

Um Sistema de Compartilhamento de Mensagens e Conteúdos multimídias Orientado a Eventos

[Introdução](#)

[Problema](#)

[Protocolo](#)

[Campos comuns a todas as mensagens](#)

[Tipos de mensagens](#)

[Alocação de identificadores](#)

[Associação entre emissores e exibidores](#)

[Detalhes de implementação](#)

[Uso de sockets TCP](#)

[Parâmetros de execução](#)

[Clientes: emissores e exibidores](#)

[Servidor](#)

[Controle de recebimento de mensagens](#)

[Entrega](#)

[Avaliação](#)

[Dicas e cuidados a serem observados](#)

[Exemplo](#)

Introdução

A proliferação de programas interativos de mensagens e conteúdos multimídias instantâneos em redes é um fato real e de acesso para todos, e a cada dia novos programas surgem. Esses programas atuam em diferentes contextos profissionais e pessoais propagando de modo instantâneo textos, imagens e vídeos. Entre os mais populares estão Whatsapp e Instagram, por exemplo.

O objetivo deste trabalho é especificar e implementar um protocolo para Compartilhamento de Mensagens e Conteúdos multimídias Orientado a Eventos em nível de aplicação, utilizando interface de *sockets*. O trabalho envolve as seguintes etapas:

1. Definição do formato das mensagens de controle e do protocolo.
2. Implementação utilizando *sockets* em C, C++, Java ou Python em uma arquitetura orientada a eventos (**centrada ao redor da função *select* ou de múltiplas *threads***). **O servidor deve ser capaz de tratar conexões simultâneas de vários clientes.**
3. Escrita do relatório.

Problema

Você desenvolverá programas para um sistema simples de troca de mensagens de texto e arquivos multimídia utilizando apenas funcionalidades equivalentes às da biblioteca de *sockets* POSIX, utilizando comunicação via protocolo TCP. O código do servidor deverá ser organizado ao redor da função *select* ou deve utilizar múltiplas *threads*, usando uma técnica conhecida como *orientação a eventos*.

Três programas devem ser desenvolvidos: um programa **servidor**, que será responsável pelo controle da troca de mensagens, um programa **exibidor** para exibição das mensagens recebidas e um programa **emissor** para envio de mensagens para o servidor. Os programas exibidores e emissores, também chamados de clientes, se comunicam por intermédio do programa servidor. Cada programa cliente se identifica com um valor inteiro único no sistema, alocado pelo servidor. Programas emissores podem enviar mensagens de texto para todos os programas exibidores (*broadcast*) ou apenas para um programa exibidor (*unicast*).

Protocolo

O protocolo de aplicação deverá funcionar sobre TCP. Isso implica que as mensagens serão entregues sobre um canal de bytes com garantias de entrega em ordem, mas é sua responsabilidade determinar onde começa e termina cada mensagem.

Campos comuns a todas as mensagens

Cada mensagem do protocolo de comunicação possui um cabeçalho com os seguintes campos:

- **Tipo da mensagem (2 bytes):** um dos 7 tipos de mensagens definidos abaixo.
- **Identificador de origem (2 bytes):** Cada mensagem carrega o identificador da origem (ou um valor pré-definido, no caso da mensagem OI, a seguir).

- **Identificador de destino (2 bytes):** Cada mensagem carrega também o identificador do destino.
- **Número de sequência (2 bytes):** Cada mensagem enviada por um programa deve receber um número de sequência, local ao programa. A primeira mensagem deve ter número de sequência zero e as mensagens seguintes devem ter o número de sequência da mensagem anterior mais 1.

Tipos de mensagens

O protocolo possui os seguintes tipos de mensagem. Todos os tipos de mensagem carregam um cabeçalho com todos os campos definidos anteriormente. O número após cada tipo é o valor do campo "Tipo da mensagem" na lista anterior.

- **OK (1):** Todas as mensagens do protocolo devem ser respondidas com uma mensagem de OK ou ERRO. Essa mensagem funcionará como uma confirmação. As mensagens de OK devem carregar o número de sequência da mensagem que está sendo confirmada. O envio de uma mensagem de OK não incrementa o número de sequência das mensagens do cliente (mensagens de OK não têm número de sequência próprio).
- **ERRO (2):** Idêntica à mensagem OK, mas enviada em situações onde uma mensagem não pode ser aceita, por qualquer motivo. Essa mensagem também é uma espécie de confirmação, porém utilizada para indicar que alguma coisa deu errado.
- **OI (3):** Primeira mensagem de um programa cliente (emissor/exibidor) para se identificar para o servidor. Apenas clientes enviam essa mensagem. O destinatário é sempre o servidor. Como o cliente envia essa mensagem antes de saber qual é o seu identificador, ele deve preencher o identificador de origem com o valor zero se ele é um exibidor; um valor diferente de zero identifica um emissor. Se esse valor estiver entre 2^{12} e $(2^{13} - 1)$, ele identifica um exibidor (que já deve estar em execução) e que deve ser associado aquele emissor. Se o servidor aceitar a mensagem do cliente, ele envia uma mensagem OK contendo no campo de destino o identificador que deverá ser usado a partir daí pelo cliente. Todo cliente deve exibir uma mensagem informando o identificador recebido do servidor.
- **FLW (4):** Última mensagem de/para um cliente para registrar sua desconexão e saída do sistema. A partir dessa mensagem o servidor/cliente que recebe a mensagem envia um OK de volta e fecha a conexão. Se o servidor recebe a mensagem de um emissor, ele deve verificar se há um exibidor associado (identificado por ele na mensagem OI). Se houver, ele (servidor) envia uma mensagem FLW para aquele exibidor. Se um exibidor recebe a mensagem, ele deve terminar sua execução depois de responder.
- **MSG (5):** Uma mensagem é originada por um emissor, enviada ao servidor, e repassada pelo servidor para um ou mais exibidores. O emissor deve colocar no campo destino o identificador do exibidor para o qual a mensagem deve ser repassada, ou de um emissor associado a um exibidor. Se o campo destino possuir o valor zero, o servidor irá enviar a mensagem para todos os exibidores (broadcast). Se o campo destino possuir o identificador de um emissor, a mensagem deve ser enviada para o exibidor associado a ele (se existir) ou um ERRO deve ser retornado.

Ao repassar uma MSG, todos os campos, inclusive os campos origem e destino, devem permanecer inalterados. Uma MSG começa com um inteiro (2 bytes, *network byte order*) logo após o cabeçalho, indicando o número de caracteres sendo transmitidos, C. Depois do inteiro, seguem C bytes contendo os caracteres da mensagem em ASCII. Note que o valor C determina quantos bytes a mais devem ser lidos.

- **CREQ (6):** Enviada pelo emissor para o servidor. Indica no campo de destino um receptor para o qual deve ser enviada a lista de clientes (emissores e exibidores) que estão conectados ao sistema. O servidor deve responder com um OK para o emissor da mensagem CREQ. O servidor deve também enviar uma mensagem do tipo CLIST para o destinatário indicado na mensagem CREQ. (Opcional: seu servidor pode suportar broadcast de CLIST para todos os exibidores caso o campo destino do CREQ tenha o valor zero.)
- **CLIST (7):** Essa mensagem possui um inteiro (2 bytes, *network byte order*) indicando número de clientes conectados, N. A mensagem CLIST possui também uma lista de N inteiros (2 bytes cada, todos em *network byte order*) que armazena os identificadores de cada cliente (exibidor e emissor) conectados ao sistema. Note que o valor N determina quantos valores a mais devem ser lidos. O remetente dessa mensagem é sempre o servidor e o destinatário é um cliente informado como destinatário da mensagem CREQ. O cliente deve responder uma mensagem CLIST com uma mensagem OK.
- **FILE (8):** Essa mensagem é utilizada para o início da transmissão de arquivos multimídia, como fotos, áudio e vídeos. Ela é originada por um emissor, enviada ao servidor, e repassada pelo servidor para um ou mais exibidores. Cada mensagem se inicia com um inteiro ID (2 bytes, *network byte order*) que representa um identificador único para o arquivo (local ao programa emissor). Na sequência, é enviado outro inteiro N (2 bytes, *network byte order*) que representa a quantidade de *chunks* em que o arquivo foi dividido pelo emissor para ser enviado ao exibidor. Por fim, é enviado um inteiro LEN_EXT (2 bytes, *network byte order*) seguido por LEN_EXT bytes contendo os caracteres da extensão EXT do arquivo em ASCII. Cada *chunk* é enviado em uma mensagem do tipo FILE_CHUNK. Após receber uma mensagem do tipo FILE, o exibidor deve esperar por N mensagens do tipo FILE_CHUNK para montar o arquivo com identificador ID. O arquivo deve ser salvo em um diretório com um nome que siga um padrão específico (sugestão: IdExibidor_IdEmissor_IdArquivo.EXT).
- **FILE_CHUNK (9):** A transmissão do arquivo propriamente dita é realizada através de *file chunks*. Cada mensagem desse tipo se inicia com três inteiros (2 bytes cada, todos em *network byte order*) que representam o identificador do arquivo (ID), o identificador do *chunk* (CID) e o tamanho do *chunk* (LEN). Por fim, cada mensagem possui LEN bytes de dados. O emissor deve aguardar a confirmação de um FILE_CHUNK antes de enviar o próximo. Após receber o último *chunk*, o arquivo deve ser salvo em disco.

Alocação de identificadores

O servidor aloca identificadores entre 1 e $(2^{12} - 1)$ para emissores e identificadores entre 2^{12} e $(2^{13} - 1)$ para exibidores. Essa distinção entre identificadores de emissores e exibidores

tem o objetivo de facilitar o encaminhamento das mensagens pelo servidor. O servidor tem identificador ($2^{16} - 1$).

Associação entre emissores e exibidores

Não há obrigatoriedade de que um emissor esteja sempre associado a um emissor. Podem haver emissores sem um exibidor, bem como exibidores sem um emissor. A associação acontece no momento da conexão do emissor, se o identificador fornecido for um exibidor presente no sistema. É responsabilidade do servidor manter o controle sobre quem está associado a quem, quando for o caso. Isso pressupõe que a informação sobre a identificação dos exibidores é manipulada pelo usuário: primeiro ele dispara um exibidor e observa o identificador que ele recebe do servidor e exibe na sua saída; em seguida o usuário deve disparar o emissor, passando como parâmetro para ele o identificador que foi informado pelo exibidor.

Detalhes de implementação

Pequenos detalhes devem ser observados no desenvolvimento de cada programa que fará parte do sistema. É importante observar que o protocolo é simples e único, de forma que programas de todos os alunos deverão ser interoperáveis, funcionando uns com os outros.

Uso de sockets TCP

Como mencionado anteriormente, a implementação do protocolo da aplicação utilizará TCP. Haverá apenas um socket em cada cliente, independente de quantos outros programas se comunicarem com aquele processo. O programador deve usar as funções *send* e *recv* para enviar e receber mensagens. No caso do servidor, ele deve manter um socket para receber novas conexões (sobre o qual ele executará *accept*) e um socket para cada cliente conectado.

Parâmetros de execução

O servidor deve ser iniciado recebendo como parâmetro apenas o número do porto onde ele deve ouvir por conexão dos clientes. Um exibidor deve ser disparado com um parâmetro obrigatório que identifica a localização do servidor como um par de endereço "IP:porto". Um emissor deve ser disparado com um parâmetro obrigatório que identifica a localização do servidor como um par de endereço "IP:porto", que pode ser seguido por um segundo parâmetro, opcional, com o identificador de um exibidor, caso uma associação deva ser feita.

Clientes: emissores e exibidores

Emissores e exibidores desempenham papéis diferentes, sendo que juntos compõem as duas partes do que seria uma interface completa de um usuário no sistema. Emissores recebem mensagens do teclado e as enviam para o servidor, enquanto exibidores recebem mensagens do servidor e as exibe na tela.

Depois de sua inicialização, o emissor executa um *loop* simples, lendo linhas do teclado, formatando-as como mensagens e enviando-as para o servidor. Sua interface deve oferecer uma forma do usuário identificar o destino (um exibidor específico ou todos os exibidores no sistema), bem como permitir que o usuário indique que deseja montar uma mensagem CREQ, identificando o exibidor que deve recebê-la. Deve também haver uma forma de indicar ao programa que ele deve terminar (quando uma mensagem FLW deve ser enviada para o servidor).

Depois de sua inicialização, o exibidor também executa um *loop* simples, esperando por mensagens do servidor e exibindo-as na tela. Ele também deve ser capaz de interpretar as mensagens CLIST para apresentar a informação para o usuário. O programa exibidor deve exibir a informação de identificação de quem enviou a mensagem (algo como "Mensagem de 42: Oi, como vai?").

Essas são todas as considerações pré-definidas sobre a parte dos clientes. Sinta-se livre para decidir (e documentar!) qualquer decisão extra. Em particular, é sua tarefa definir a interface de interação do usuário com cada programa.

Servidor

Um sistema simples de mensagens de texto apresenta apenas um processo servidor ou repetidor e sua função é distribuir as mensagens de texto dos emissores para os exibidores a ele conectados. O servidor deve repassar mensagens entre os programas que se identificam por mensagens OI, desconectando os que enviem mensagens FLW.

O código do servidor deverá ser organizado ao redor da função *select* (ou de múltiplas *threads*), que permite a observação de diversos *sockets* em paralelo. O programa deve montar um conjunto de descritores indicando todos os *sockets* dos quais espera uma mensagem (inclusive o *socket* usado para fazer o *accept*, que deve ser um só). A função permite indicar *sockets* também onde se deseja escrever ou onde se procura por alguma exceção, mas esses dois conjuntos não interessam neste trabalho. Também não é necessário usar o temporizador que pode ser associado ao *select*. Uma vez chamada, a função *select* bloqueia até que algo ocorra nos *sockets* que foram indicados na chamada. Ao retornar, o conjunto indica quais *sockets* tem operações pendentes.

Ao receber qualquer mensagem, o servidor deve primeiro confirmar que o identificador de origem corresponde ao do cliente que a enviou, verificando se o cliente indicado na origem e o cliente que está conectado no *socket* onde a mensagem foi recebida. Esse teste evita que um cliente se passe por outro.

Depois disso, o servidor deve observar o identificador de destino contido na mensagem. Se o identificador de destino indicado na MSG for zero, a mensagem deve ser enviada a todos os exibidores conectados ao sistema. Caso o identificador seja diferente de zero, ele deve verificar se o valor indica um emissor ou um exibidor. Se for um exibidor, o servidor deve procurar pelo registro de um exibidor com o valor indicado e repassar a MSG apenas para aquele cliente. Se for um emissor, o servidor deve verificar se há um exibidor associado a ele e enviar a mensagem para aquele cliente. Caso um exibidor não seja encontrado, o servidor deve responder ao emissor com uma mensagem ERRO, caso contrário uma mensagem OK. Ao enviar uma mensagem MSG a qualquer exibidor, o servidor deve esperar pela mensagem OK em resposta apenas para confirmar que o cliente estava ativo e exibiu corretamente a mensagem. **O servidor deve ser capaz de lidar com pelo menos 255 clientes, ou seja, deve tratar conexões simultâneas.**

Ao receber uma mensagem CREQ de um emissor, o servidor deve consultar seu estado e determinar quais clientes (emissores e exibidores) estão ativos e criar uma mensagem do tipo CLIST com o contador e lista de identificadores de clientes. Como mencionado, CLIST será enviada para um exibidor identificado pelo emissor.

Controle de recebimento de mensagens

Como mencionado anteriormente, toda mensagem deve ser respondida com uma mensagem OK ou ERRO. Isso tem um efeito de confirmar se a mensagem foi processada corretamente pelo servidor. Note que ela não é necessária para confirmação da entrega, já que TCP é usado. Ela é necessária apenas para a confirmação no nível da aplicação de que a mensagem foi aceita e processada (ou não).

Entrega

Cada aluno deve entregar junto com o código um relatório de até 4 páginas (duas folhas), sem capa, utilizando fonte tamanho 10, e figuras de tamanho adequado ao tamanho da fonte. Ele deve conter uma descrição da arquitetura adotada para o servidor, os refinamentos das ações identificadas no mesmo, as estruturas de dados utilizadas e decisões de implementação não documentadas nesta especificação. Devem ser discutidos, também, os desafios, dificuldades e imprevistos do projeto, bem como as soluções adotadas para os problemas. Como sugestão, considere incluir as seguintes seções no relatório: introdução, arquitetura, emissor, exibidor, servidor, discussão e conclusão. O relatório deve ser entregue em formato PDF. A forma de modularização do código fica a seu critério, mas é importante descrever no relatório como compilar, executar e utilizar seus programas.

Avaliação

Este trabalho deve ser realizado individualmente e deve ser implementado em uma das seguintes linguagens: C, C++, Java ou Python, utilizando as bibliotecas *sockets* das respectivas linguagens. Seu programa deve rodar no sistema operacional Linux e, em particular, não deve utilizar bibliotecas do Windows, como o winsock. Procure escrever seu código de maneira clara, com comentários pontuais e bem indicados; isto facilita a correção dos monitores e tem impacto positivo na avaliação.

- **Será adotada média harmônica entre as notas da documentação e da execução, o que implica que a nota final será 0 se uma das partes não for apresentada.**
- **Não será permitida a entrega do trabalho após a data de entrega especificada.**

Dicas e cuidados a serem observados

- O guia de programação em rede do Beej (<http://beej.us/guide/bgnet/>) e o Python Module of the Week (<https://pymotw.com/2/select/>) tem bons exemplos de como organizar um servidor com select.

- Poste suas dúvidas no fórum do Moodle de forma publicamente acessível para os outros alunos.
- Procure escrever seu código de maneira clara, com comentários pontuais e bem indentado.
- Consulte o monitor antes de usar qualquer módulo ou estrutura diferente dos indicados.
- Não se esqueça de conferir se seu código não possui erros de compilação ou de execução.
- Implemente o trabalho por partes. Por exemplo, crie o formato da mensagem e tenha certeza que o envio e recebimento da mesma está correto antes de se envolver com o sistema de mensagens.

Exemplo

Terminal 1 - Emissor (ID = 1)

```
> send msg to 4096 "Oi, como vai?"
> send msg to 4097 "Bom dia!"
> send msg to 0 "Até mais tarde!"
> send file to 4096 /home/Aluno/enviar/video.mp4
> send file to 0 /home/Aluno/enviar/foto.png
> send file to 4097 /home/Aluno/enviar/audio.mp3
```

Terminal 2 - Exibidor (ID = 4096)

```
< msg from 1: "Oi, como vai?"
< msg from 1: "Até mais tarde!"
< received file from 1: /home/Aluno/recebidos/4096_1_0.mp4
< received file from 1: /home/Aluno/recebidos/4096_1_1.png
```

Terminal 3 - Exibidor (ID = 4097)

```
< msg from 1: "Bom dia"
< msg from 1: "Até mais tarde!"
< received file from 1: /home/Aluno/recebidos/4097_1_1.png
< received file from 1: /home/Aluno/recebidos/4097_1_2.mp3
```