A Chess Client-Server Model

Călin Irina, second year, group E2

Alexandru Ioan Cuza University of Iași, Faculty of Computer Science

Abstract. Networking has become the main selling point for computer games: commercial games are expected to support multiplaying and the online game sites aim at supporting an ever increasing number of users. At the same time, new game console releases rely heavily on the appeal of online gaming, and a whole new branch of mobile entertainment has emerged with intention to develop distributed multiplayer games for wireless applications.[1] This report focuses on the latest research done on networked computer games and presents the concepts of a basic Chess game, exploring at the same time different aspects of online multiplayer games.

1 Introduction

Online games are ubiquitous on modern gaming platforms, including PCs, consoles and mobile devices, and span many genres. The design of online games can range from simple text-based environments to the incorporation of complex graphics and virtual worlds.[3] The chosen topic is a Chess Client-Server model, centered especially on the Client part, providing also an interface for the user.

The existence of online components within the game can range from being minor features, such as an online leaderboard, to being part of core gameplay, such as directly playing against other players.[3]

Any multiplayer computer game requires both consistent and responsive networking. Consistency is important for maintaining a similar set of data for all players, whereas responsiveness requires that updates to the data are done as promptly as possible. These two requirements, however, are often contradictory and solving this consistency–responsiveness dichotomy lies in the heart of real-time interactive networking.[1]

We may assume the game can have a great amount of simultaneous players from all of over the world, which means that the scalability of the chosen network architecture becomes critical. Moreover, a massive multiplayer game often requires maintaining a persistent game world, where the game progresses around the clock regardless the involvement of a player [1] (taking into account the possibility of implementing a leaderboard, for instance).

2 Used technologies

In order to keep the program reliable and robust, **the TCP/IP has to be used for the communication part**, since it is of great importance to have all the

signals and messages sent from the client to the server and vice-versa correctly and in the right order (which requires a connection-oriented socket).

However, for the I/O data flow we shall use the **BSD socket API**, as it is easy to use and stable and can be used for communication among remote hosts.

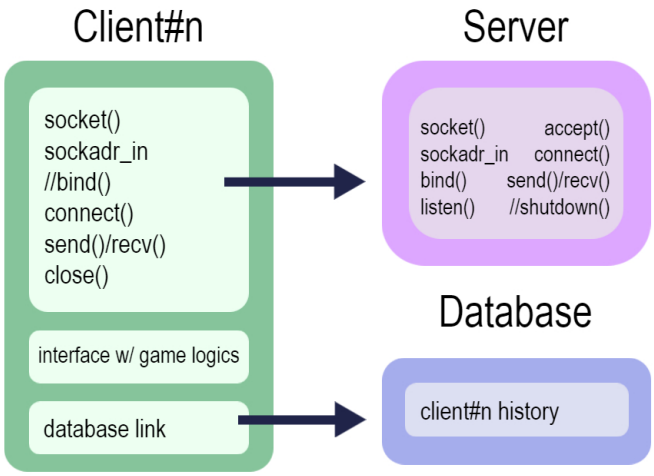
Another element which should improve the Client part is creating a database in order to keep track of all the matches ever played (e.g. a history page). This will be done by using the **SQLite library**, because it is easy to use and provides more than enough features for this case.

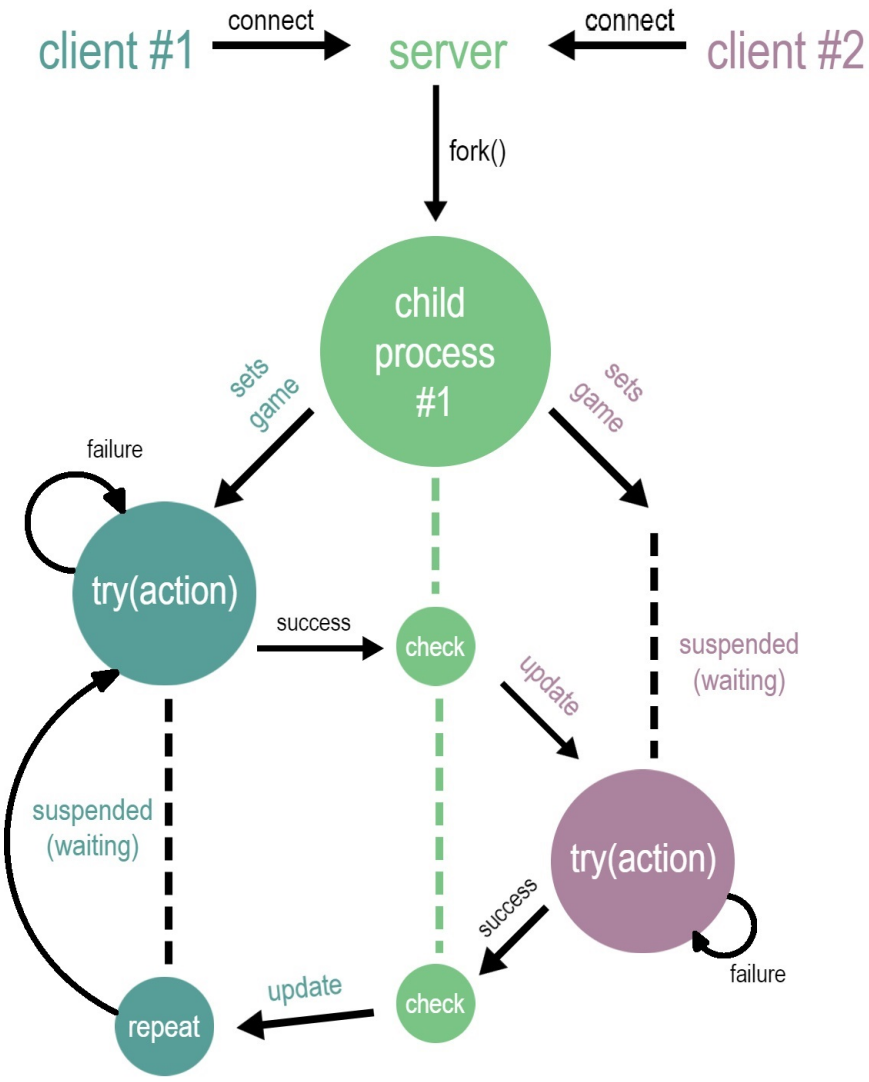
And last, but not least, for the interface part we will use Qt, which is a cross-platform application development framework for desktop, embedded and mobile and which comes with its own IDE - **Qt Creator**. One more aspect to be taken into account is that with Qt, GUIs can be written directly in C++ using its Widgets module and it also comes with an interactive graphical tool called Qt Designer which functions as a code generator for Widgets based GUIs.[7]

3 Application architecture

As mentioned above, the Chess Client should be designed with respect to the TCP/IP protocol, which results in the following ideas (in theory):

- The mechanism should be **synchronous**, since we are interested in the client not being able to take action unless it is their turn
- The client creates a socket, prepares the data structures - sockaddr_in, connects to the server, enters a loop containing send/receive until the connection is closed (the server sends a signal when the game is over)
- This means that, once the player has a valid movement, the client passes to the server, for instance, the new coordinates for a certain piece. Now the server processes that, updates the board, sends the second player a signal and waits for him to take action.





4 Implementation details

The client contains the functions required in order to connect to the server using the TCP/IP and also includes a header file "chess.h" and a cpp which define the game logics. The SQLite library is also binded. We are interested also in having the client and the server on diferent machines, and so we will use the AF_INET family, but **we have to consider both IPv4 and IPv6 for this**, which is why we should treat both scenarios in the first place. Actions can be taken by

the client only in case of receiving a specific signal (otherwise, the player could move a piece no matter whose turn it is and the game would crash).

Note: **getaddrinfo()** returns one or more addrinfo structures, each of which contains an Internet address that can be specified in a call to bind() or connect(). The getaddrinfo() function combines the functionality provided by the gethostbyname() and getservbyname() functions into a single interface, **but** unlike the latter functions, **getaddrinfo()** is reentrant and **allows programs to eliminate IPv4-versus-IPv6 dependencies.** [9]

Note(2): The getaddrinfo function: int getaddrinfo(const char *node, const char *service, const struct addrinfo *hints, struct addrinfo **res);

The hints argument points to an addrinfo structure that specifies criteria for selecting the socket address structures returned in the list pointed to by res. If hints is not NULL it points to an addrinfo structure whose ai_family, ai_socktype, and ai_protocol specify criteria that limit the set of socket addresses returned by getaddrinfo().

Note(3): Part of the following functions has been adapted from the Linux Manual [9].

```
//in ChessClient.cpp:
//get sockaddr, IPv4 or IPv6:
void *get_in_addr(struct sockaddr *sa)
{
    if (sa->sa_family == AF_INET)
    {
        return &(((struct sockaddr_in *)sa)->sin_addr);
    }
    return &(((struct sockaddr_in6 *)sa)->sin6_addr);
}
```

```
//setting the structure
    struct addrinfo hints;
    int r;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC; //The value AF_UNSPEC indicates that
                 //getaddrinfo() should return socket addresses for any
                 //address family (either IPv4 or IPv6 in this case)
                 //equivalent to AF_INET/AF_INET6
    hints.ai_socktype = SOCK_STREAM; //TCP
    if ((r = getaddrinfo(argv[1], PORT, &hints, &servinfo)) != 0)
    {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
        return 1;
    }
```

```
//loop through all the results and connect to the first possibility
    struct addrinfo *servinfo, *prot;
```

```
int sockfd;
for (prot = servinfo; prot != NULL; prot = prot->ai_next)
{
    if ((sockfd = socket(prot->ai_family, prot->ai_socktype,
        prot->ai_protocol)) == -1) //ai_protocol field specifies the
              //protocol for the returned socket
              //addresses. Specifying 0 in this field indicates that
              //socket addresses with any protocol can be returned
              //by getaddrinfo().
    {
        perror("client: socket");
        continue;
    }
    if (connect(sockfd, prot->ai_addr, prot->ai_addrlen) == -1)
    {
        perror("client: connect");
        close(sockfd);
        continue;
    }
    break;
}
if (prot == NULL)
{
    fprintf(stderr, "client: failed to connect\n");
    return;
}
//to test our connection
inet_ntop(prot->ai_family, get_in_addr((struct sockaddr
    *)prot->ai_addr), s, sizeof s); //convert IPv4 and IPv6
    addresses from binary to text form
printf("client: connecting to %s\n", s);
freeaddrinfo(servinfo); // all done with this structure
```

```
//taking actions according to the server's signals
    while(signal)
    {
        if ((bytes = recv(sockfd, &signal, MAXDATASIZE - 1, 0)) == -1)
            //receiving the signal which tells me if it's my turn
        {
            perror("recv");
        }
        if(signal==1) //if yes
        {
            //after getting the action code from the player
            if ((bytes = send(sockfd, &code, MAXDATASIZE - 1, 0)) == -1)
                //send the action code to the server
            {
                perror("send");
            }
```

```
            signal=2; //wait until it's my turn again
        }
       if ((bytes = recv(sockfd, (Board)* b, MAXDATASIZE - 1, 0)) == -1)
            //receive the updated board from the server
       {
             perror("recv");
       }
   }
```

5 Conclusions

Although the very basic Chess client-server model doesn't require a very complex architecture (it is basically a ping-pong model), the number of improvements regarding the efficiency and the available options is pretty solid.

An extention would consist of using the RPC, since the way this protocol works is that a client can create a request (for a certain movement, for instance) in the form of a procedure, function or method call to a remote server, which RPC translates and sends. When the remote server receives the request, it sends a response back to the client and the application continues its process. We are interested in RPC in particular, because an RPC is a synchronous operation requiring the requesting program to be suspended until the results of the remote procedure are returned (a characteristic which might prove being useful if we take into account the fact that, during a match, we want the players to be able to take action only when it is their turn). We could use rpclib for all this. (Note: msgpack-RPC is the protocol that rpclib uses for dispatching and encoding calls to functions. The protocol is based on msgpack, a fast and compact format.)

Other improvements to the game may include a login system, an internal chat, the possibility of choosing the opponent based on a certain username and so on, but these require additional modifications to the server part too.

References

1. Jouni Smed: Networking for Computer Games. International Journal of Computer Games Technology, vol. 2008. Article ID 928712, 1 page, 2008
2. Larry Peterson, Bruce Davie: Computer Networks: A Systems Approach, Elsevier, 2012
3. Online Games Reference, https://en.wikipedia.org/wiki/Online_game.
4. BSD Sockets Reference, http://wiki.treck.com/Introduction_to_BSD_Sockets.
5. Linux Manual - RPC Section, http://man7.org/linux/man-pages/man3/rpc.3.html.
6. Alboaie Lenuța, Panu Andrei: RPC Paradigm, UAIC Computer Networks Course, https://profs.info.uaic.ro/ computernetworks/files/11rc_ParadigmaRPC_En.pdf .
7. Qt About Page, https://wiki.qt.io/About_Qt.
8. Linux Manual, http://man7.org/linux/man-pages/man3/getaddrinfo.3.html.
9. Linux Manual, http://man7.org/linux/man-pages/man3/inet_ntop.3.html.

10. Alboaie Lenuța, Panu Andrei: Network Programming, UAIC Computer Networks Course, https://profs.info.uaic.ro/ computernetworks/-files/5rc_ProgramareaInReteaI_en.pdf.
11. Remote Procedure Calls, Linux Journal, https://www.linuxjournal.com/article/2204.
12. Inet Client Model, https://www.potaroo.net/ispcol/2011-12/client.c.