Relatório de Projeto

Um sistema de Chat com envio de arquivos

Erik de Godoy Perillo – RA 135582

Resumo

Neste relatório, serão descritas as funcionalidades e decisões de implementação do sistema de Chat para a disciplina MC833 – Laboratório de Redes.

Universidade Estadual de Campinas

28 de junho de 2016

Sumário

1 Introdução **2**

2 Uso **2**
2.1 Instalação 2
2.2 Execução 2
2.3 Comandos disponíveis 2

3 Detalhes de Projeto **3**
3.1 Arquitetura 3
3.1.1 Chat 4
3.1.2 Servidor 5
3.1.3 Cliente 5
3.2 Rede 5
3.2.1 Abstrações 6
3.2.2 Comunicação 6

1 Introdução

A aplicação é um sistema de chat cliente/servidor. São suportadas conversas entre dois usuários ou em grupos. Pode-se também enviar/receber arquivos, com confirmação de recebimento. Mensagens/arquivos enviados são guardados no servidor até que o usuário destino esteja *online* para recebê-las. Os usuários que enviaram as mensagens/arquivos são notificados assim que as mensagens são marcadas para envio e enviadas ao destino.

2 Uso

2.1 Instalação

Basta, do diretório raiz do projeto, entrar no diretório `src` e invocar `make`. Não foram usadas bibliotecas adicionais além das padrão do C++14 e `pthread`.

2.2 Execução

É necessário primeiro ter o servidor rodando em algum lugar:

```
$ server <porta>
```

Onde `<porta>` é o número da porta onde deseja-se servir. Após isso, é só utilizar um cliente do *chat*. Basta usar:

```
$ client <ip_servidor> <porta_servidor> <nome>
```

Onde `<ip_servidor>` é o IP do servidor do chat e `<porta_servidor>` é a porta onde ele funciona. `<nome>` é o seu nome de usuário. Caso o nome de usuário ainda não exista e tudo ocorra bem, o servidor responderá com `OK`.

2.3 Comandos disponíveis

Os comandos disponíveis são os seguintes (não são diferenciadas maiúsculas de minúsculas nos nomes dos comandos):

- `help` – Mostra uma mensagem com os comandos suportados.
- `send <usuario> <mensagem>` – Envia ao usuário `<usuario>` a mensagem `<mensagem>`.
- `createg <nome_grupo>` – Cria um grupo nomeado `<nome_grupo>`.
- `joing <nome_grupo>` – Entra no grupo `<nome_grupo>`.

- `sendg <nome_grupo> <mensagem>` – Envia `<mensagem>` a todos os usuários do grupo `<nome_grupo>`.
- `sendf <usuario> <arquivo>` – Envia o arquivo localizado em `<arquivo>` para o usuário `<usuario>`.
- `who` – Lista todos os usuários do chat, assim como o *status* de cada um (*online/offline*).
- `accept <usuario>:<arquivo>` – Aceita o arquivo `<arquivo>` enviado por `<usuario>`, se há algum.
- `exit` – Sai do chat.

Para cada comando, a resposta do servidor é interpretada e é mostrado na tela uma mensagem informativa sobre o que aconteceu.

3 Detalhes de Projeto

O projeto foi implementado usando-se a linguagem C++, lidando com os aspectos de rede com bibliotecas da linguagem C e abstraindo seus detalhes.

3.1 Arquitetura

Foi escolhido para implementação do projeto a arquitetura MVC[1], pela modularização que ela proporciona, que é muito adequada para lidar com sistemas onde há múltiplas iterações ao mesmo tempo. A figura 1 ilustra a arquitetura geral do projeto e as iterações que acontecem entre cada módulo.

[1] do inglês: *Model-View-Controller*

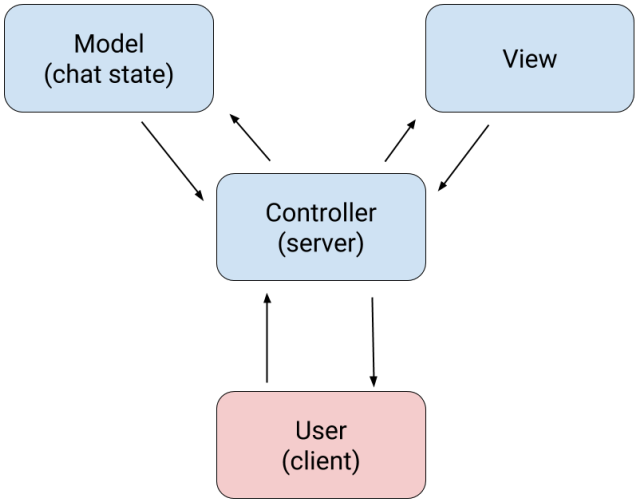

Figura 1: A arquitetura geral do projeto

3.1.1 Chat

O componente *model* do sistema é a classe *Chat*, que mantém o tempo todo o estado inteiro do chat. No modelo, há informações sobre:

- **Usuários** – Representados por uma classe *User*, que contém informações sobre seu nome no chat e endereço da Internet (representado por uma classe *NetAddr* que será discutida mais a fundo em breve).
- **Grupos** – Representados por uma classe *Group*, que contém informação sobre o nome do grupo e os usuários nele inseridos.
- **Mensagens** – Representadas por uma classe *Message*, que representa uma mensagem no chat, com seu usuário fonte, destino e o corpo da mensagem. Pode também representar um arquivo. Nesse caso, os bytes do arquivo são colocados no campo da mensagem. Para o cálculo das funções *hash*, usou-se uma especialização do método *hash* provido pela biblioteca padrão do C++. Ela detecta se a mensagem em questão é uma mensagem comum ou um arquivo: se for uma mensagem comum, faz uma combinação usando o destino, fonte e mensagem. Se for um arquivo, faz a combinação usando o destino, fonte e o nome do arquivo.

Para se manter o estado dos usuários *online* e *offline*, simplesmente cria-se grupos *online* e *offline* e, então, os usuários são inseridos/removidos desses grupos conforme necessário.

A classe *ChatView* é o componente *View* do sistema. É através dela que se obtém representações visuais sobre o estado do chat. É por ela, por exemplo, que é construída a tabela de usuários do *chat* que é enviada para os clientes que a pedirem.

3.1.2 Servidor

O servidor do chat é composto de três tipos de *threads* principais:

- **Despachadora** – Fica ouvindo por requisições de conexões novas e, caso a conexão se estabeleça e haja *threads* disponíveis, cria uma nova *thread* de interação com o usuário novo.
- **Iteração com o usuário** – Permanece esperando por novas requisições de um certo usuário e, quando a recebe, toma a ação apropriada.
- **Manipulação de mensagens** – Checa continuamente por mensagens novas a serem enviadas. Para cada mensagem na fila de envio, tenta enviar a mensagem para o destino e, em sucesso, notifica quem enviou a mensagem que ela foi entregue.

Desse modo, múltiplos clientes podem ser atendidos ao mesmo tempo (cada um com uma thread de interação dada pelo servidor) e as mensagens são manipuladas de forma assíncrona.

3.1.3 Cliente

O cliente é composto de duas *threads*:

- **Recebedora de comandos** – Essa thread continuamente recebe comandos do console e manda a requisição apropriada ao servidor.
- **Notificações** – Nessa thread, fica-se escutando por mensagens do servidor e as ações necessárias são tomadas.

3.2 Rede

A comunicação entre usuários/servidor é toda feita através do protocolo TCP. Tal protocolo foi escolhido para não haver a necessidade de lidar com confirmação de recebimento de mensagens e recebimento de bytes fora de ordem. Entretanto, ainda é necessário manipular os bytes que chegam pela conexão, pois há um formato específico para as mensagens e o protocolo TCP não garante que todos os bytes enviados por uma chamada a `send`

serão recebidos da mesma forma pela chamada a `recv` do outro lado (pode acontecer, por exemplo, de o último *buffer* ser concatenado ao primeiro em uma chamada a `recv` feita duas vezes – Ainda em ordem, claro).

3.2.1 Abstrações

Foram feitas abstrações para todos os componentes importantes na comunicação por rede a fim de facilitar seu uso por aplicações como o *chat* e outras. As principais abstrações foram feitas para:

- **Endereços de rede** – Foi criada uma classe *NetAddr* que representa um endereço de rede. Assim pode-se, por exemplo, instanciar um objeto *NetAddr* passando-se apenas como parâmetros o endereço IP e a porta, e a classe toma conta de preencher as estruturas como `sockaddr_in` apropriadamente.

- **Mensagens entre *Hosts*** – Por motivos citados acima, foi preciso criar um formato padronizado de mensagens a ser entendido por cliente e servidor. Para representar isso, há a classe *NetMessage*, que contém uma mensagem em um formato padronizado (a ser discutido a seguir) com informações sobre destino e fonte. Para se receber tais mensagens, há uma classe *NetReceiver*, que faz chamadas repetidas a `send` e separa apropriadamente os bytes de forma a se obter mensagens no formato padronizado.

3.2.2 Comunicação

Toda mensagem a ser trocada por cliente e servidor tem o formato:

```
CABEÇALHO <| CAMPO 1 | CAMPO 2 ...>
```

Onde os campos são opcionais. É necessário, ao menos, ter um cabeçalho, que indica qual a mensagem que chegou. Esse cabeçalho tem sempre um *byte*. A codificação das mensagens é feita na biblioteca `protocol` criada para a aplicação. A mensagem é sempre terminada com um caractere especial e, se há campos adicionais na mensagem, estes são separados por outro caractere especial. Há funções que garantem que **toda** cadeia de caractere inserida nas mensagens é sanitizada, isto é, não há os caracteres especiais nelas. Foram criadas funções especiais para cada mensagem. Ao usuário, só cabe passar os argumentos e a biblioteca cuida de garantir com que a mensagem seja formatada adequadamente. Após o recebimento da mensagem pelo outro lado, há outras funções que desmontam a mensagem e dão de volta a informação passada pelo usuário.

Como um exemplo, tomemos o caso de enviar uma mensagem para algum usuário do *chat*. O processo então é:

- O usuário A digita em seu console o comando para enviar uma mensagem para o usuário B:

  ```
  $ send B MENSAGEM
  ```

- O comando passa por *parsing*, é identificado e é chamada uma função para montar a mensagem que vai ser transmitida pela rede, onde cada campo é sanitizado, a mensagem é montada e enviada pela rede:

  ```
  msg = hostToNetSendMsg(A, B, MENSAGEM);
  ```

- Do outro lado, o servidor recebe a mensagem, extrai o cabeçalho e então chama `netToHostSendMsg(msg)`, que devolve um objeto da classe `Message` pronto para ser colocado na fila de mensagens.