MAC0352 — EP 2

André Souza Abreu Bruno Pereira Campos

04/06/2022

TTTP

Para implementar o jogo da velha utilizando um sistema hibrido (P2P + client-servidor) criamos o TTTP — TikTakToe Protocol (*Protocolo do Jogo da Velha*).

Todas as mensagens trocadas (servidor-cliente, cliente-cliente) são em *plain text*. Existem dois tipos de mensagens: `requests` (requisições) e `replies` (respostas).

O protocolo foi implementado totalmente em `Python` por ser uma linguagem de desenvolvimento rápido e com suporte de várias bibliotecas, tais como `API` de sockets, `threading`, manipulação de arquivos, etc.

Nota: pelo fato de ser *plaintext*, é possível olhar as mensagens no *Wireshark* e até mesmo simular um cliente TCP usando o *telnet* (basta saber o protocolo).

requests

Os `requests`, também chamado de comandos, seguem sempre o seguinte formato:

`<CMD> <ARGS>`

Onde `CMD` é um comando de 4 letras (para facilitar a padronização) e `ARGS` são o conjunto de argumentos (se houver) passado junto com o comando.

Alguns comandos (`HELO`, `PING`, `PINL`) não precisam de argumentos, já outros precisam de um ou mais argumentos, os quais devem ser separados por **espaços** e devem conter **somente** caracteres ASCII.

replies

Já os `replies` seguem o formato:

```
<CMD> <RETURN CODE>
<PAYLOAD>
```

Onde `CMD` é o comando original, `RETURN CODE` é um código (inteiro) utilizado para indicar o *status* da requisição, e `PAYLOAD` contém dados da resposta. Apenas alguns comandos devolvem um *payload*:

- `MSTR` devolve o ID da partida para ser usado posteriormente pelo cliente para especificar o resultado da partida
- `GTIP` devolve o endereço do usuário usado para partidas.
- `UHOF` lista a pontuação dos usuários, um por lista.
- `USRL` devolve a lista de usuários conectados, 1 usuário por linha.

códigos

Nós nos baseamos nos `status code` do protocolo *HTTP* e eis alguns códigos:

- **200**: sucesso ao executar o comando
- **201**: comando executado resultou em um novo recurso
- **400**: comando mal formatado
- **401**: usuário não autentificado
- **403**: ação proibida
- **404**: recurso não encontrado

Neste contexto *recurso* significa um *usuário* ou *match* (partida). Alguns exemplos de mensagens:

exemplos de requisições

- `HELO`: Handshake usado para iniciar comunicação (*Hello*)
- `PING`: Ping utilizado para *heartbeat*
- `PINL`: Ping utilizado para medir latência (*PING Latency*)
- `USRL`: Lista usuários ativos (*User List*)
- `NUSR <USER> <PASS>`: Cria novo usuário (*New User*)
- `LOGN <USER> <PASS>`: Usuário entra (*Log In*)
- `LOUT`: Usuário sai (*Log Out*)
- `CPWD <OLD> <NEW>`: Muda a senha (*Change Password*)
- `MSTR <OPPONENT>`: Início de partida (*Match Start*)
- `MEND <ID> <WINNER>`: Resultado da partida `<ID>` (*Match End*)
- `GBYE`: Desconecta cliente (*Good Bye*)
- `GTIP <USER>`: Obtém IP e porta de outro usuário (*Get IP*)
- `SADR <PORT>`: Cliente anuncia sua porta (*Set Address*)
- `UHOF`: Lista pontuação dos usuários (*User Hall Of Fame*)

Mensagens

Alguns comandos só podem ser executado se o usuário estiver logado (`CPWD`, `MSTR`, `LOUT`), deslogado (`LOGN`, `GBYE`), ou jogando (`MEND`). Para isso, utilizamos máquinas de estados no servidor e no cliente, as quais indicam se o comando pode ou não ser executado.

Algumas mensagens são enviadas no *background* (`HELO`, `PING`, `PINL`, `SADR`, `GTIP`) sem a intervenção do usuário. Na verdade, estas mensagens não podem ser enviadas pelo usuário.

Em vez disso, os comandos digitados pelo usuários são transformados em mensagens especificadas pelo protocolo. Estes comandos adotam a especificação que está no PDF do enunciado deste EP.

Ciclo de vida do cliente

Após o cliente inciar sua conexão com o servidor, ele envia as mensagens `HELO` (handshake) e `SADR` (especificação da porta pública do cliente para partidas). O cliente então entra em modo REPL (*Read-Eval-Print Loop*), recebendo comandos do usuário e devolvendo respostas.

O cliente envia no background um *PING* para o servidor a cada 2 minutos para indicar que ainda está ativo. Durante uma partida, há algo semelhante: o cliente envia uma requisição *PINL* para o outro cliente para medir a latência.

A partir daí o comportamento do cliente dependerá dos comandos do usuário, que poderá criar novos usuários, logar/deslogar, mudar a senha do próprio usuário, listas os usuários ativos, listas a pontuação dos usuários, entrar numa partida, sair do programa.

Partidas

Para iniciar uma partida, o cliente envia um `GTIP` ao servidor para conseguir o endereço público de partida do oponente. Então, envia uma requisição `CALL` para chamar o cliente para a partida (que pode ser aceita ou recusada).

Se for aceita, ambos os clientes trocaram mensagens até a partida a acabar, enviando o tabuleiro e suas jogadas.

O cliente que chama a partida é sempre o jogador X (jogador 1) e é ele que deverá informar o resultado para o servidor.

Caso o servidor tenha caído durante este intervalo, o cliente tentará reconectar-se com o servidor para enviar uma mensagem informando o ganhador da partida. Esta tentativa de reconexão durará no máximo 3 minutos (do contrário, o cliente desiste).

Servidor

O servidor fica esperando por conexões TCP e UDP nas portas 5000 e 5001 por padrão (é possível mudar isto).

Conexões TCP e UDP são tratadas de forma igual do ponto de vista do cliente (não há diferença de tratamento na troca de mensagens), porém internamente o servidor têm uma array dos *sockets* dos clientes TCP.

Para cada cliente (TCP ou UDP), o servidor mantém seu endereço IP e porta da conexão, sua porta pública para partidas, o nome do usuário logado (se houver), e o estado atual do cliente (logado, deslogado, jogando, etc.

Servidor

Eventos importantes (inicialização/termino do servidor, conexão/desconexão de cliente, tentativas de login, início/finalização de partidas, etc) são registrados em um arquivo de texto (`log.txt` por padrão) e imprimidos no console também.

Alguns dados são armazenados de forma persistente (tais como dados sobre usuários e partidas) em um arquivo JSON para fácil leitura. Outros dados (endereço IP e estado de execução do cliente) são dados efêmeros que estão presentes somente durante o tempo de vida do servidor.

O servidor também realiza *heartbeats* periódicos, no qual checa qual foi a última vez que recebeu mensagem de um cliente. Para isso, toda vez que o cliente envia qualquer mensagem (incluindo *PING*), o servidor atualiza um timestamp associado ao cliente em questão. Se passar de 5 minutos, o cliente é desconectado.

Servidor

Aqui também utilizamos máquinas de estado para verificar se o usuário pode executar determinados comandos. Também utilizamos outros mecanismos de validação (tais como verificar se o usuário pode alterar a senha, se suas credenciais estão certas, etc).

Criamos interfaces específicas para cada tarefa do servidor (`Database`, `Logger`, `ClienteSocket`) para deixar o código mais modular. Assim, o código principal do servidor só cuida da lógica por trás das requisições, enquanto o resto é delegado para as outras classes de objetos.

É importante frisar que estamos detalhando nossa implementação do servidor, mas o servidor poderia ser implementado de várias formas diferentes desde que ele utilize o protocolo especificado pelo TTTP.

Testes

Fizemos três tipos de testes, conforme especificado no EP.

Foram feitos 5 testes para cada tipo de teste rodando o servidor e os clientes (se for o caso) dentro de um container Docker. Os dados foram coletados usando `docker stats` e filtrados usando shell scripts. Cada teste durou cerca de 100 segundos.

Para o teste do tipo 1 (servidor rodando sozinho), obtivemos um uso médio de CPU igual à 0 e uso da rede igual à 0 também.

Testes

Para o teste 2, obtivemos:

	avg cpu	dp cpu	avg net	dp net
servidor	0	0	2.48	0.96
cliente	99.97	0.35	700.03	15.90

Onde `avg` significa média, `dp` significa desvio padrão, `cpu` é o uso da CPU, e `net` é o uso da rede.

Testes

Para o teste 3, obtivemos:

	avg cpu	dp cpu	avg net	dp net
servidor	0	0	2.66	1.03
cliente	99.40	0.33	1300.01	19.39

Gráficos

Os valores variaram um pouco de teste para teste, mas os valores de uso da CPU e uso da rede não variaram quase nada durante um mesmo teste (ambos para o cliente e para o servidor).

Então, iremos apresentar apenas dois gráficos para mostrar o desempenho durante o teste (visto que os outros gráficos são praticamente uma linha horizontal devido à pouca variação dentro do mesmo teste).

Gráfico de uso do CPU pelo cliente (teste 2)

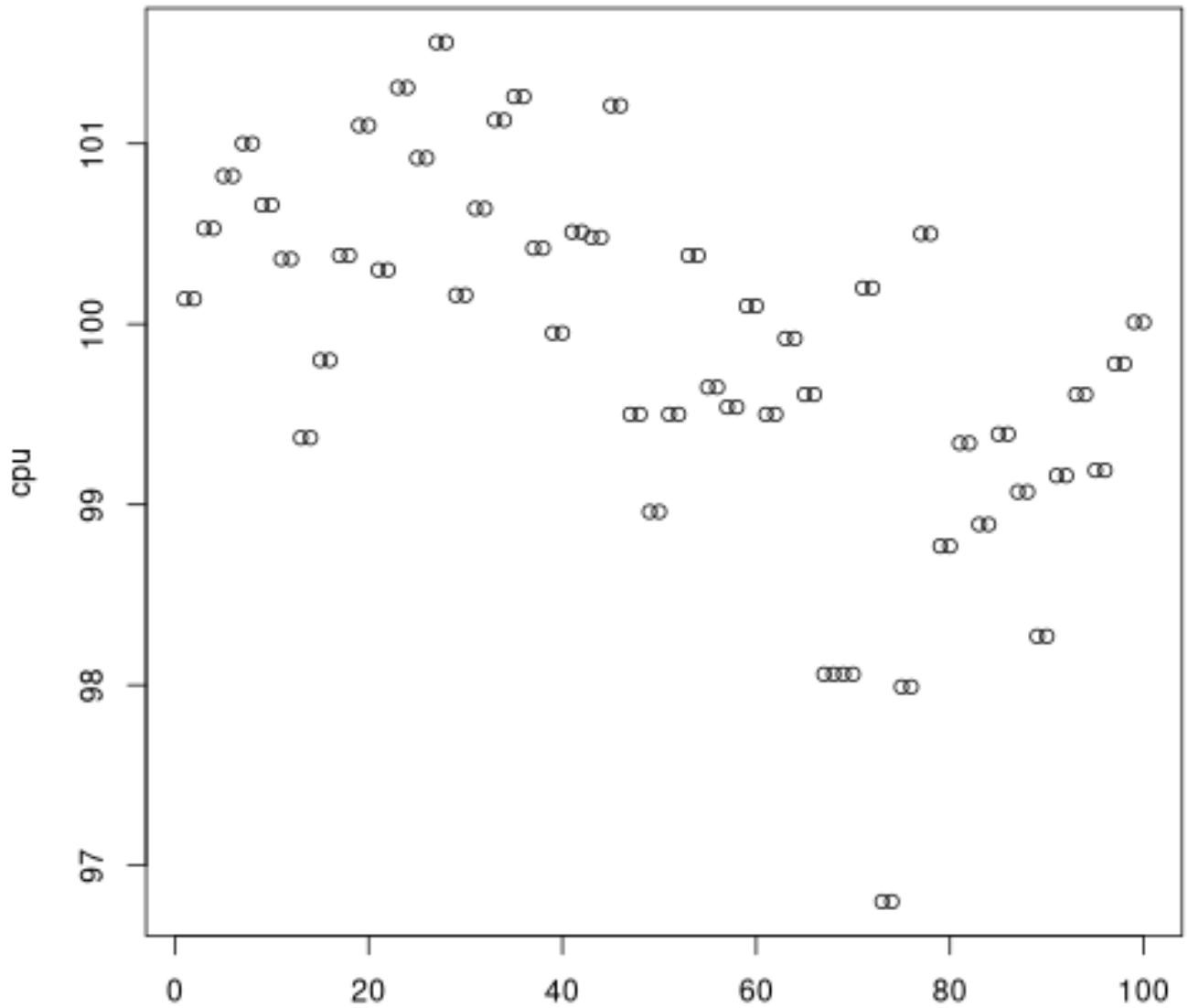
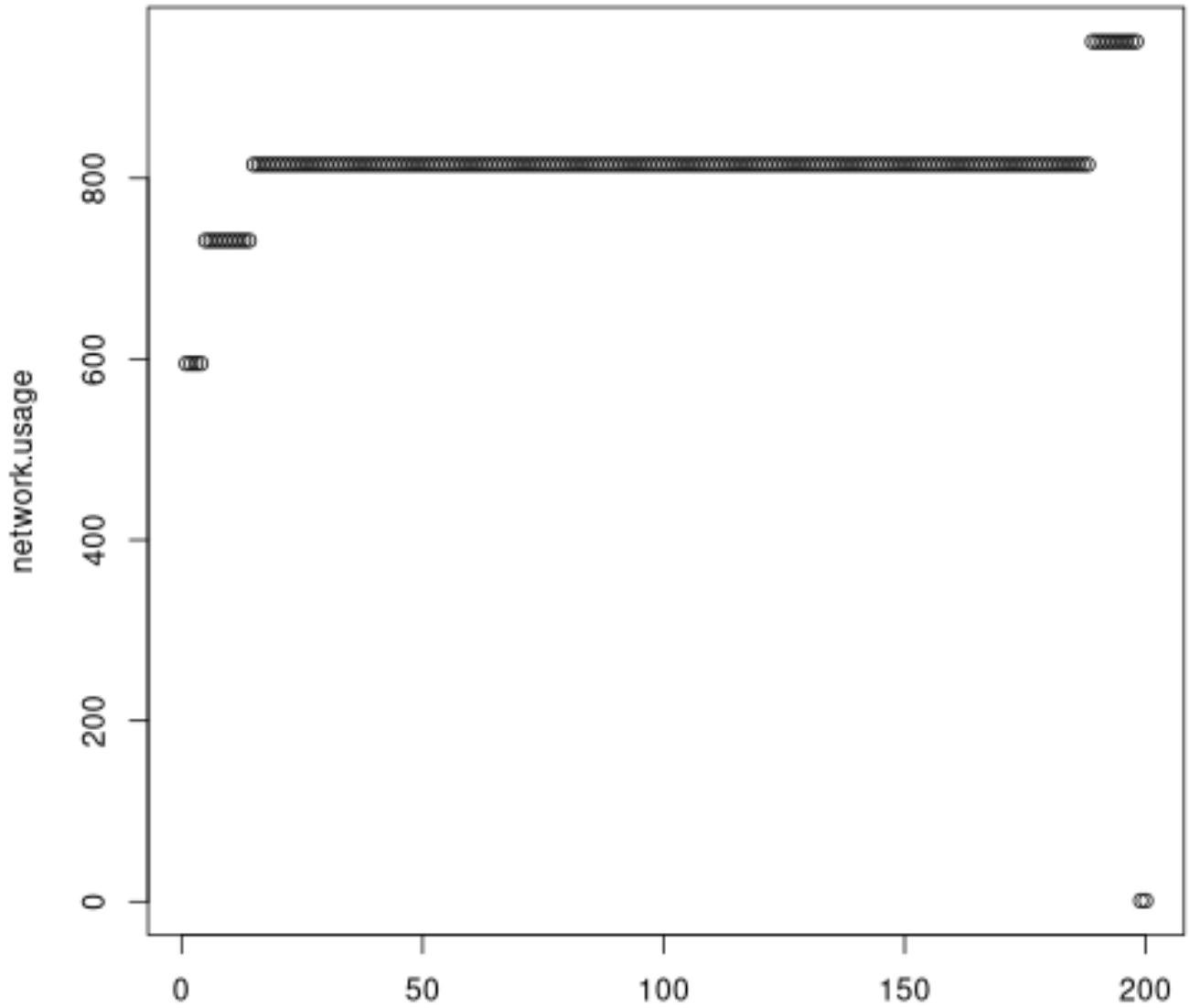


Gráfico de uso da rede pelo cliente (teste 2)



Conclusão

Pelos gráficos feitos e estatísticas observadas, podemos concluir que o servidor ficou com bom desempenho e o cliente não.

Provavelmente porque o cliente utilizou várias threads para tarefas tais como medição de latência, heartbeat, e conexão com o oponente.

Por outro lado, utilizamos a biblioteca `select` para responder à conexão de vários clientes simultaneamente sem bloquear o programa e sem usar threads. Esta biblioteca só executa o código quando algum socket tem dado para ler e no restante do tempo o servidor fica "iddle", sem executar.