_SOCKET;
97
98      // Create a SOCKET for the server to listen for client connections
99
100     m_GameServerListenSocket = socket(result->ai_family, result->ai_socktype,
```

```
        result->ai_protocol);

    if (m_GameServerListenSocket == INVALID_SOCKET) {
        printf("Error at socket(): %ld\n\n", WSAGetLastError());
        freeaddrinfo(result);
        WSACleanup();
        return false;
    }

    // Setup the TCP listening socket
    iResult = bind(m_GameServerListenSocket, result->ai_addr, (int)result->ai_addrlen);
    if (iResult == SOCKET_ERROR) {
        printf("bind failed with error: %d\n\n", WSAGetLastError());
        freeaddrinfo(result);
        closesocket(m_GameServerListenSocket);
        WSACleanup();
        return false;
    }

    freeaddrinfo(result);

    return true;
}

bool Coordinator::userConnectionsThread() {
    SOCKET ClientSocket;

    while (true) {
        if (listen(m_UserListenSocket, SOMAXCONN) == SOCKET_ERROR) {
            printf("Listen failed with error: %ld\n\n", WSAGetLastError());
            closesocket(m_UserListenSocket);
            WSACleanup();
            return false;
        }

        // Accept a client socket
        ClientSocket = accept(m_UserListenSocket, NULL, NULL);
        if (ClientSocket == INVALID_SOCKET) {
            printf("accept failed: %d\n\n", WSAGetLastError());
        }

        //m_userThreads.push_back(std::thread(&Coordinator::userThread, this, (LPVOID) ClientSocket));
        std::thread t = std::thread(&Coordinator::userThread, this, (LPVOID) ClientSocket);
        t.detach();
    }

    return true;
}

```

```
149 bool Coordinator::gameServerConnectionsThread() {
150     SOCKET ClientSocket;
151 
152     while (true) {
153         if (listen(m_GameServerListenSocket, SOMAXCONN) == SOCKET_ERROR) {
154             printf("Listen failed with error: %ld\n\n", WSAGetLastError());
155             closesocket(m_GameServerListenSocket);
156             WSACleanup();
157             return false;
158         }
159 
160         // Accept a client socket
161         ClientSocket = accept(m_GameServerListenSocket, NULL, NULL);
162         //if (ClientSocket == INVALID_SOCKET) {
163         //  printf("accept failed: %d\n", WSAGetLastError());
164         //}
165 
166         //m_serverThreads.push_back(std::thread(&Coordinator::gameServerThread,
              this, (LPVOID) ClientSocket));
167         std::thread t = std::thread(&Coordinator::gameServerThread, this,
              (LPVOID)ClientSocket);
168         t.detach();
169     }
170 
171     return true;
172 }
173 
174 bool Coordinator::userThread(LPVOID lParam)
175 {
176     /*
177 
178     1. Recieve login attempts until one is successful
179     2. Send profile information
180     3. Wait for commands
181 
182     */
183 
184     SOCKET userSocket = (SOCKET)lParam;
185     int32_t userId;
186     std::list<COMMAND> tasks;
187 
188     // Used by goto. A goto just seemed a cleaner way of going back here if the
          user changes account.
189     USER_THREAD_START:
190 
191     userId = -1;
192     while (true) {
193         char recvBuff[66];
194 
195         if (!recieveData(userSocket, recvBuff, 66)) {
196             // Connection lost
197             closesocket(userSocket);
```

```
198             return false;
199         }
200
201         std::string logInAttempt(recvBuff);
202
203         std::string username = logInAttempt.substr(1, 32);
204         long long clip = username.find_first_of('#');
205         username = username.substr(0, clip);
206
207         std::string password = logInAttempt.substr(33, 32);
208         clip = password.find_first_of('#');
209         password = password.substr(0, clip);
210
211         // Create Account
212         if (logInAttempt[0] == 'Y') {
213             userId = m_db->addUser(username, password);
214         }
215         // Else log in to existing account
216         else {
217             userId = m_db->logIn(username, password);
218
219             for (auto p : m_connectedPlayers) {
220                 if (userId == p.id) {
221                     userId = -5;
222                     break;
223                 }
224             }
225         }
226
227         //std::string msg("123");
228         //char sendBuff[9];
229         //strcpy_s(sendBuff, _countof(sendBuff), msg.c_str());
230         sendData(userSocket, (char*)&userId, sizeof(userId));
231
232
233         if (userId > 0) {
234             PlayerThread player;
235             player.id = userId;
236             player.threadTasks = &tasks;
237
238             m_connectedPlayers.push_back(player);
239
240             std::cout << "Player " << userId << " has connected, ";
241             break;
242         }
243     }
244
245     // send profile information
246     int32_t data[2] = { 0,0 };
247     m_db->getUserGameInfo(userId, &data[0], &data[1]);
248     std::cout << "they have played " << data[0] << " games and won " << data[1]
        << " of them." << std::endl << std::endl;
```

```
249
250     sendData(userSocket, (char*)data, 8);
251     // -----------------
252
253     // Wait for commands
254
255     fd_set recieveSocket;
256     FD_ZERO(&recieveSocket);
257     timeval waitTime;
258     waitTime.tv_sec = 0;
259     waitTime.tv_usec = 100;
260
261     while (true) {
262
263         FD_SET(userSocket, &recieveSocket);
264         if (select(0, &recieveSocket, NULL, NULL, &waitTime) == SOCKET_ERROR) {
265             printf("Player %i disconnected.\n\n", userId);
266
267             mtx.lock();
268             std::list<Player*>::iterator it;
269             for (it = m_matchmakingQueue.begin(); it != m_matchmakingQueue.end(); ↵
                    it++) {
270                 if ((*it)->id = userId) {
271                     m_matchmakingQueue.erase(it);
272                     break;
273                 }
274             }
275             std::list<PlayerThread>::iterator playerIt = m_connectedPlayers.begin ↵
                   ();
276             while (playerIt != m_connectedPlayers.end()) {
277                 if (playerIt->id == userId) {
278                     m_connectedPlayers.erase(playerIt);
279                     break;
280                 }
281             }
282             mtx.unlock();
283             closesocket(userSocket);
284             return false;
285         }
286         if (FD_ISSET(userSocket, &recieveSocket)) {
287             int32_t int32Buff = 0;
288             if (!recieveData(userSocket, (char*)&int32Buff, 4))
289             {
290                 // Connection lost
291                 printf("Player %i disconnected.\n\n", userId);
292                 mtx.lock();
293                 std::list<Player*>::iterator it;
294                 for (it = m_matchmakingQueue.begin(); it !=                    ↵
                       m_matchmakingQueue.end(); it++) {
295                     if ((*it)->id = userId) {
296                         m_matchmakingQueue.erase(it);
297                         break;
```

```
298                     }
299                 }
300                 std::list<PlayerThread>::iterator playerIt =
                      m_connectedPlayers.begin();
301                 while (playerIt != m_connectedPlayers.end()) {
302                     if (playerIt->id == userId) {
303                         m_connectedPlayers.erase(playerIt);
304                         break;
305                     }
306                 }
307                 mtx.unlock();
308                 closesocket(userSocket);
309 
310                 return false;
311             }
312             // Respond to command
313             switch (int32Buff) {
314             case JOIN_QUEUE:
315             {
316                 mtx.lock();
317                 int32Buff = m_servers.size();
318 
319                 sendData(userSocket, (char*)&int32Buff, sizeof(int32Buff));
320                 in6_addr* addrBuff = new in6_addr[int32Buff];
321                 for (auto server : m_servers) {
322                     memcpy(addrBuff, server->ip, sizeof(in6_addr));
323                     addrBuff++;
324                 }
325                 mtx.unlock();
326 
327                 addrBuff -= int32Buff;
328                 sendData(userSocket, (char*)addrBuff, int32Buff * sizeof
                      (in6_addr));
329 
330                 if (int32Buff == 0) {
331                     std::cout << "Player " << userId << " requested to join the
                          matchmaking queue but there are no servers available.\n\n";
332                     continue;
333                 }
334 
335                 int64_t* pingBuff = new int64_t[int32Buff];
336                 recieveData(userSocket, (char*)pingBuff, int32Buff * sizeof
                      (pingBuff[0]));
337 
338                 bool usableServers = false;
339                 for (int i = 0; i < int32Buff; i++) {
340                     if (pingBuff[i] >= 0) {
341                         usableServers = true;
342                     }
343                 }
344 
345                 if (!usableServers) {
```

```
346                     std::cout << "Player " << userId << " requested to join the
                         matchmaking queue but the player does not have a suitable
                         connection to any of the servers.\n\n";
347                     continue;
348                 }
349
350                 Player p;
351                 p.id = userId;
352                 p.playerSocket = &userSocket;
353                 p.threadTasks = &tasks;
354                 std::cout << "Player " << userId << " requested to join the
                      matchmaking queue:" << std::endl;
355
356                 // Insertion sort
357                 for (int i = 1; i < int32Buff; i++) {
358                     int64_t tempPing = pingBuff[i];
359                     in6_addr tempIp = addrBuff[i];
360
361                     int index = i;
362                     while (index > 0 and tempPing < pingBuff[index - 1]) {
363                         pingBuff[index] = pingBuff[index - 1];
364                         addrBuff[index] = addrBuff[index - 1];
365                         index--;
366                     }
367                     pingBuff[index] = tempPing;
368                     addrBuff[index] = tempIp;
369                 }
370
371                 for (int a = 0; a < int32Buff; a++) {
372                     p.addConnection(addrBuff[a], pingBuff[a]);
373                     char str[64];
374                     inet_ntop(AF_INET6, addrBuff + a, str, 64);
375                     std::cout << str << ":\t" << pingBuff[a] << "\n";
376                 }
377
378                 delete[] pingBuff, addrBuff;
379
380                 mtx.lock();
381                 m_matchmakingQueue.push_back(&p);
382                 mtx.unlock();
383
384                 std::cout << std::endl;
385                 break;
386             }
387             case LEAVE_QUEUE:
388             {               std::cout << "Player " << userId << " requested to
                  leave the matchmaking queue." << std::endl << std::endl;
389
390             bool playerFound = false;
391             mtx.lock();
392             std::list<Player*>::iterator it;
393             for (it = m_matchmakingQueue.begin(); it != m_matchmakingQueue.end();
```

```
                it++) {
394                 if ((*it)->id = userId) {
395                     m_matchmakingQueue.erase(it);
396                     playerFound = true;
397                     break;
398                 }
399             }
400             mtx.unlock();
401             if (playerFound) {
402                 // Player Successfully left the queue
403                 int32_t buff = LEFT_QUEUE;
404                 sendData(userSocket, (char*)&buff, 4);
405 
406                 printf("Player %i left the queue.\n\n", userId);
407             }
408             break;
409             }
410             case SWITCH_ACCOUNT:
411             {
412                 printf("Player %i has logged out.\n\n", userId);
413                 std::list<PlayerThread>::iterator playerIt =
                      m_connectedPlayers.begin();
414                 while (playerIt != m_connectedPlayers.end()) {
415                     if (playerIt->id == userId) {
416                         m_connectedPlayers.erase(playerIt);
417                         break;
418                     }
419                 }
420                 goto USER_THREAD_START;
421             }
422             }
423         }
424 
425         // Carry out any tasks required by other threads.
426         //mtx.lock();
427         while (tasks.size() > 0) {
428             switch (tasks.front().type) {
429             case USER_NEWGAME:
430             {
431                 int32_t buff = GAME_FOUND;
432                 sendData(userSocket, (char*)&buff, 4);
433 
434                 sendData(userSocket, (char*)tasks.front().data, sizeof(*
                      (IN6_ADDR*)tasks.front().data));
435                 tasks.pop_front();
436                 break;
437             }
438             case USER_STATS:
439             {
440                 // send profile information
441                 int32_t data[2] = { 0,0 };
442                 m_db->getUserGameInfo(userId, &data[0], &data[1]);
```

```
443
444                 sendData(userSocket, (char*)data, 8);
445                 tasks.pop_front();
446                 break;
447             }
448             }
449         }
450         //mtx.unlock();
451     }
452
453     std::list<PlayerThread>::iterator playerIt = m_connectedPlayers.begin();
454     while (playerIt != m_connectedPlayers.end()) {
455         if (playerIt->id == userId) {
456             m_connectedPlayers.erase(playerIt);
457             break;
458         }
459     }
460     closesocket(userSocket);
461     return false;
462 }
463
464 bool Coordinator::gameServerThread(LPVOID lParam)
465 {
466     SOCKET serverSocket = (SOCKET)lParam;
467
468     std::list<COMMAND> tasks;
469
470     sockaddr_in6 s;
471     int nameSize = sizeof(s);
472     getpeername(serverSocket, (sockaddr*)&s, &nameSize);
473
474     char serverAddrBuff[64];
475     inet_ntop(AF_INET6, &s.sin6_addr, serverAddrBuff, 64);
476     std::cout << "New server connected: " << serverAddrBuff << std::endl <<
        std::endl;
477
478     Server server;
479     server.ip = &s.sin6_addr;
480     server.socket = &serverSocket;
481     server.threadTasks = &tasks;
482
483     mtx.lock();
484     m_servers.push_back(&server);
485     mtx.unlock();
486
487     fd_set recieveSocket;
488     FD_ZERO(&recieveSocket);
489     timeval waitTime;
490     waitTime.tv_sec = 0;
491     waitTime.tv_usec = 100;
492
493     while (true) {
```

```
494
495         FD_SET(serverSocket, &recieveSocket);
496
497         if (select(0, &recieveSocket, NULL, NULL, &waitTime) == SOCKET_ERROR) {
498             closesocket(serverSocket);
499             std::cout << "Connection lost" << std::endl << std::endl;
500
501             mtx.lock();
502             std::list<Server*>::iterator it;
503             for (it = m_servers.begin(); it != m_servers.end(); it++) {
504                 if ((*it)->ip == &s.sin6_addr) {
505                     m_servers.erase(it);
506                     break;
507                 }
508             }
509             mtx.unlock();
510             return false;
511         }
512
513         if (FD_ISSET(serverSocket, &recieveSocket)) {
514             int32_t int32Buff;
515             if (!recieveData(serverSocket, (char*)&int32Buff, 4))
516             {
517                 std::cout << "Connection lost" << std::endl << std::endl;
518                 closesocket(serverSocket);
519
520                 mtx.lock();
521                 std::list<Server*>::iterator it;
522                 for (it = m_servers.begin(); it != m_servers.end(); it++) {
523                     if ((*it)->ip == &s.sin6_addr) {
524                         m_servers.erase(it);
525                         break;
526                     }
527                 }
528                 mtx.unlock();
529                 return false;
530             }
531             switch (int32Buff) {
532             case GAME_FINISHED:
533             {
534                 std::cout << "A game has finished successfully." << std::endl <<
                      std::endl;
535
536                 GameInfo game;
537
538                 time_t now = time(0);
539                 tm ltm;
540                 localtime_s(&ltm, &now);
541                 // Date is in 'DD/MM/YYYY' format plus a NULL character at the
                      end
542                 const char* format = "%02d/%02d/%04d";
543                 sprintf_s(game.date, format, ltm.tm_mday, ltm.tm_mon + 1,
```

```
                    ltm.tm_year + 1900);

                // Game Duration
                game.duration = 0;
                recieveData(serverSocket, (char*)&game.duration, 4);

                // Number of Participating Teams
                game.numberOfTeams = 0;
                recieveData(serverSocket, (char*)&game.numberOfTeams, 4);

                // Team Scores
                game.scores = new int32_t[game.numberOfTeams];
                recieveData(serverSocket, (char*)game.scores, game.numberOfTeams
                  * 4);

                // Number of participants in each team
                game.numbersOfParticipants = new int32_t[game.numberOfTeams];
                recieveData(serverSocket, (char*)game.numbersOfParticipants,
                  game.numberOfTeams * 4);

                // Participant IDs
                game.participants = new int32_t*[game.numberOfTeams];
                for (int t = 0; t < game.numberOfTeams; t++) {
                    game.participants[t] = new int32_t[game.numbersOfParticipants
                     [t]];
                    recieveData(serverSocket, (char*)game.participants[t],
                     game.numbersOfParticipants[t] * 4);
                }

                m_db->addGame(game);

                // Tell user threads to resend user stats
                for (int t = 0; t < game.numberOfTeams; t++) {
                    for (int p = 0; p < game.numbersOfParticipants[t]; p++) {
                        for (auto connectedPlayer : m_connectedPlayers) {
                            if (connectedPlayer.id == game.participants[t][p]) {
                                COMMAND c;
                                c.type = USER_STATS;
                                connectedPlayer.threadTasks->push_back(c);
                            }
                        }
                    }
                }

                delete[] game.scores;
                for (int t = 0; t < game.numberOfTeams; t++) {
                    delete[] game.participants[t];
                }
                delete[] game.participants;
            }
            }
        }
```

```
591         //mtx.lock();
592         while (tasks.size() > 0) {
593             switch (tasks.front().type) {
594             case SERVER_NEWGAME:
595             {
596                 int32_t buffer;
597                 buffer = START_GAME;
598                 sendData(serverSocket, (char*)&buffer, sizeof(buffer));
599 
600                 buffer = reinterpret_cast<GAME*>(tasks.front().data)->numberOfPlayers;
601                 sendData(serverSocket, (char*)&buffer, sizeof(buffer));
602 
603                 int32_t numberOfTeams = reinterpret_cast<GAME*>(tasks.front().data)->numberOfTeams;
604                 sendData(serverSocket, (char*)&numberOfTeams, sizeof(int32_t));
605 
606                 sendData(serverSocket, (char*)(reinterpret_cast<GAME*>(tasks.front().data)->userIds), sizeof(int32_t)*reinterpret_cast<GAME*>(tasks.front().data)->numberOfPlayers);
607 
608                 sendData(serverSocket, (char*)(reinterpret_cast<GAME*>(tasks.front().data)->teams), sizeof(int32_t)*reinterpret_cast<GAME*>(tasks.front().data)->numberOfPlayers);
609                 for (int i = 0; i < buffer; i++) {
610                     std::cout << reinterpret_cast<GAME*>(tasks.front().data)->userIds[i] << '\t';
611                     std::cout << reinterpret_cast<GAME*>(tasks.front().data)->teams[i] << std::endl;
612                 }
613 
614                 tasks.pop_front();
615                 break;
616             }
617             }
618         }
619         //mtx.unlock();
620     }
621     mtx.lock();
622     std::list<Server*>::iterator it;
623     for (it = m_servers.begin(); it != m_servers.end(); it++) {
624         if ((*it)->ip == &s.sin6_addr) {
625             m_servers.erase(it);
626             break;
627         }
628     }
629     mtx.unlock();
630     return true;
631 }
632 
633 bool Coordinator::matchmakingThread()
634 {
```

```
635     std::list<Player*>::iterator playerA, playerB;
636     int maxPing = 120;
637     bool foundGame;
638     mtx.lock();
639     while (true) {
640
641         mtx.unlock();
642         Sleep(2);
643         mtx.lock();
644
645         /*
646         Two players A and B.
647         Initally player A is first in the queue and player B is second.
648         Whilst keeping player A the same, go through each player in the queue and
               see if a game can be made between the two.
649         If none can be made, the second player in the queue becomes player A and
              the process repeats.
650         */
651         playerA = m_matchmakingQueue.begin();
652         playerB = std::next(m_matchmakingQueue.begin(), 1);
653         foundGame = false;
654         while (playerA != std::prev(m_matchmakingQueue.end(),1)){
655             while (playerB != m_matchmakingQueue.end()){
656                 // Try to match players in a game.
657
658                 // For now, all that is needed a server which they both have a
                      low ping with.
659                 for (int s1 = 0; s1 < (*playerA)->pings.size(); s1++) {
660                     if ((*playerA)->pings[s1].ping < 0 || (*playerA)->pings
                          [s1].ping > 120) continue;
661                     for (int s2 = 0; s2 < (*playerB)->pings.size(); s2++) {
662                         bool serverIsSame = true;
663                         for (int w = 0; w < 8; w++) {
664                             if ((*playerA)->pings[s1].serverIP.u.Word[w] !=
                          (*playerB)->pings[s2].serverIP.u.Word[w]) {
665                                 serverIsSame = false;
666                                 break;
667                             }
668                         }
669                         if (serverIsSame) {
670                             if ((*playerB)->pings[s2].ping < 0 || (*playerB)-
                          >pings[s2].ping > 120) {
671                                 break;
672                             }
673                             else {
674                                 // Create game
675                                 // First request server to host game
676                                 // Find server object
677                                 bool serverFound = false;
678                                 std::list<Server*>::iterator serverIt;
679                                 for (serverIt = m_servers.begin(); serverIt !=
```

```
                    m_servers.end(); serverIt++ ) {
680                             bool serverIsSame = true;
681                             for (int w = 0; w < 8; w++) {
682                                 if ((*playerA)->pings[s1].serverIP.u.Word ↵
                    [w] != (*serverIt)->ip->u.Word[w]) {
683                                     serverIsSame = false;
684                                     break;
685                                 }
686                             }
687                             if (serverIsSame) {
688                                 serverFound = true;
689                                 break;
690                             }
691                         }
692 
693                         if (!serverFound) {
694                             // Server must have been disconnected, don't ↵
                    try to use it again.
695                             (*playerA)->pings[s1].ping = -1;
696                             (*playerB)->pings[s2].ping = -1;
697                             continue;
698                         }
699 
700                         // Notify server about the game
701                         COMMAND c;
702 
703                         GAME game;
704                         game.numberOfPlayers = 2;
705                         game.numberOfTeams = 2;
706                         game.userIds = new int32_t[2];
707                         game.userIds[0] = (*playerA)->id;
708                         game.userIds[1] = (*playerB)->id;
709                         game.teams = new int32_t[2];
710                         game.teams[0] = 0;
711                         game.teams[1] = 1;
712                         c.type = SERVER_NEWGAME;
713                         c.data = &game;
714 
715                         (*serverIt)->threadTasks->push_back(c);
716 
717                         // Notify players they have been found a game.    ↵

718                         c.type = USER_NEWGAME;
719                         c.data = (*serverIt)->ip;
720                         (*playerA)->threadTasks->push_back(c);
721                         (*playerB)->threadTasks->push_back(c);
722 
723                         char ipBuff[64];
724                         inet_ntop(AF_INET6, (*serverIt)->ip, ipBuff, 64);
725                         printf("A game was created between players %i and ↵
                     %i\non server %s.\n", (*playerA)->id, (*playerB)->id,       ↵
                    ipBuff);
```

```
726
727                                 // Remove players from matchmaking queue
728                                 m_matchmakingQueue.erase(playerA++);
729                                 m_matchmakingQueue.erase(playerB++);
730                                 foundGame = true;
731                             }
732                         }
733                         if (foundGame) break;
734                     }
735                     if (foundGame) break;
736                 }
737                 if (foundGame) break;
738                 ++playerB;
739             }
740             if (foundGame) break;
741             ++playerA;
742         }
743     }
744     return false;
745 }
746
```