Casino Blackjack Protocol (CBP) Requirements Document

Stephen Hansen
June 4, 2021
CS 544

Contents

1 STATEFUL

Both the client and server validate the state of the DFA. This is mainly useful for the server, to prevent PDUs that are invalid at certain states from triggering unexpected events (such as adjusting an account balance prior to authenticating). The client tracks the state just to ensure that invalid commands are not sent at the wrong states, but ultimately you could write a client that doesn't track the state and it would still work fine with the server. The DFA states are represented as an enum in `src/protocol/dfa.h`.

The server state validation is present in the main method in `src/server/server.cpp`. There is a global map `conn_to_state` which maps SSL TCP connections to states, so that the server can track the states of multiple connected clients. Validation is done in a loop in main, where the server first parses a PDU from the client via the `parse_pdu_server` method. The PDU is wrapped in a class and the server then uses a large if/else chain based on the current state with the client. Inside the relevant state branch, the server does a dynamic type check to figure out what PDU is sent. If the PDU is invalid for the current state, the server responds with a 5-series error response, otherwise the PDU is handled by the server. State transitions are handled here and are also handled in `run_blackjack` as part of the TableDetails class in `src/server/server.h`. This method runs in a separate thread so that each blackjack game has its own thread. This design did present some concurrency issues, so as a result the PlayerInfo class (also in `src/server/server.h`) wraps the SSL connection and uses a mutex lock to update the player's state. A Boolean flag can disable the thread's ability to alter a given player's state (through the `disconnect` method), so the DFA and this mutex lock protects state transitions from being accidentally overwritten.

The client validates the state at the end of the main method in `src/client/client.cpp`. By the time the user is authenticated, the client program enters a loop on reading player inputs and translating commands to PDUs. The state is checked prior to relevant commands to ensure that the command is valid at the current state. State transitions are handled in `src/client/client.h` in the method `handle_state_transition`. The `listen_to_server` method reads off a PDU from the SSL TCP connection, calls `parse_pdu_client` to package the PDU in a class, passes the header information to `handle_state_transition` to make the appropriate state transition, and then does a dynamic type check to determine what PDU was received and how it should be handled. The three-byte reply code is parsed to determine the state to transition to, thus all state transitions are handled by the client on reads from the server. This method runs in a separate thread from the client's main method (to ensure that the client can send messages while receiving PDUs from the server). Since all state transitions (except the authentication sequence, done in the main method) are done in the read thread, there are no issues with concurrency for the global state. The client is less strict than the server when it comes to validating responses - any responses that do not have explicit transitions cause `handle_state_transition` to be a no-op, meaning the state does not change, but there is no checking of any invalid response codes at any given state (in case of a new protocol version, which adds a new response code but is still compatible with an older client version).

2 CONCURRENT

The server can handle multiple clients at the same time. In `src/server/server.cpp`, the main method loops forever on accepting connections. These socket connections are then passed to the method `connection_handler` which is run in a detached thread. There is one thread per socket client connected to the server. When the client disconnects from the server, the server fails to parse a PDU, and the loop on the server read from the client is broken. The SSL connection is removed from the current connected game (if it exists), and the connection is freed before the connection handler thread terminates. There is also concurrency in that every blackjack game maintains a separate game thread (the thread runs `run_blackjack` in TableDetails in `src/server/server.h`). This did present some interesting concurrency challenges with managing connections on two threads.

3 SERVICE

As mentioned in the design paper, the server defaults to port 21210 if no port is specified in the arguments. This is seen at the start of the main method in `src/server/server.cpp`; the server can be run as either `./server <port> <cert-file> <key-file>` or as `./server <cert-file> <key-file>`. The second option does not re-assign the port number (`short port`) which defaults to 21210.

The client also by default connects to port 21210. In `src/client/client.cpp`, the client defaults the string `port_number` to "21210". The client can be run in one of three ways: `./client`, `./client <hostname/ip-addr>`, and `./client <hostname/ip-addr> <port>`. The second method will leave the port number to the default, but the third will set the port number that the client connects to as whatever you specify. The first method uses UDP broadcast to find the server, which may or may not be running on the default port - more information on this is provided in the Extra Credit section.

4 CLIENT

As mentioned above the client (`src/client/client.cpp`) accepts the hostname or IP address as the first argument in main. The `OpenConnection` method in this file looks up the address information for the given hostname/IP through `getaddrinfo` and then establishes a socket connection to the given address.

5 UI

The client implements a simple command line UI at the end of main in `src/client/client.cpp`. A loop is done on the authentication sequence for username and password. Once the user is successfully authenticated (a 2-series response is received after password), then there is a loop on entering commands. A list of commands is provided, and the user may enter commands at the different states to send various PDUs and data to the server.

6 EXTRA CREDIT

I implemented the extra credit through the use of UDP broadcast. UDP broadcast enables the client to send a datagram to all devices listening on the network at a given port. The UDP broadcast port that the CBP server listens on is port 21211. The client, when run with no arguments, sends a datagram to the local network broadcast address (see `get_blackjack_server` in `src/client/client.h`). It then kicks off a thread (see `get_udp_datagram`, same file) to receive a UDP response from the server. While it waits, the client may re-try sending the datagram if it gets no response. The datagram contains the null-terminated ASCII string "CBP".

The server, on start-up, creates a detached UDP receiver thread in main (`src/server/server.cpp`) which runs `handle_broadcast` (see `src/server/server.h`). This method opens a server connection to the dedicated broadcast port and waits to listen for a message. On a message, the server checks if the message contains the string "CBP". If it does, the server assumes that a client sent the message; the server then sends back the CBP service port, encoded as a null-terminated ASCII string, with the intention for the client to use this port.

The client runs a receiver thread `get_udp_datagram` as mentioned. This method gets back the port from the server. The client determines the server's IP address from the socket connection itself (using `inet_ntop`) and gets the port from the UDP datagram. The UDP broadcast loop and the UDP receiver thread both terminate, returning a string with the IP and port to connect to for service. The client then splits this string off a delimiter to get the IP and port separately, and then initiates the SSL TCP connection with those details.

My approach here should work on any local network thanks to UDP broadcast but of course will not work between a client and server over the internet that do not share the same local network. It is also a fairly simple service lookup mechanism, without any form of security. It is also possible, that if someone wanted, they could write a fake server which responds to the UDP request with an invalid port, and the client would then get stuck trying to establish a connection to a CBP server that does not exist. So, assuming that the client and server share the same local network, and that there are no bad actors, this approach should work. In my testing, the client is able to successfully locate and connect to the server every time I run it.