Matchmaking TicTacToe

By: Rishabh Hatgadkar

Motivation

Matchmaking TicTacToe is a networked TicTacToe game where incoming pairs of clients automatically get matched to games. This is similar to many multiplayer matchmaking systems found in many computer games, such as Blizzard Entertainment's Battle.net in Warcraft III and Microsoft's Internet Checkers in Windows XP. The goal of this project was to develop a multiplayer matchmaking system for a TicTacToe game.

Game Logic

Clients connect to a parent server that is hosted on port 4950. The parent server will send a port from 4951-5050 to the client. The ports from 4951-5050 represent the ports that the child servers will be hosted on. Child servers are created by forking the parent server process. Child servers are the servers where pairs of opponents play their matches in. So 100 matches can be played concurrently. A child server is considered "fully occupied" when the number of clients connected to the child server is 2. When all the child servers are "fully occupied," clients will be forced to wait until a child server is available. The server assigns ports to connecting clients for which there is already 1 client connected to it.

When clients successfully connect to a child server, they can provide login information. Login information allows player win/loss records to be saved. The child server will return the win/loss records to the client if provided. When 1 client is connected to a child server, that client will have to wait for up to 40 seconds for a second client to connect to the child server. If a second client has not connected after 40 seconds, the initial client will automatically begin searching for a new child server to join. The former child server will close.

Once both clients have connected to a child server, the match will begin. The child server is the intermediary system that allows for communication between both players. Players have 15 seconds to make a move. If 15 seconds have passed, and a move has not been received by the player whose turn it is, this will be considered as a loss for the player. The other player would win the match in this case. If 30 seconds have passed, and a move has not been received by the player who is waiting for the other player to make a move, this is considered as a connection loss. In a connection loss, no player would win the game. There are circumstances when both players have lost connection. In this case, the child server would be able to detect this if it had not received a move in 40 seconds. If this is the case, then no player would have won the game, and the child server would exit. If a move has been received within 15 seconds, that move will be forwarded to the child server, and the child server will forward the move to the other player. If a player wins or ties a game, this information will be forwarded to the child server, which will be forwarded to the other player. A player is able to forcefully give-up by quitting out of the client. In all these cases, the child server will record the outcomes of the scores in the database and then close.

Technologies Used

The server is written in C and C++. The client is written in Java.

- The server and clients use TCP sockets
  How is this used: TCP sockets are used to enforce reliable communication between the clients and servers. Most read operations on the sockets for both the server and clients use timeouts to prevent socket read blocking. TCP sockets are used as opposed to UDP, because TicTacToe is a turn-based game and not a real-time, time-critical game.
- The server utilizes queues to assign child server ports to incoming clients
  How is this used: Empty servers queue stores the ports from 4951-5050 that have population 0. Waiting servers queue stores the ports that have population 1. The priority is to choose ports that have population 1. So the priority is to first pick from the waiting servers queue. The assigning of child server ports to clients is done in the parent server.
- The server utilizes a named pipe for interprocess communication (IPC).
  How is this used: When a child server closes, this corresponds to the child process of the parent server closing. So before the child server process closes, the port of the child server is written to a named pipe. The parent process reads the named pipe and adds the port to the empty servers queue.
- The server utilizes pthreads in child servers.
  How is this used: Each thread is used to handle messages sent and received from both clients concurrently. Even when it is another player's turn, the non-playing player can still give-up. This needs to be handled while the other player's receiving server thread is waiting for a move.
- The server uses PostgreSQL.
  How is this used: The database maintains two tables about login records and player records. Login records maintains information about player username, password, and whether the player is currently in game. Player records maintain win/loss records for each username in login records. When a client sends its login information to a child server, the child server first checks to see if this login information is currently in the login records table. There are 4 cases to this:
    - The username and password match, and the player is not in game. In this case, the client is allowed to proceed to join the child server.
    - The username and password match, and the player is already in game. In this case, the client is not allowed to proceed to join the child server, because the same user is already in game.
    - The username and password do not match. In this case, an entry will be added to the login records table to add the new username and password. The player records table will set the win/loss records to 0 for the new username.
    - The username matches, but the password does not match. In this case, the client will be denied entry, because the password does not match the username from the login records table.
- The client uses Runnable implemented threads.
  How is this used: Multithreading in the clients is used to implement the timers for receiving and sending moves. If 15 seconds have passed and a move has not been sent,

this corresponds to a give-up. If 30 seconds have passed and a move has not been received, this corresponds to a connection loss. There is also a receiver thread in both clients, that continuously checks for messages received from the child server. This is necessary, because information from the child server can happen at any time when the player is playing a move or waiting for a move. A give-up message is an example. Shared variables, such as the receive buffer, are safely locked using synchronized methods with the main thread and the receiver thread.

Testing

Testing was done by spawning multiple client sessions using GNU Screen and getting the output of Screen sessions by saving them in log files. A custom Java client was created that automatically plays user input moves from provided arguments. Every time a game would complete and the client exits, the client will start again with the same user input moves in the Screen session. The automation of starting client sessions after games are over and parsing the log files and database records from the server are done using Python. The number of win/loss records are found by counting the number of win/loss statements in the CLI output using Grep and comparing them with the win/loss records of the database in the server. This is automated using Python.

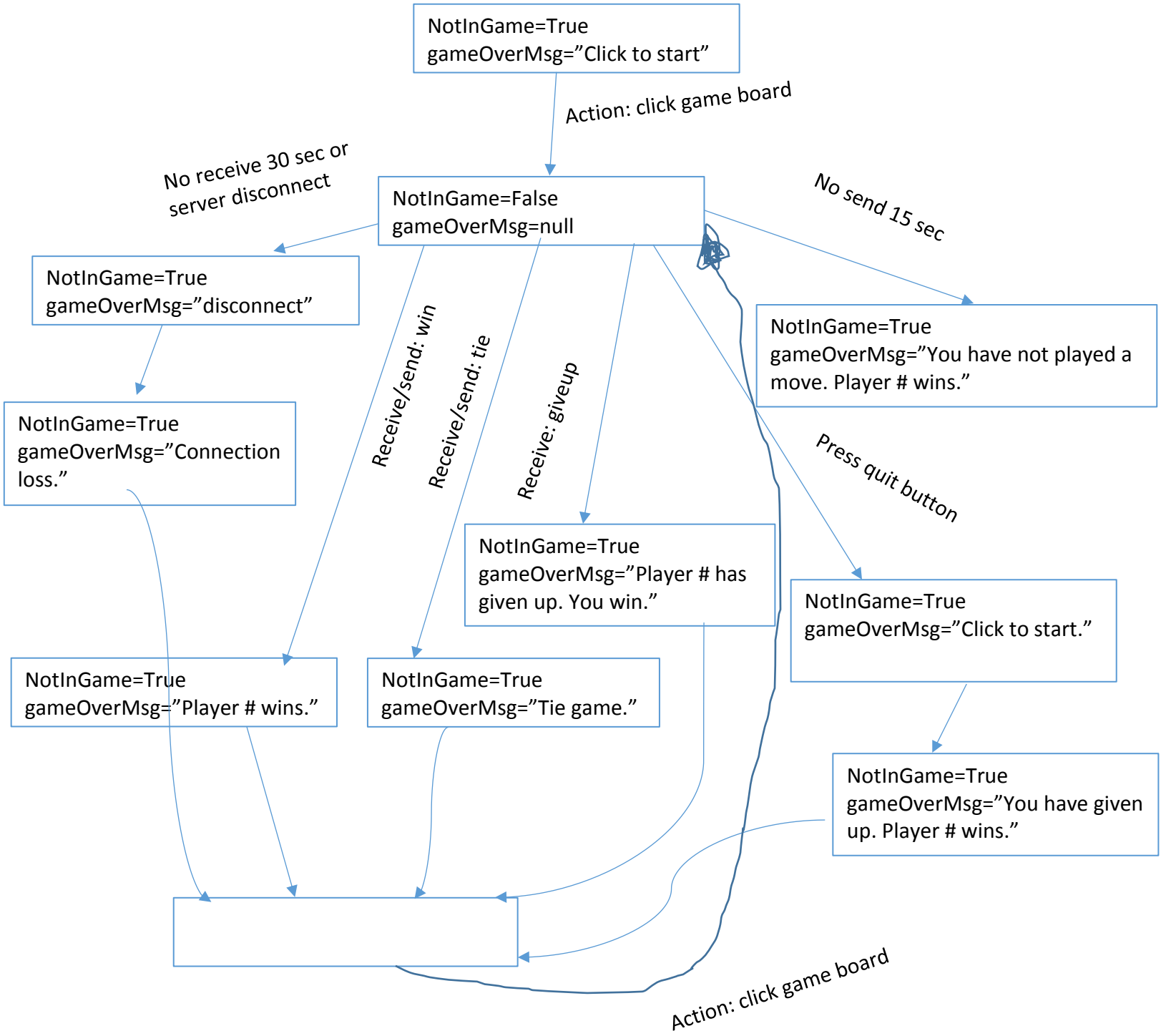
Functional tests were written to test scenarios such as when a player wins, gives up, and when simultaneous clients are in game.

A stress test was done by spawning 512 clients simultaneously. The Python script deploys 31 mock clients in 17 EC2 instances. After each instance has finished running the script, it compares its clients' win/loss records with the logs and the database server, to make sure the clients' win/loss are correctly recorded with the database's win/loss records.

Unit testing was done for the clients and servers. The client unit tests test scenarios to verify the correct game over message is displayed when given a new incoming message from the child server and the current layout of the board. The server unit tests check for correct behavior of both the parent server and the child server. The parent server unit tests verify that the correct population and port is sent to the client given values in empty queue and waiting queue. The child server unit tests verify that correct messages are forwarded to the clients from the match threads and when receiving player records.

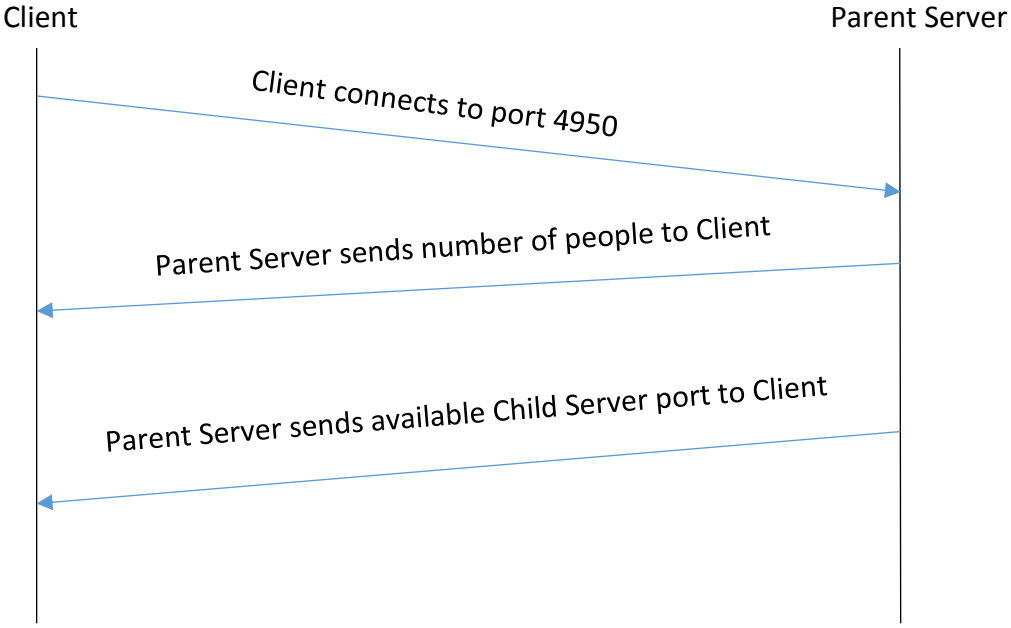
Java Client State Using NotInGame and gameOverMsg Variables

The game starts with NotInGame=True and gameOverMsg="Click to start" after logging in.
When the client clicks the game board, NotInGame=False and gameOverMsg=null.

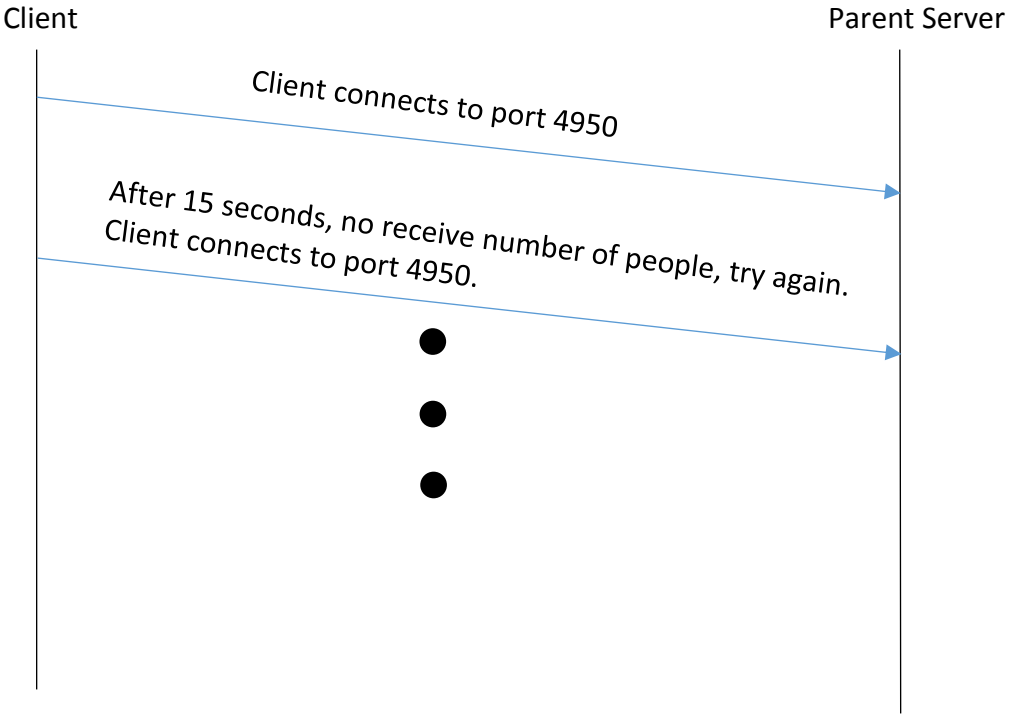


Sample Server and Client Interactions

Client connection to Parent Server -> Successful

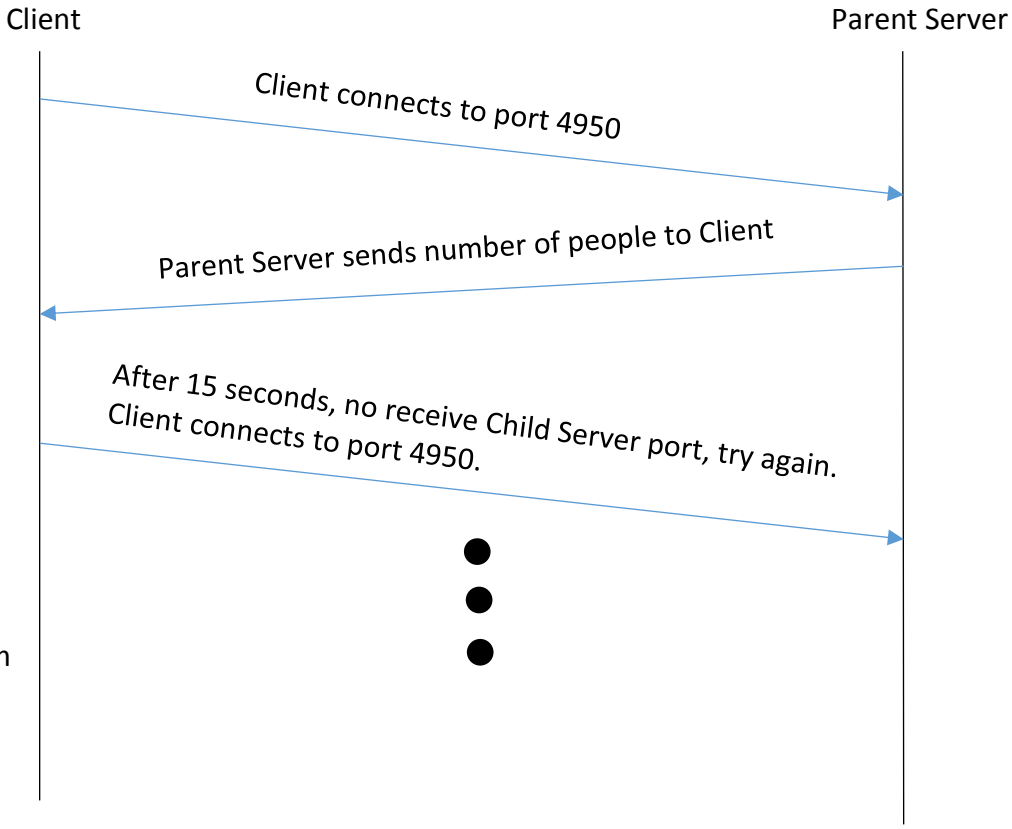


Client connection to Parent Server -> Timeout for number of people



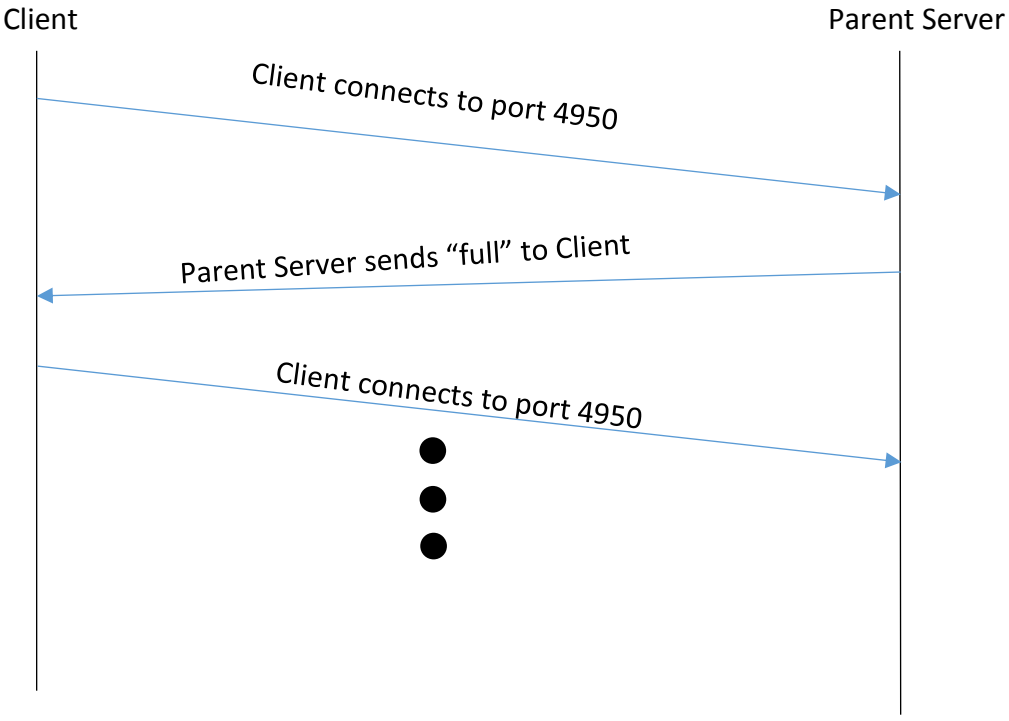
After 10 retries if not receiving from Parent Server, abort the connection.

Client connection to Parent Server -> Timeout for Child Server port



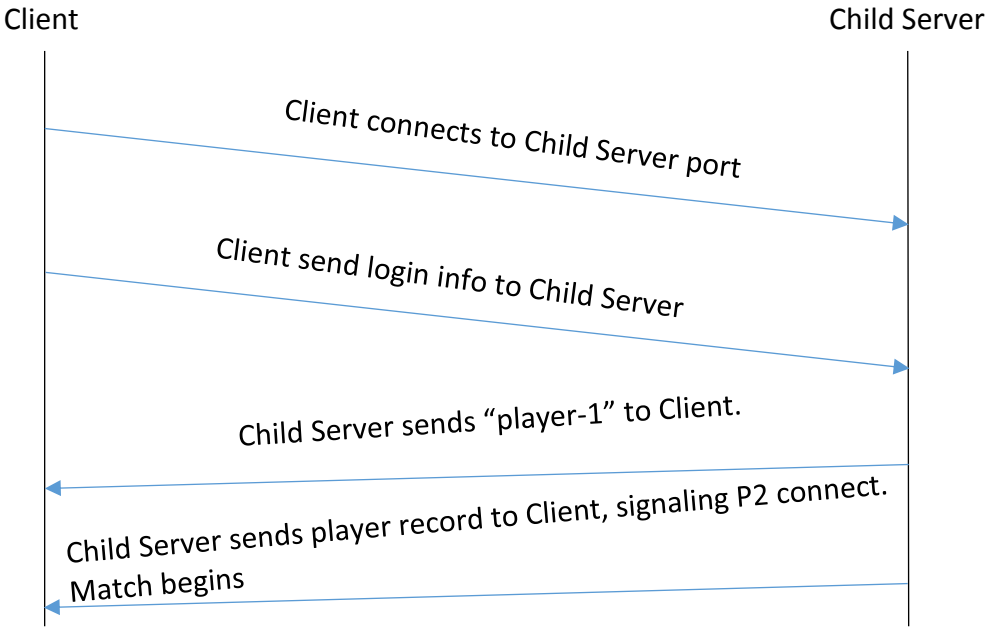
After 10 retries if not receiving from Parent Server, abort the connection.

Client connection to Parent Server -> Full server

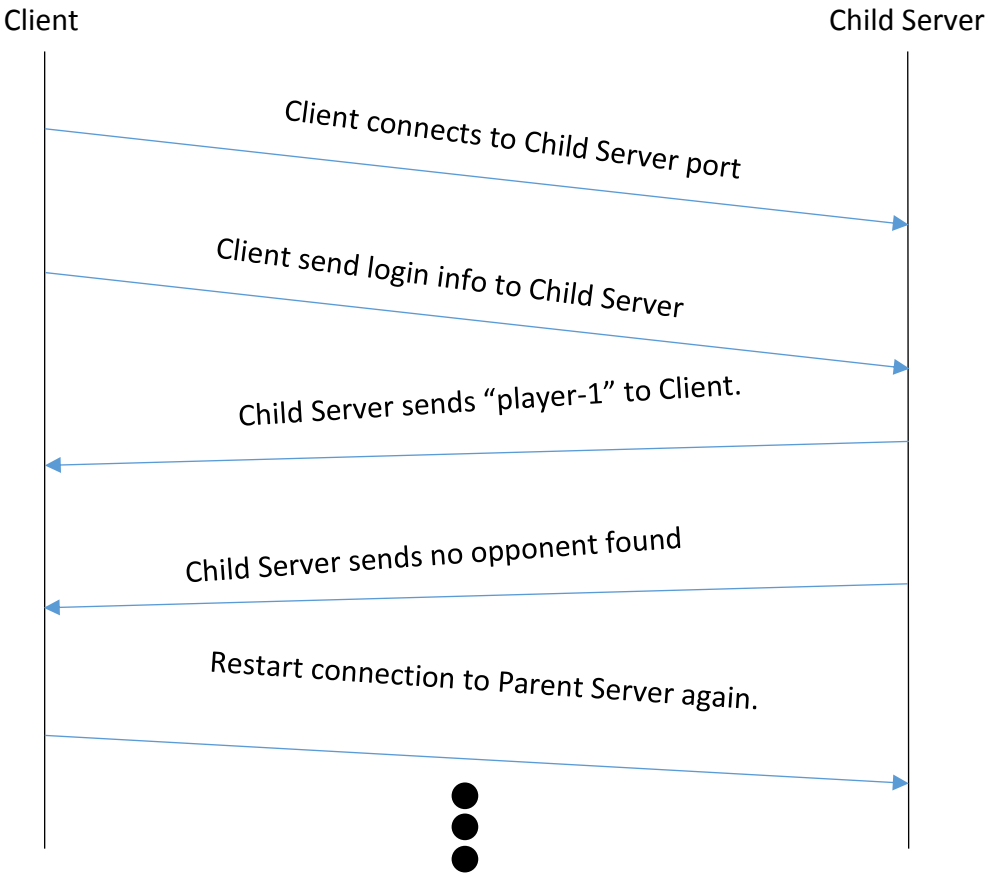


After 10 retries if server is still full, abort the connection.

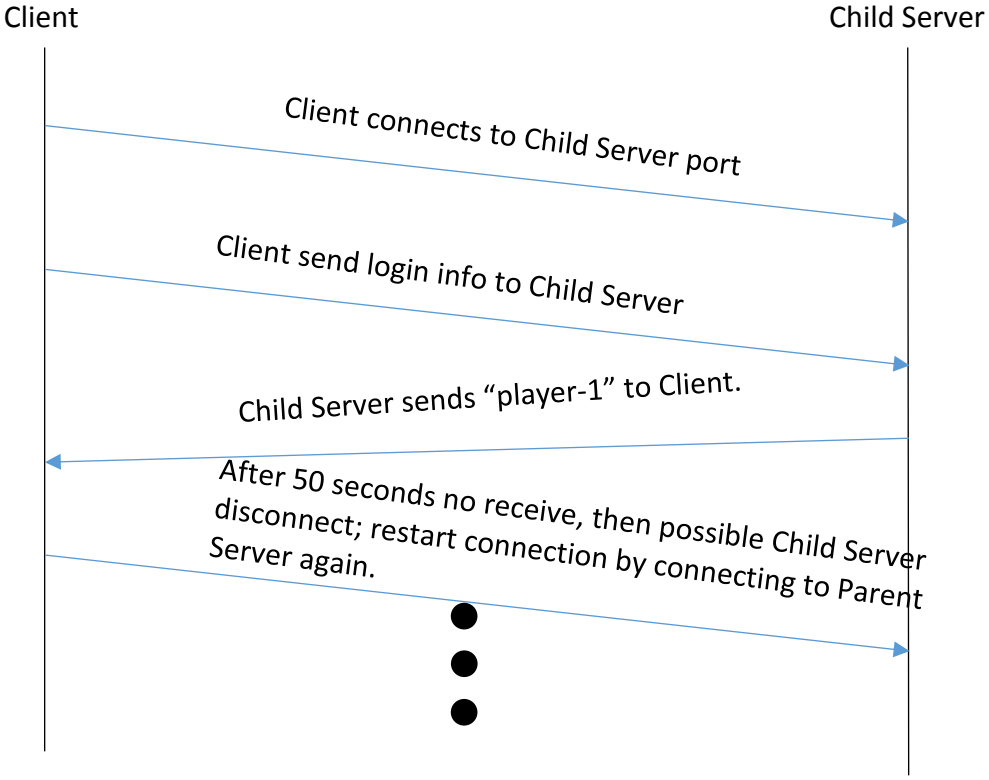
Client connection to Child Server from P1 -> successful



Client connection to Child Server from P1 -> no opponent found

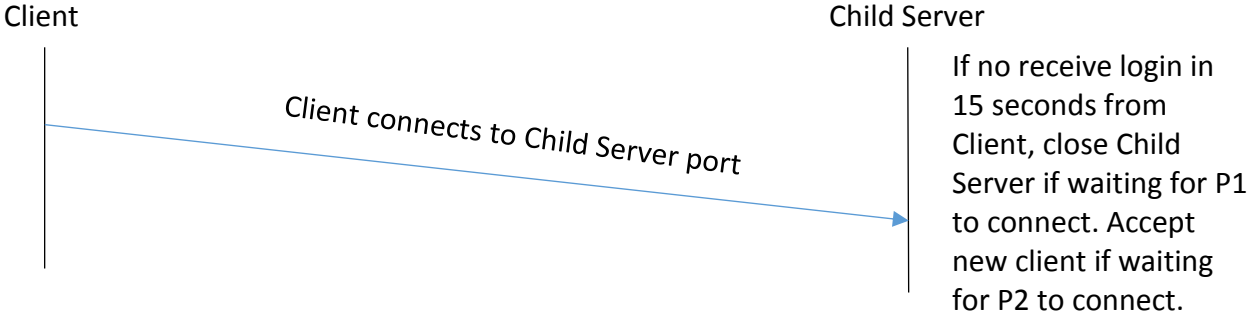


Client connection to Child Server from P1 -> Child Server disconnect

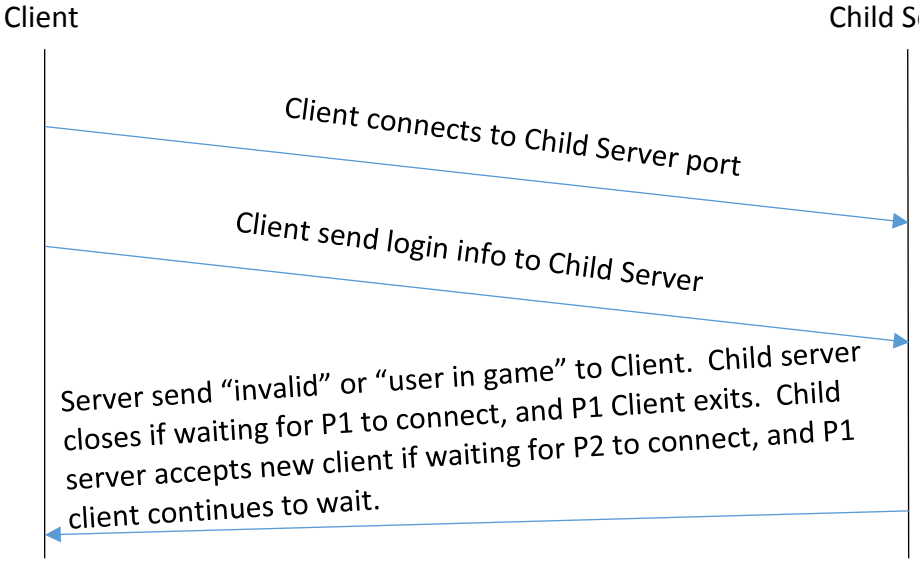


Client connection to Child Server from P1 -> no Client connect

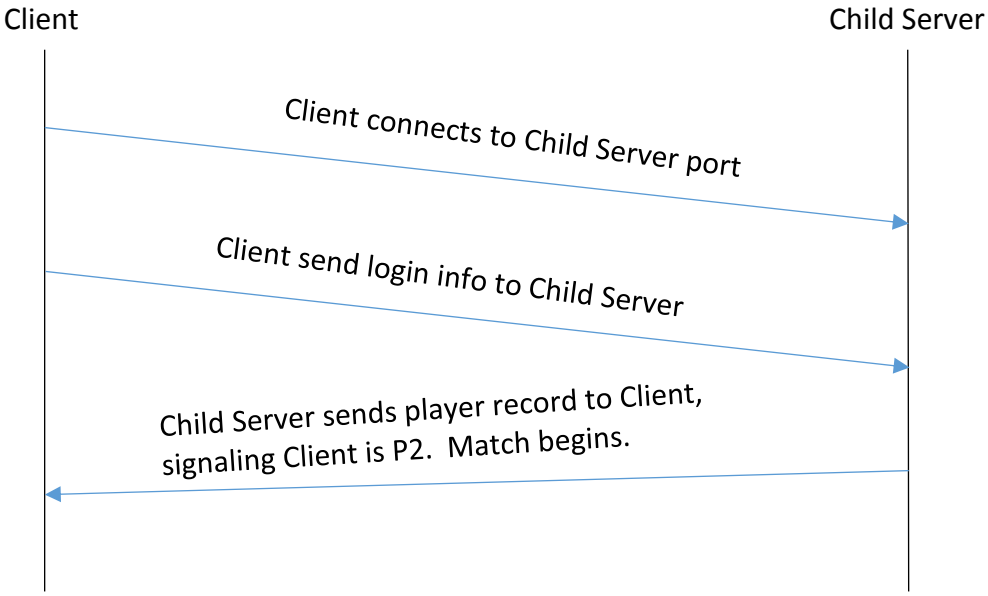
Client connection to Child Server from P1/P2 -> Child Server no receive login



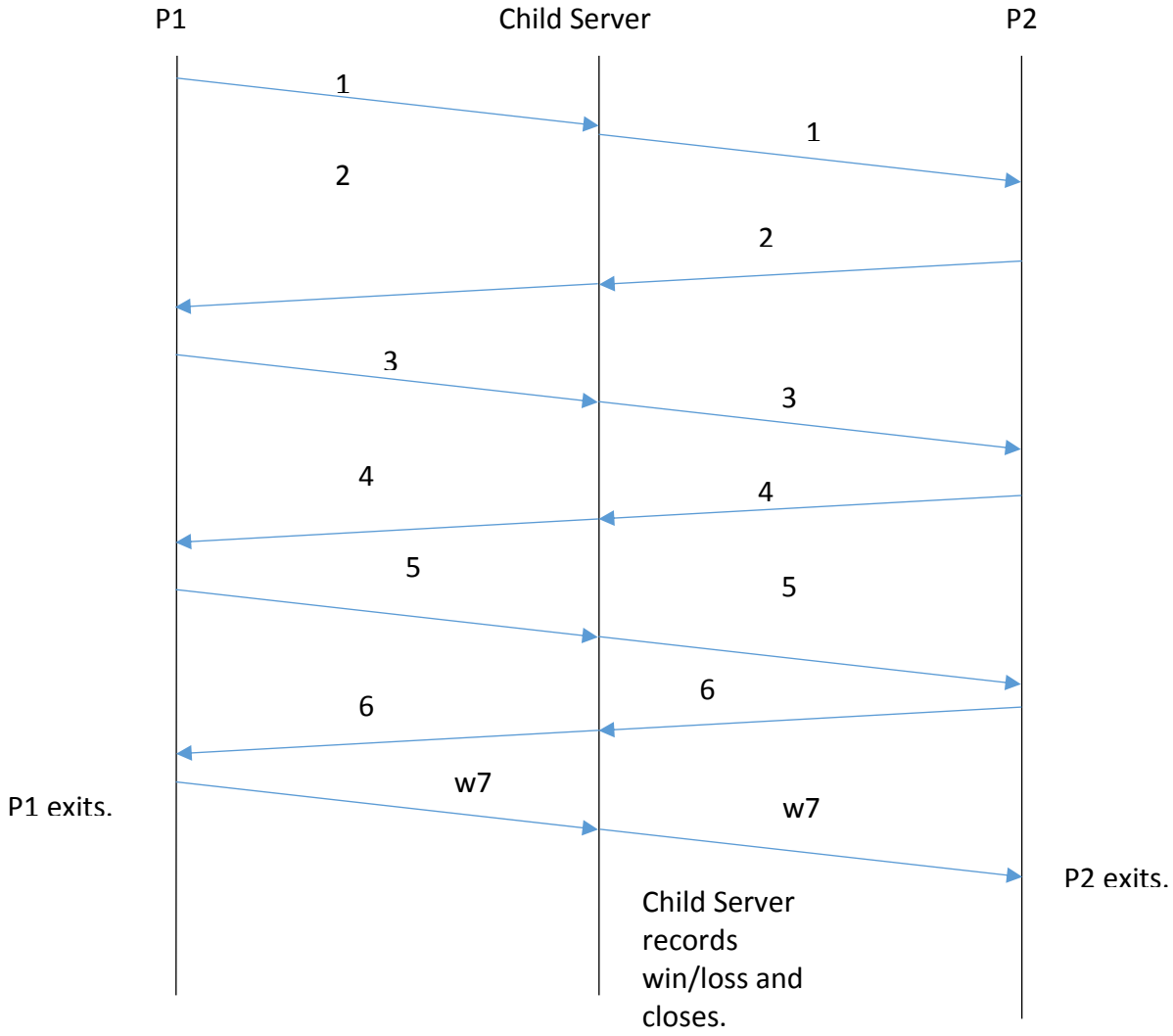
Client connection to Child Server from P1/P2 -> invalid login or user already in game



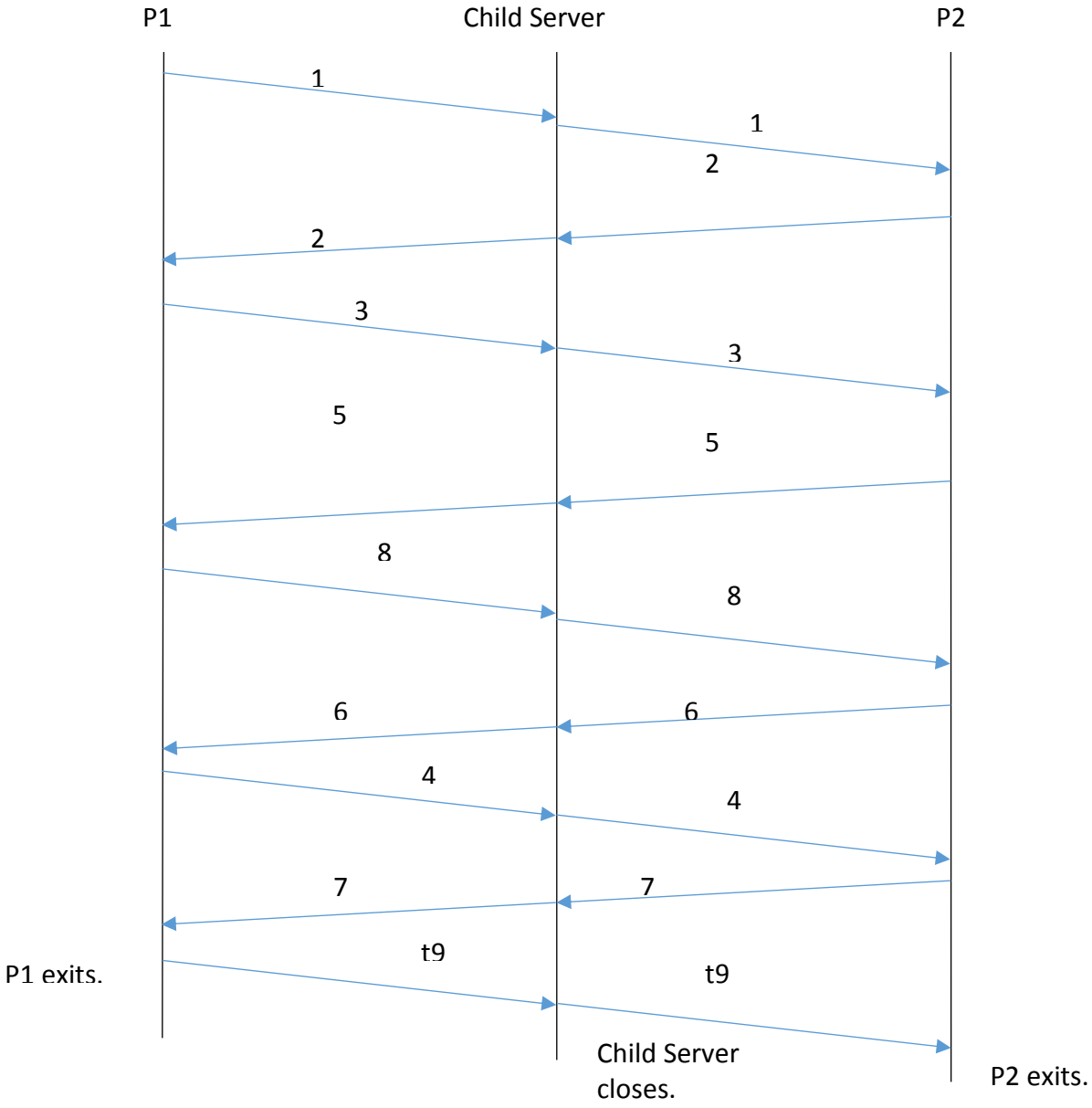
Client connection to Child Server from P2 -> successful



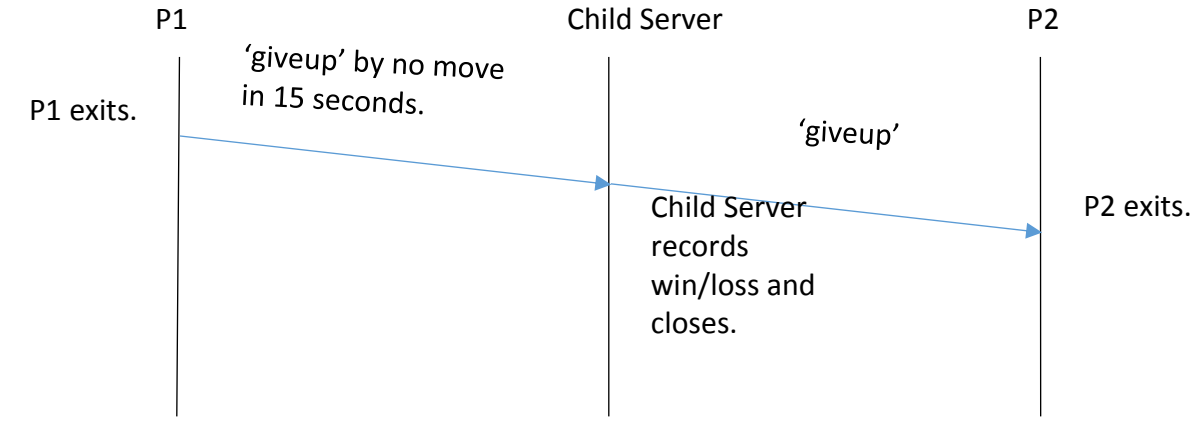
Example match: normal game



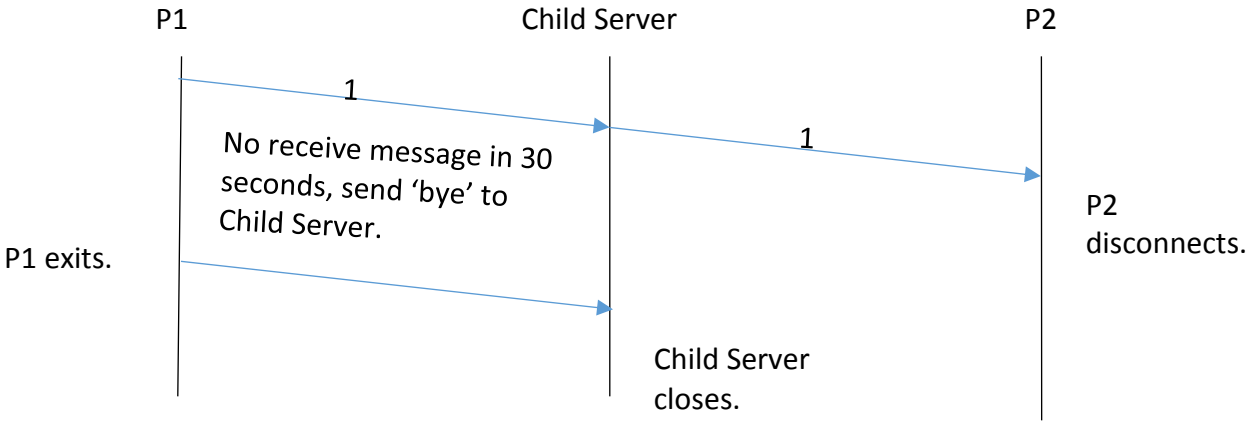
Example match: tie game



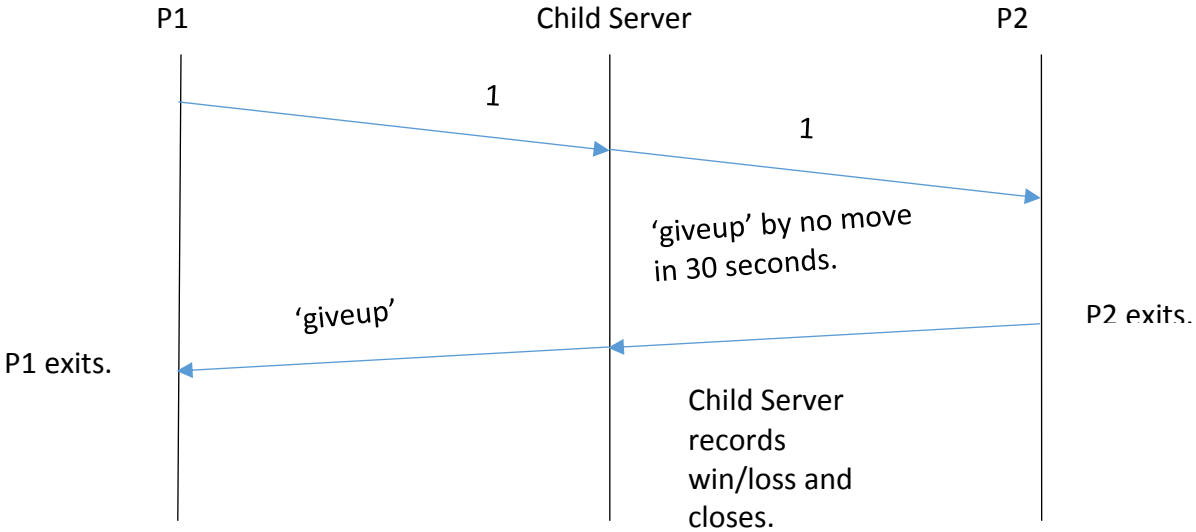
Example match: P1 giveup



Example match: P2 disconnect



Example match: P2 client exit



Example match: no receive from server: