

Sistemas Operacionais II N - Relatório do Trabalho Prático 2

Instituto de Informática, Universidade Federal do Rio Grande do Sul
INF01151 - Sistemas Operacionais II N 2021/1

Jonatas Tschá Santoro	00273172
Rodrigo Nogueira Wuerdig	00280361
Tiago Comassetto Fróes	00240512

November 21, 2021

- Link da Apresentação: <https://youtu.be/ODLCZsEwNLI>

A aplicação apresentada neste trabalho é dividida em duas partes: a aplicação servidor, assinalada como *Server* na Fig. 1, e a aplicação cliente, assinalada como *Client* <number> na Fig. 1 (onde <number> indica os diferentes clientes conectados ao mesmo servidor).

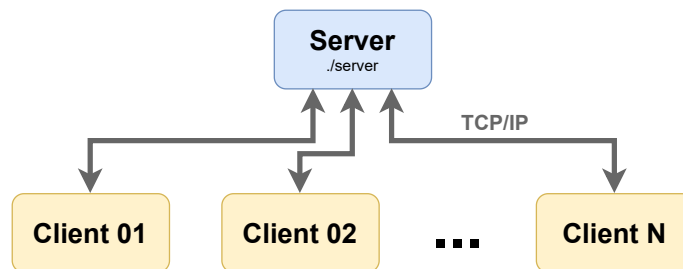


Figure 1: *Overview da Arquitetura de Interação*

O *Inter-process Communication (IPC)* é um conjunto de técnicas que podem ser utilizadas para a comunicação, i.e., troca de dados, entre processos. Diversas abordagens foram desenvolvidas com o passar dos anos para suprir diferentes requisitos de sistema, software, performance, modularidade, banda de rede e latência. Como uma aplicação de troca de mensagens prevê a utilização do mesmo em diferentes sistemas através da internet, o cliente ter acesso a uma memória compartilhada ou *pipe* na máquina *host*/servidor é extremamente improvável. Logo, deve-se optar por mecanismos *IPC* (e combinação dos mesmos) que permitam a comunicação inter-sistemas, como *sockets TCP*, *UDP* ou *SCTP*.

1 Descrição do trabalho

Para realizar a troca de informações entre as aplicações cliente e servidor foi utilizado *sockets TCP*. Dessa forma, algumas características foram garantidas como a ordem do recebimento das mensagens ou a garantia de envio.

1.1 Instalação

1.1.1 Requisitos

- Git
- C

- gcc
- Make

1.1.2 Passo a passo

A aplicação fora desenvolvida utilizando *git* e pode ser vista através do *link* no *GitHub* <https://github.com/akahenry/message-notifier/>. Para realizar a sua instalação, basta clonar o repositório:

```
$ git clone git@github.com:akahenry/message-notifier
```

Dentro do repositório existem alguns arquivos *Makefile*, mas é suficiente fazer o uso do arquivo contido no diretório raiz do repositório. Portanto, compile a aplicação cliente e servidor através:

```
$ make all
```

Se tudo ocorrer como o esperado, os binários das aplicações estarão dentro da pasta *bin*, presente no diretório raiz.

1.2 Sintaxe dos Comandos

Para executar o servidor, o usuário deve executar o seguinte comando:

```
./bin/server <port>
```

Para abrir um terminal como usuário, deve-se executar a aplicação cliente como:

```
./bin/client <username> <server-ip> <server-port>
```

Onde o campo *<username>* deve ser preenchido com um nome válido de acordo com a especificação. No campo *<server-ip>* deve ser inserido o endereço da máquina que contém a aplicação servidor em execução. Já no *<server-port>* deve ser inserida a porta a qual a aplicação servidor está utilizando para fazer a comunicação via *socket TCP/IP*.

- **FOLLOW** @username: segue o usuário *username*;
- **SEND** message: envia a mensagem *message* a todos os *followers*;
- **EXIT**: indica a finalização da sessão e término da execução do programa cliente.

2 Arquitetura das aplicações

As aplicações foram desenvolvidas para que pudessem ser executadas em máquinas distintas (sem compartilhamento de recurso físico). Dessa forma, o uso de *sockets* como meio de interação se faz extremamente benéfico. Entretanto, esse é um tipo de conexão simples que não contém algumas garantias e que devem ser tratadas adequadamente.

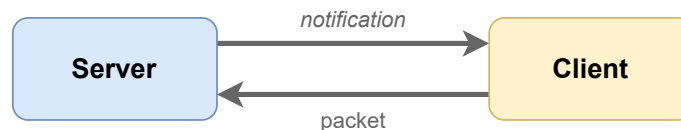


Figure 2: *Overview* da Arquitetura de Mensagens

2.1 Socket

O uso de *sockets* pode tornar o processo de desenvolvimento confuso pois não há uma indicação clara e nativa da conexão. Por este motivo, foi desenvolvida uma abstração dos *sockets* dada pela classe *Socket*. Essa classe contém informações sobre a conexão com um outro processo. Entretanto, não é necessário indicar e lidar com atributos como o descritor de arquivo. Para fazer o uso dessa classe, é suficiente entender que cada instância desse módulo se trata de uma conexão (por hora apenas *TCP*) com um outro processo. Dessa forma, é possível enviar e receber mensagens facilmente.

2.2 Cliente

A aplicação cliente é composta por três partes: *client*, *interpreter* e *listener*.

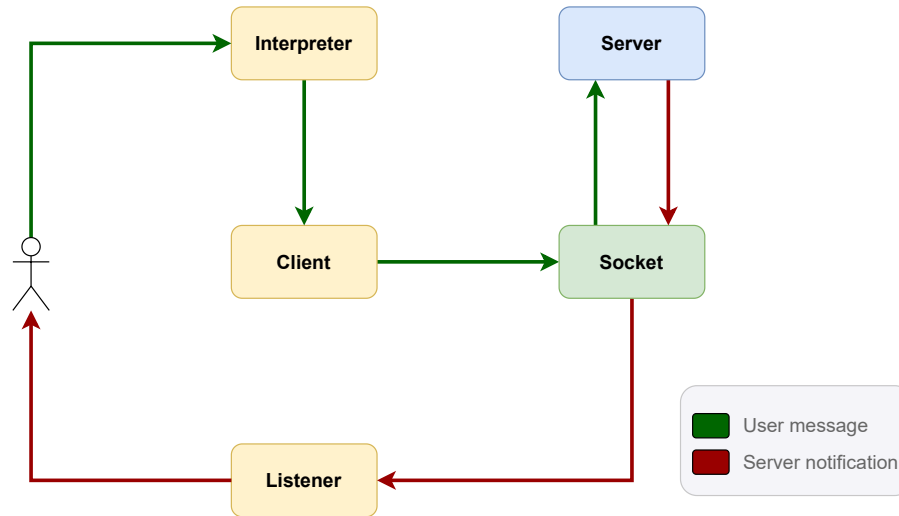


Figure 3: *Overview da Arquitetura do cliente*

2.2.1 Interpreter

O *interpreter* se trata do componente que identifica as entradas do usuário da aplicação cliente. Neste módulo será feita a análise da entrada do usuário, a adequação da mensagem e o envio para o módulo que irá tratar a mensagem adequadamente. Este módulo, portanto, ao receber a entrada "FOLLOW @pedro" do usuário, irá indicar ao próximo componente que o usuário quer seguir o usuário *pedro*.

2.2.2 Client

No módulo *client* é feito todo o controle da aplicação, configurando a conexão com o servidor, inicializando o módulo *listener* e intermediando as interações do *interpreter* com o servidor.

2.2.3 Listener

A execução do módulo *listener* ocorre paralelamente a execução da *thread* principal do programa. Através dessa concorrência, é feito o recebimento da notificação a partir do servidor, a sua interpretação e disposição para o usuário da aplicação cliente.

2.3 Servidor

A aplicação servidor foi desenvolvida através do pilar arquitetural de *produtor-consumidor* usando conceitos de orientação a eventos. Para que isso seja possível, alguns módulos foram criados, sendo eles: *packet handler*, *session*, *user* e *server*.

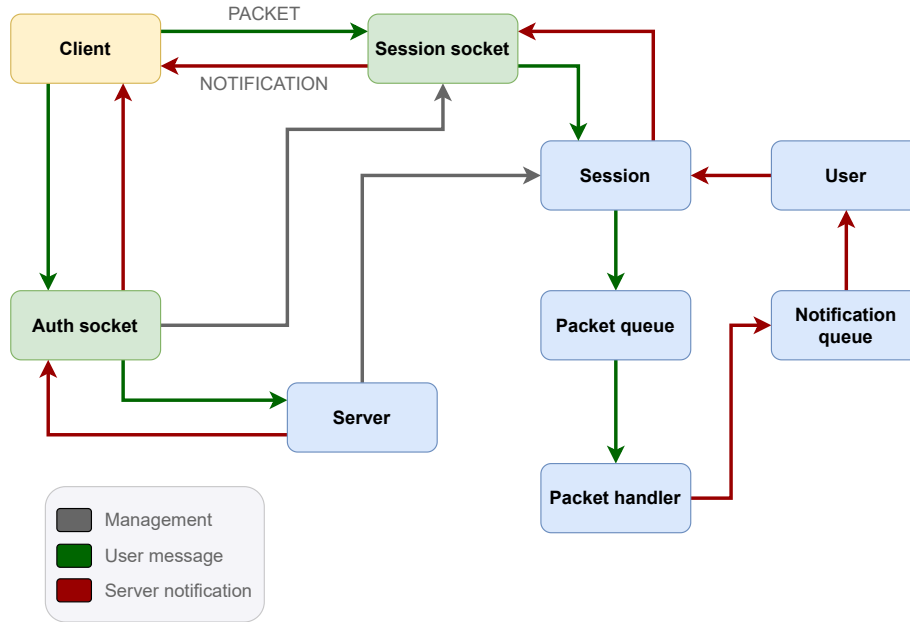


Figure 4: *Overview* da Arquitetura do servidor

2.3.1 *Packet handler*

Esse módulo compõe o consumidor dos eventos. Cada evento é uma interação de um cliente com o serviço. Por este motivo, é necessário que a ordem de recebimento dos eventos seja mantida em relação a ordem de envio desses objetos. Além da garantia através do uso do protocolo *TCP*, foi utilizado uma fila de pacotes para guardar e ordenar o tratamento desses eventos. Dessa forma, o papel desse módulo é consumir eventos dessa fila e tratá-los adequadamente, sendo que esses eventos podem ser:

- **SEND**: indica o recebimento de uma mensagem que deve ser encaminhada a todos os *followers* de um determinado usuário;
- **FOLLOW**: indica que um usuário deverá receber todas as mensagens enviadas por outro usuário (ambos definidos na mensagem);
- **EXIT**: indica o fim de uma sessão com um usuário;
- **CONNECT**: indica o início de uma sessão com usuário.

Ao receber um pacote do tipo *SEND*, o *packet handler* irá colocar na fila de notificações de cada seguidor do remetente da mensagem. No caso do *FOLLOW*, o remetente será adicionado como seguidor do usuário informado na mensagem. Por fim, o pacote de *EXIT* irá liberar uma sessão de cliente para o remetente da mensagem e fechar a conexão do *socket*.

2.3.2 *Session*

Esse módulo compõe o produtor de eventos após a primeira interação entre o cliente e o servidor. Ele contém toda a informação da conexão, sendo o único caminho para envio e recebimento de notificações entre esses dois processos.

Devido ao fato de ter sido desenvolvido sobre a arquitetura de produtor-consumidor, essa aplicação implementa uma fila encadeada utilizando o conceito de monitores para que não haja problema de acesso indevido a memória. Dessa forma, nunca haverá uma inserção simultânea a uma remoção, por exemplo.

2.3.3 User

O componente *User* contém a abstração do usuário. Assim sendo, ele que irá controlar o recebimento e envio de notificações através da fila de notificações. Além disso, este módulo possui todas as informações de sessões ativas juntamente com a indicação de *followers* e *following* do usuário respectivo.

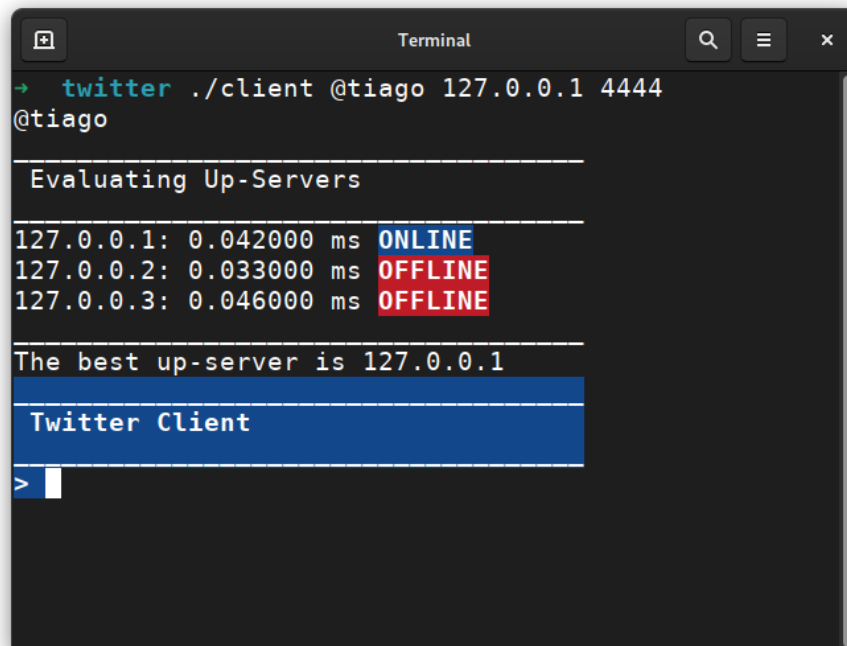
2.3.4 Server

Por fim, o *Server* é o controlador geral da aplicação. Nele será feita a primeira conexão com o cliente, sendo, a partir dele, criadas outras linhas de execução como as sessões (uma para cada cliente) ou, inicialmente, o *packet handler*. Por este motivo, ele também é o módulo que realiza o controle de finalização da aplicação, mandando as mensagens de finalização para os clientes em caso de término de execução do servidor.

Além disso, esse componente contém, juntamente ao *packet handler*, o mapa de usuários, onde é guardado todos os endereços de memória que contém os usuários. Como há um compartilhamento de memória entre essas duas classes, é necessário que haja algum tipo de sincronização de interação. Para isso, o comportamento dessa estrutura também se assemelha a um monitor, sendo impossível acessar, remover ou editar usuários simultaneamente.

3 Replicação

Para o algoritmo de replicação, foi utilizado uma métrica de Quality-of-Service (QoS) onde é avaliado a melhor conexão (latência) entre os servidores online.



```
→ twitter ./client @tiago 127.0.0.1 4444
@tiago

Evaluating Up-Servers

127.0.0.1: 0.042000 ms ONLINE
127.0.0.2: 0.033000 ms OFFLINE
127.0.0.3: 0.046000 ms OFFLINE

The best up-server is 127.0.0.1

Twitter Client

> |
```

Figure 5: Conexão de um cliente quando há apenas um servidor online.

Servidor	Melhor para	Líder
127.0.0.1	N1 (ex. N1=1)	Replica
127.0.0.2	N2 (ex. N2=2)	Líder
127.0.0.3	N3 (ex. N3=0)	Replica

Table 1: Tabela para Eleição de Líderes. N2 é líder já que $N2 > N1 > N3$.

```

→ twitter ./client @tiago 127.0.0.1 4444
@tiago

Evaluating Up-Servers

127.0.0.1: 0.065000 ms ONLINE
127.0.0.2: 0.055000 ms ONLINE
127.0.0.3: 0.031000 ms OFFLINE

The best up-server is 127.0.0.2

Twitter Client

>

```

Figure 6: Conexão de um cliente quando há dois servidores online.

A eleição do líder e replicas é feita com base no voto de QoS dos usuários. Seguindo um padrão de tabela similar a apresentada na Tab.1. Onde, supondo um usuário 1, ele avalia todos os servidores online e o melhor para ele foi o "127.0.0.2". Então o mesmo vai enviar a informação para fazer incremento da variável N2, conforme Tab.1. Porém, isso não diz que ele vá se conectar no "127.0.0.2", o mesmo vai conectar no servidor online onde há mais votos.

Quando só há um servidor online, ele é o líder porém mesmo assim é feito o incremento de sua variável. Pois caso outro servidor entre em operação no tempo ele vai ter uma avaliação justa.

4 Dificuldades

Durante o desenvolvimento do trabalho alguns problemas surgiram, desde problemas relacionados a conexão e padronização de interação até problemas de armazenamento e interpretação.

4.1 Conexão

4.1.1 Autenticação

Um problema de conexão que tivemos foi o de explicitar a primeira interação de um cliente com o servidor. Dessa forma, o cliente precisa, além de fazer a conexão com o *socket*, precisa enviar uma mensagem ao servidor perguntando se ele autoriza a conexão. Para resolver essa situação, foi necessário implementar um novo tipo de pacote chamado *CONNECT*. Esse pacote indica qual é o usuário do cliente que está tentando conectar ao servidor. Dessa forma, o servidor consegue

identificar e barrar novas conexões de usuários que já possuem o número máximo de sessões ativas no momento. Entretanto, isso implica em um comportamento padrão para que clientes conectem no servidor, visto que será necessário um passo a mais (mesmo que invisível) para que o usuário possa usar o cliente e enviar mensagens.

4.1.2 *Graceful shutdown*

Assim como na primeira conexão, a última conexão também foi problemática pois o cliente deve parar o seu funcionamento assim que o servidor finalizar (ou recusar a conexão de um cliente). Para isso, foi implementado um indicador para que o cliente identificasse possíveis problemas na interação com o servidor e pudesse finalizar sua execução graciosamente. Da mesma forma, a finalização da execução do cliente deve ser feita após indicar ao servidor que o cliente está fechando (para que o servidor possa realocar a sessão do usuário para outro cliente) e, para isso, foi utilizado um novo tipo de pacote chamado de *EXIT*, indicando esse comportamento.

O servidor tem uma particularidade no seu *shutdown* devido a forma com que fora implementado o envio de notificações: como as mensagens são enfileiradas e enviadas de acordo com a execução do *packet handler*, o servidor pode identificar um sinal de interrupção, por exemplo, para finalizar sua execução com a sua fila de pacotes não vazia. Isso significa que, caso a aplicação termine diretamente, o servidor perderá todos os eventos associados aos usuários. Além disso, como o servidor ainda não finalizou, os clientes ainda podem enviar e receber mensagens, o que é problemático pois o servidor pode nunca finalizar sua execução se continuar recebendo mensagens ininterruptamente. Dessa forma, visando solucionar esse problema, o servidor envia uma mensagem aos clientes ativos para informá-los que qualquer mensagem enviada a partir deste momento será ignorada. Portanto, o servidor não precisa fechar a conexão com os clientes e pode terminar de tratar os pacotes presentes na fila e finalizar o envio das notificações pendentes aos usuários com ao menos uma sessão ativa (embora seja opcional, visto que as notificações pendentes são salvas para serem reestabelecidas na reinicialização do servidor) antes do término de sua execução.

4.2 Armazenamento

A especificação do trabalho indica que é necessário armazenar o estado do servidor após finalizá-lo de modo que seja possível reestabelecer o mesmo estado numa segunda inicialização. Para isso, precisamos guardar as informações de todos os usuários e suas notificações pendentes. Entretanto, a maneira com que fora implementada a conexão de usuários e seus *followers* se deu através de uso de ponteiros e endereços de memória. Assim sendo, não é possível armazenar diretamente os usuários visto que não há garantia que os mesmos usuários irão possuir os mesmos endereços de memória em duas inicializações distintas. Além disso, a referência é mútua, então o usuário possui referências aos usuários que ele segue e referências aos usuários que ele está seguindo (o que dificultou ainda mais esta implementação). Para resolver este problema, foi necessário armazenar apenas os dados primordiais dos *followers* no momento de referenciá-los nos arquivos de cada usuário, criando-os e populando-os de acordo com a leitura de cada usuário individualmente.

5 *Hardware* utilizado

Para desenvolver e executar a aplicação, fora utilizado uma máquina com as seguintes especificações:

- **Sistema operacional:** *Zorin OS 16 (Ubuntu based)*
- **Processador:** *AMD Ryzen 5 3600*
- **Memória RAM:** *2x 8GB DIMM DDR4 2800MHz*
- **Compilador:** *g++ 9.3.0*

A aplicação também foi testada em outras configurações:

- **Sistema operacional:** *Fedora 34*

- **Processador:** *AMD FX-6300*
- **Memória *RAM*:** *4x 4GB DIMM DDR3 1600MHz*
- **Compilador:** *g++ 9.3.0*