Relatório do 3º Lab de CES-33

Miguel Macedo e Luckeciano Carvalho

Introdução

Historicamente, acredita-se que o jogo da velha surgiu no Antigo Egito, embora a primeira referência escrita que se tem dele com o nome que conhecemos seja de 1864[1], O jogo é normalmente jogado por crianças, já que a lógica é simples e, para dois jogadores que conheçam a "jogada perfeita", o jogo sempre resulta em empate.

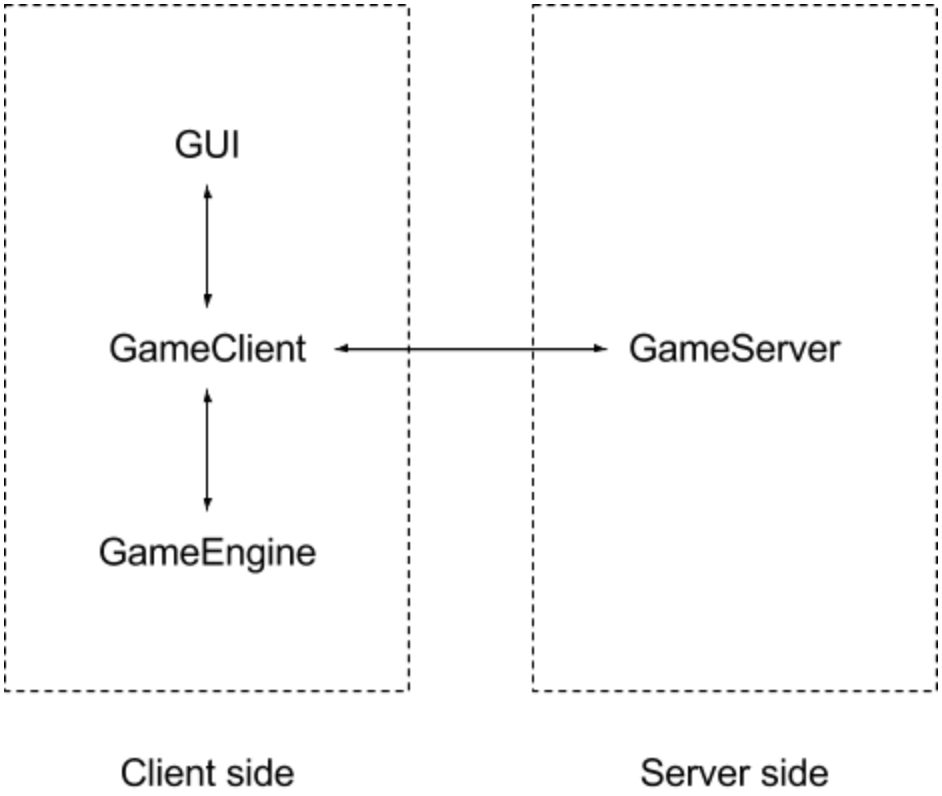
Neste laboratório, implementou-se este jogo. Naturalmente multiplayer, gerou-se uma aplicação na qual cada jogador pode acessar o jogo de uma máquina diferente, na mesma LAN (ou ambos os jogadores no mesmo computador, em threads diferentes dentro do processo).

O principal problema de uma aplicação multiplayer (com clientes e um servidor central) encontra-se na comunicação com o servidor, onde compartilha-se informação entre os clientes. Caso este mediador não esteja sincronizado com os clientes (por exemplo, em uma condição de disputa na informação guardada pelo servidor - no caso, informações do tabuleiro), pode-se ocorrer mau funcionamento do jogo (por data race e deadlocks, como exposto em seções posteriores).

O objetivo deste laboratório, portanto, encontra-se na implementação da aplicação e no evidenciamento das condições críticas, e como pode-se tratar tais condições no projeto, a fim de evitar race condition.

Funcionamento das Classes[2]

Há 4 classes responsáveis pelo funcionamento do jogo. A comunicação entre os dois jogadores se dá através de um servidor (*server*), de modo que os jogadores são clientes (*clients*). 3 classes ficam no *client side* e uma classe fica no *server side*, de acordo com o esquema a seguir:

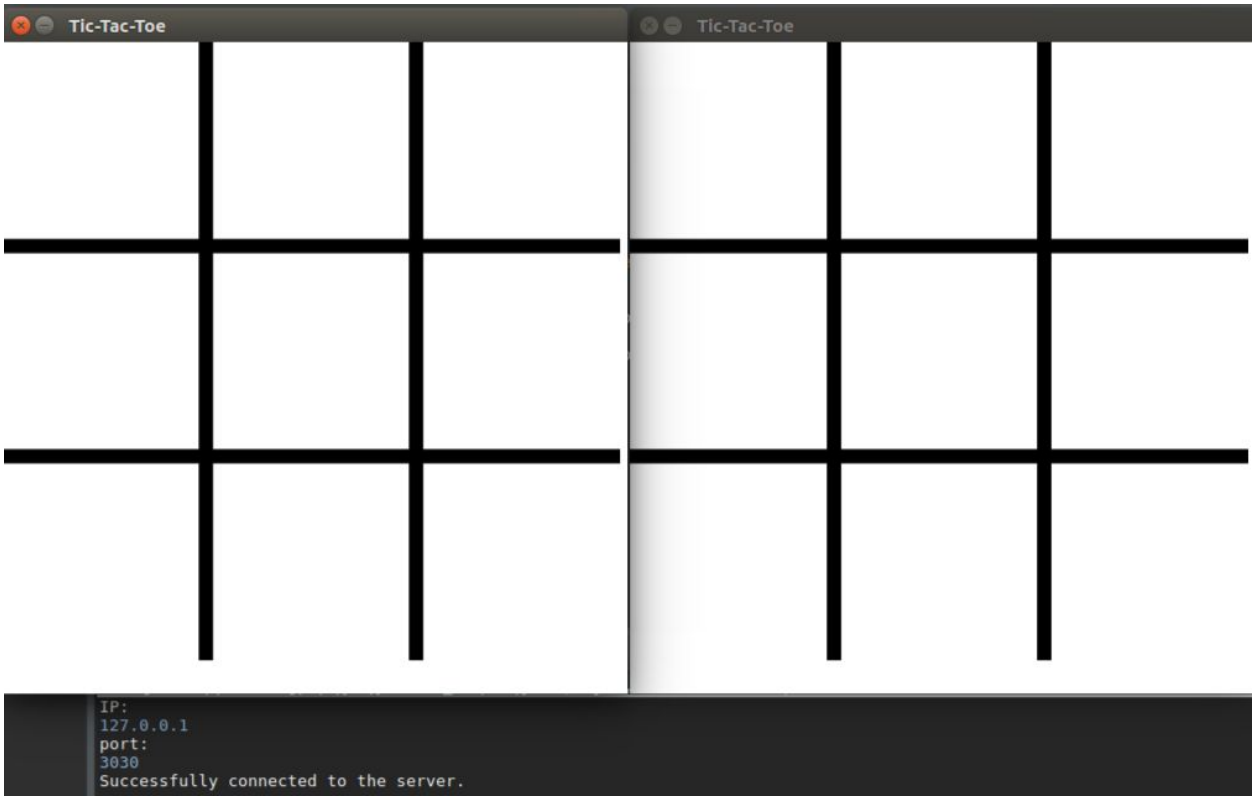


A GUI é responsável por imprimir na tela o tabuleiro do jogo e as jogadas já feitas. O GameEngine é o jogo em si, ele faz os cálculos e verifica se o jogo ainda está em curso ou se já acabou, para então indicar o vencedor ou empate. O GameServer fica responsável pela comunicação entre os dois jogadores, para que as jogadas de um jogador apareçam para o outro, garantindo a multijogabilidade do jogo. Por fim, o GameClient conecta todas as classes, enviando informações de uma classe à outra.

Mas e o Multithreading?

O multithreading se dá na comunicação entre os dois clientes. Ao executarmos a Main do jogo para o primeiro jogador, ele pede um IP e uma porta. Obviamente, o primeiro jogador não vai encontrar um servidor, e assim ele *cria* um servidor naquele IP e naquela porta, e fica esperando pelo próximo jogador. Na figura abaixo podemos ver no console como ele faz a criação do servidor, e como a GUI fica esperando pelo próximo jogador:

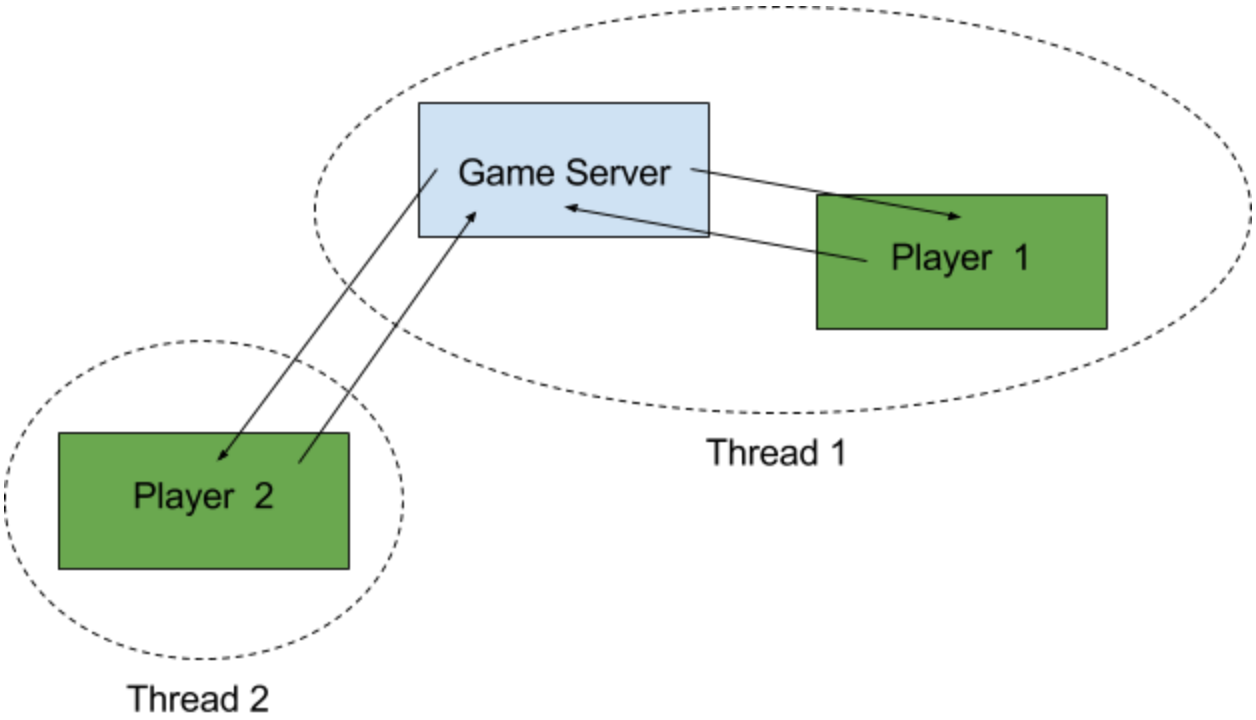
O segundo jogador, inserindo os mesmos valores de IP e porta, encontra o servidor criado pelo primeiro jogador e se conecta a ele, como na figura abaixo. No console podemos ver que a conexão com o servidor teve sucesso, e agora temos os dois jogadores prontos para jogar:



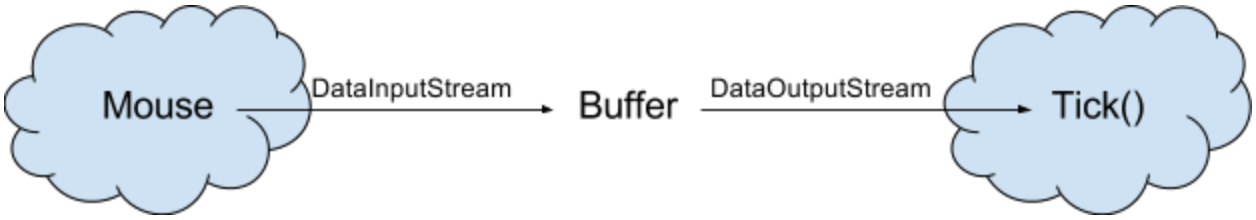
A conexão é feita por *sockets*. Cada GameClient possui um *web socket* que se conecta ao *server socket* do servidor.

Mas onde fica o multithreading?

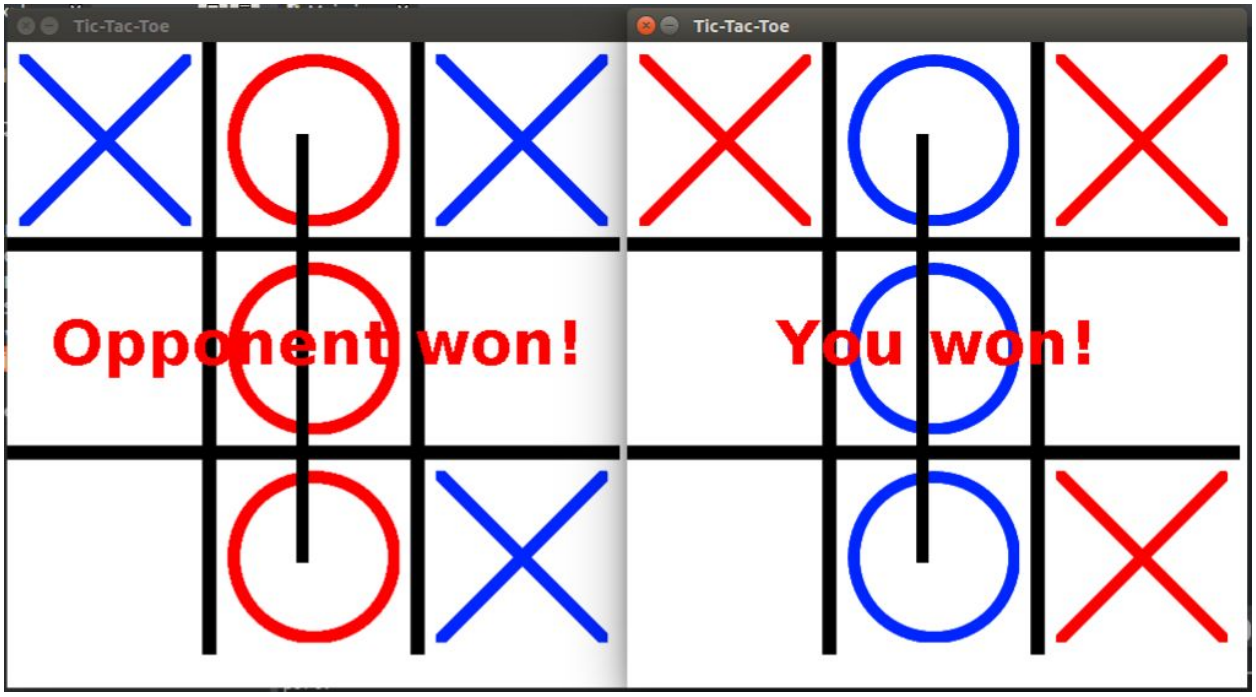
Se você visualizou cada jogador (o *client*) como um thread, você está quase correto. Na verdade, cada Main executada é um thread, de acordo com o esquema abaixo. As setas indo e voltando são uma conexão socket: o *server socket* pode se conectar a múltiplos *clients*, e o *web socket* dos clients se conecta a um único servidor. Ainda, tal tipo de conexão possibilita que o jogo seja jogado em dois computadores diferentes que estejam na mesma rede a partir do IP dos dois computadores (a conexão para computadores de redes diferentes não foi implementada por não ser tão importante para o assunto). O primeiro jogador criaria então um GameServer em um computador a partir do seu IP, em uma dada porta, e então um segundo jogador se conectaria a esse IP nessa porta.



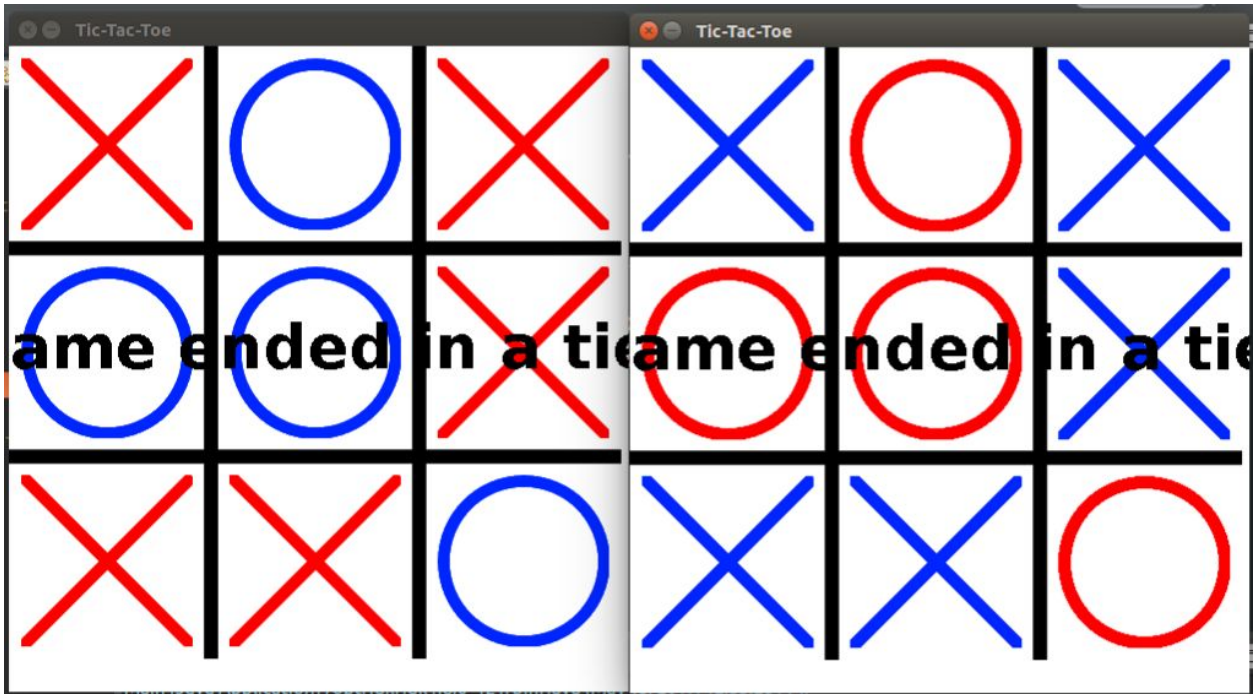
Com relação a compartilhamento, o jogo da velha pode ser enxergado como um problema de produtor-consumidor bem simplificado, pois o buffer é de um único elemento (o input no tabuleiro, que indica a posição clicada de 0 a 8) e não há múltiplas produções ou consumos, apenas um de cada por vez. A função *tick* na GUI é o consumidor, e o clique do mouse no tabuleiro é o produtor. Vale notar que o clique de mouse em um jogador é produtor para o Tick() do outro jogador também, pela sincronização dos *threads*. O servidor é o mediador, e ele tem o buffer, **ainda que cada jogador tenha as suas Input- e OutputStream**.



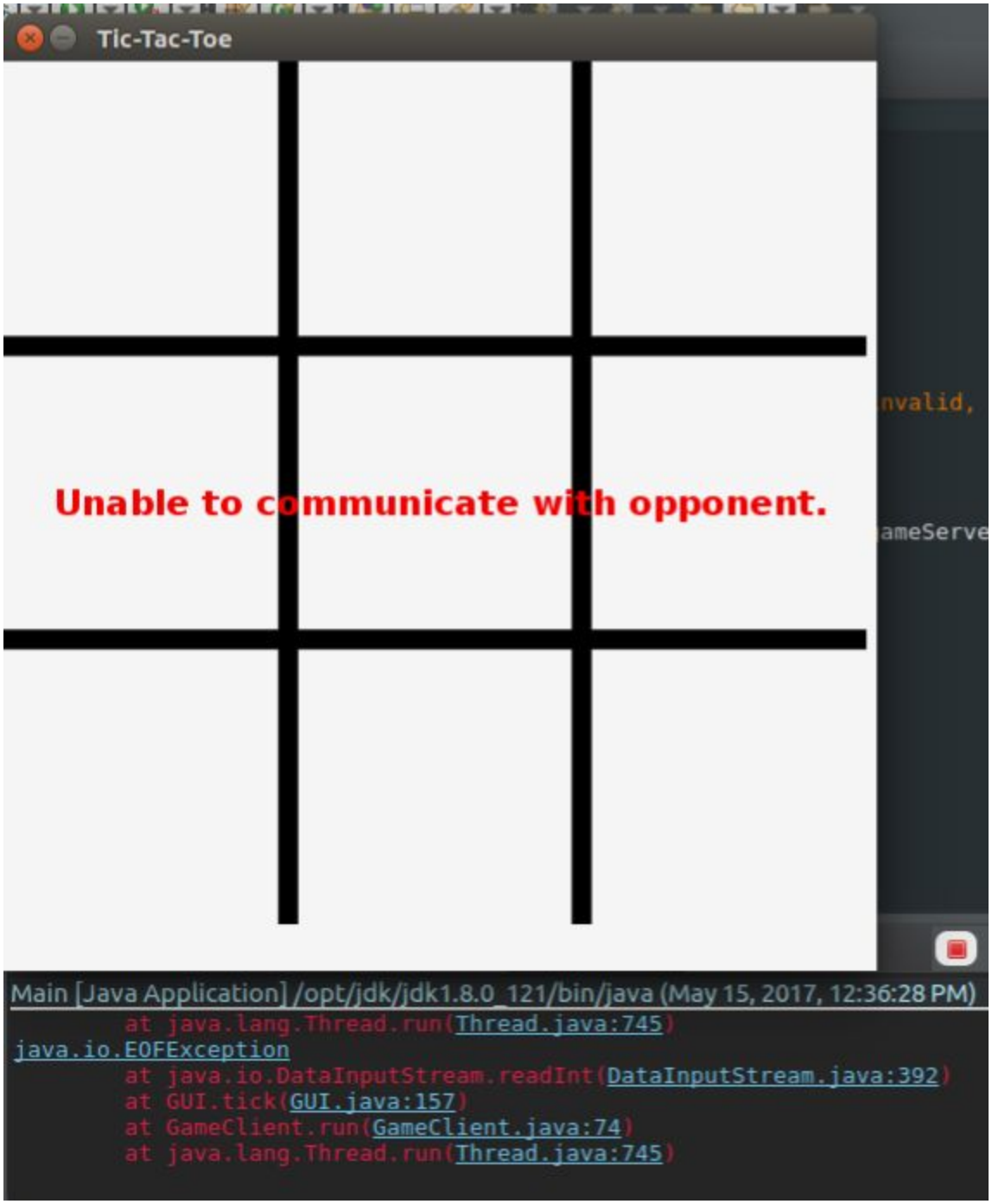
Por fim, o jogo utiliza **cores relativas**, isto é, as suas jogadas são representadas em azul e as do seu oponente em vermelho. Para uma partida que acabou em vitória, o tabuleiro fica da seguinte forma:



Para um caso de empate:



Para o caso de uma janela ser fechada durante o jogo, ocorre uma exceção ao clicarmos no tabuleiro de um, e a GUI do outro jogador fica como na figura abaixo:

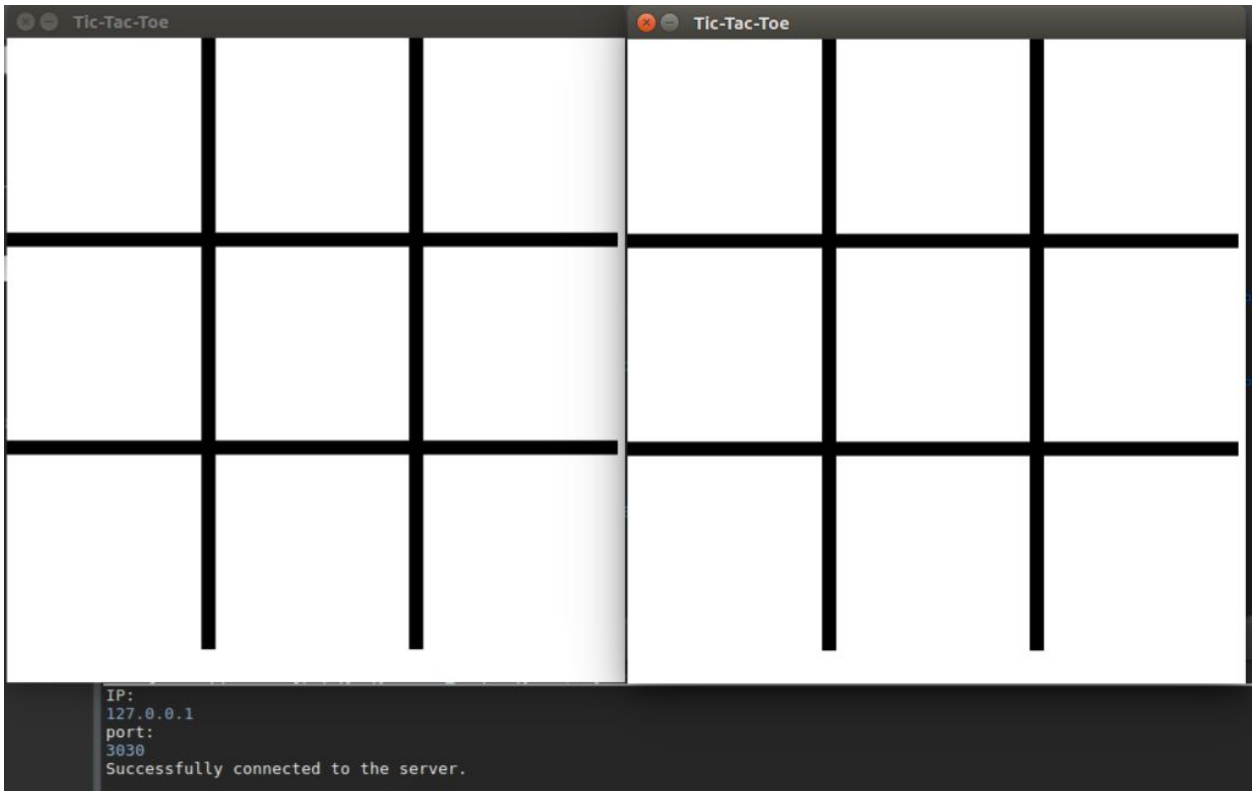


Condições de disputa

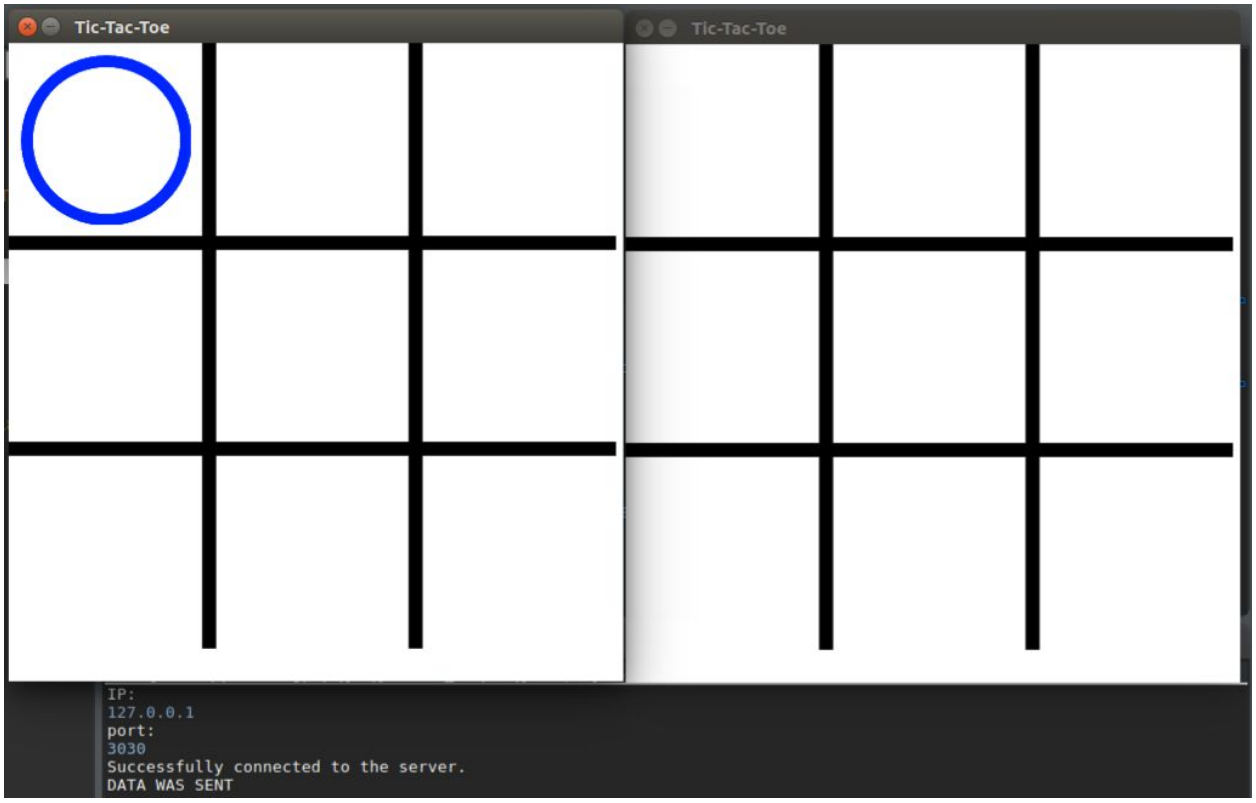
O programa foi implementado em Java. Isso não quer dizer que a linguagem se encarrega de impedir qualquer condição de disputa, é possível atingir esse problema. No caso deste jogo, a sincronização é garantida através de uma variável *_yourturn*, que impede que um tabuleiro fique ativo quando é a vez do outro. Inicialmente, *_yourturn* começa com *true* pro

client que cria o servidor e *false* para o segundo jogador, e a cada jogada a variável *_yourturn* de cada um deles alterna para permitir a jogada do próximo.

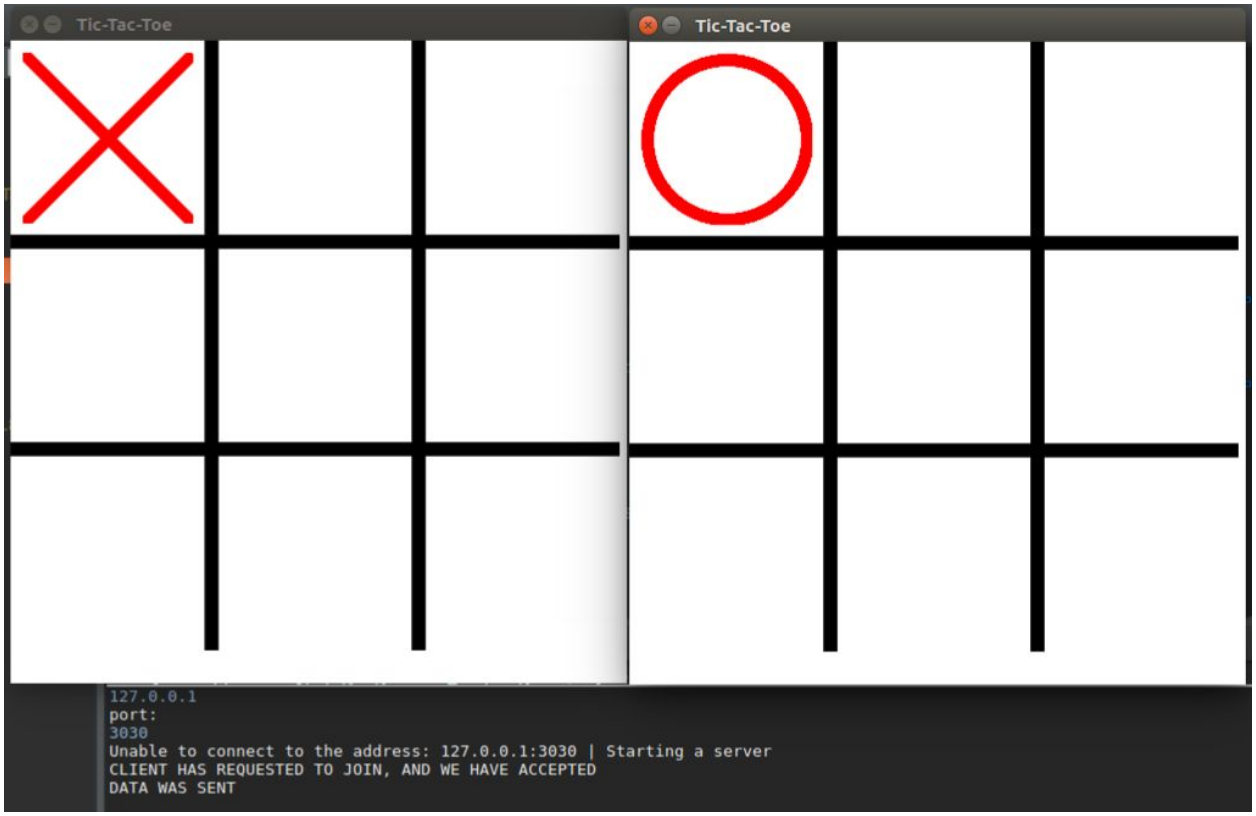
Na inicialização da classe, a variável *_yourturn* é inicializada com *false*, e é setada para *true* no momento que se cria o servidor. Se mudarmos a inicialização da classe para que *_yourturn* já comece com *true*, a sincronização não ocorre, e podemos chegar a problemas, como mostrado nas figuras a seguir:



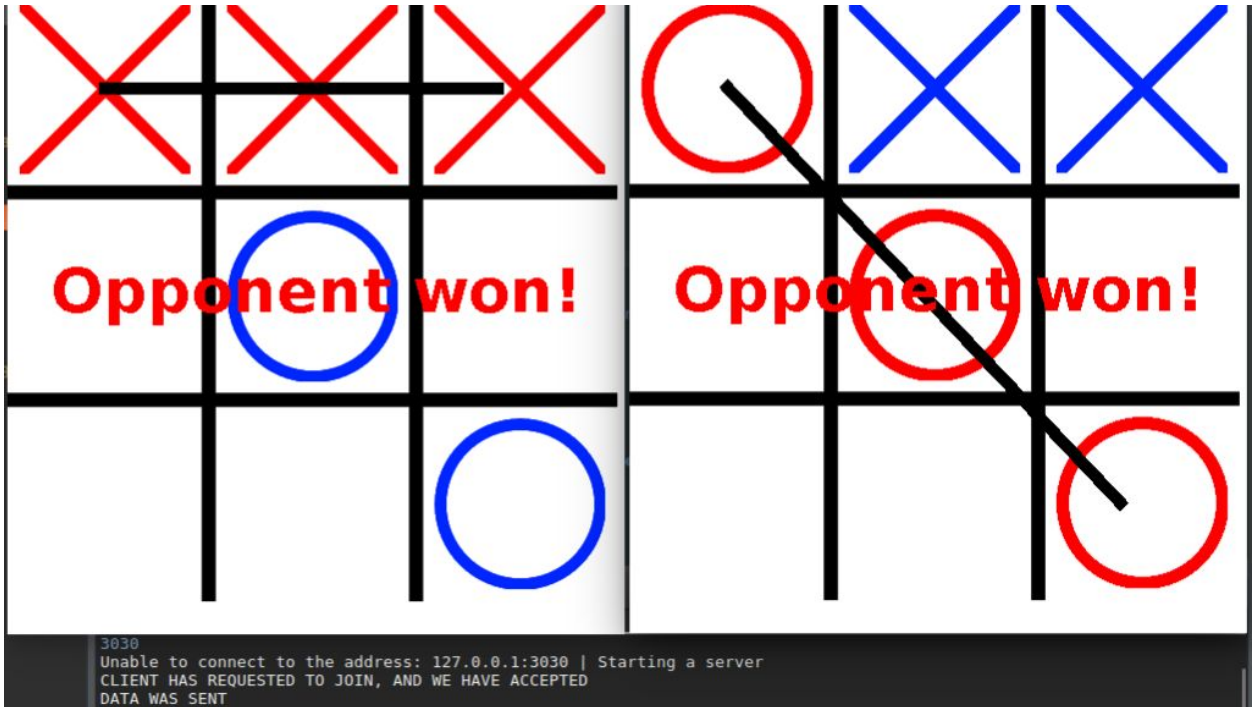
A conexão ocorre normalmente, como mostrado no console, mas ao tentar jogar:



Os tabuleiros não ficam sincronizados, e ainda tem mais problemas:



Os tabuleiros permitiram que se jogasse duas vezes no mesmo lugar, que pode levar a uma situação absurda como os dois ganharem:



Isso ocorre porque o buffer não está sendo compartilhado pelos dois, ainda que os dois estejam conectados pelo servidor, de modo que é possível ver que um completou o trio de X (ou O) por causa da conexão, mas é como se fossem 2 jogos diferentes “se atrapalhando”, com os movimentos chegando atrasados, não mais o mesmo jogo entre os dois.
No caso em que não setamos a variável *_yourturn* do primeiro jogador para true quando ele cria o servidor, chegamos a um *deadlock*, pois nenhum jogador consegue jogar.

Para ilustrar o funcionamento da variável *_yourturn* como *semáforo*, há um bloco comentado na função run() do GameClient. Ele aplica um atraso na verificação de informação, o que geralmente é suficiente para que o programa decida sair de um *thread* e ir para outro. Mesmo com o atraso, a variável *_yourturn* garante que a sincronia dos eventos. O jogo roda normalmente, sem disputa por recursos, apenas mais atrasado do que deveria.

Conclusão

Neste laboratório, desenvolveu-se o jogo da velha em rede, no qual é possível dois jogadores conectados na mesma rede local, mesmo que em máquinas diferentes. Neste contexto, compartilham-se recursos do jogo por meio do GameServer - um mediador - e existe a condição de disputa de recursos, tratada a fim do correto funcionamento da partida por meio de uma variável de sincronização (*_yourturn*).

Mesmo usando uma linguagem que supostamente cuida da sincronização dos *threads*, podemos chegar a situações de corrida até mesmo para sistemas muito simples, como o desse jogo (que há poucas produções e consumos). Entender o processo de sincronização e o compartilhamento de recursos comuns é importante ao desenvolver programas exatamente para saber lidar com esses problemas que ocorrem tão frequentemente.

Referências

[1] Histórico do jogo: https://en.wikipedia.org/wiki/Tic-tac-toe#History. Acesso em 15/05/2017.
[2] “Make a Networked Tic-Tac-Toe in Java”. Disponível em: https://www.youtube.com/watch?v=aIaFFPatJjY&t=50s. Acesso em 15/05/2017.