Trabalho Prático 3 - Redes de Computadores
Um Sistema Peer-to-peer de Armazenamento Chave-valor

Humberto Lopes
Matricula

João Paulo Martins Castanheira
Matricula

Junio Cezar Ribeiro da Silva
2012075597

Universidade Federal de Minas Gerais
Departamento de Ciência da Computação

1. Introdução

No desenvolvimento de sistemas distribuídos temos como base a arquitetura P2P (*peer-to-peer*) caracterizada pelo compartilhamento direto de recursos sem a necessidade de um servidor centralizado. Cada ponto ou nó da rede pode funcionar tanto como cliente quanto servidor, sendo que a motivação para esse tipo de relação P2P está na colaboração oferecida por cada máquina ligada ao sistema, o que possibilita a oferta de dados e recursos computacionais descentralizados.
Esta documentação descreve a implementação de um sistema de armazenamento chave-valor do tipo P2P, utilizando interface de sockets UDP, onde 2 programas foram desenvolvidos:

- **Cliente:** Aplicação responsável por receber chaves inseridas pelo usuário, realizar consultas a algum *servent* e imprimir resultados na tela do usuário.
- **Servent:** Programa responsável pelo armazenamento da base de dados chave-valor e pelo controle da troca de mensagens com seus pares.

Para manter esta documentação tão curta quanto possível decidimos por omitir informações sobre o funcionamento do servent e do cliente que estão presentes na própria documentação do trabalho.

2. Arquitetura

2.1. Formato das Mensagens

Cada chave transferida em nosso sistema pode possuir um máximo de 41* caracteres, enquanto que os valores podem ser compostos por 160*. As mensagens enviadas e recebidas neste sistema podem transferir quantidades máxima de bytes distintas de acordo com os pares da comunicação. A quantidade de bytes transferida é associada ao tipo da mensagem em trânsito, sendo que as possíveis variações são listadas abaixo:

* Chaves tem um total de 40 caracteres inseridos pelo usuário + um caractere extra de terminação `'\0'`. Valores tem um total de 160 caracteres de informação + um caractere extra de terminação `'\0'`.

Mensagem CLIREQ

Tamanho total: 43 bytes

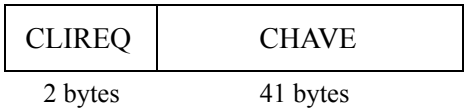


Figura 1: *CLIREQ* = 1. A mensagem CLIREQ sempre tem origem em uma aplicação cliente e tem como destino um servent.

O objetivo desta mensagem é requisitar ao servent o valor associado a *CHAVE* especificada pelo usuário. Como a mensagem deve ser transmitida em uma única chamada da função `sendto` da biblioteca de sockets *POSIX*, criamos um *array* do tipo `uint8_t` com 43 posições em que empacotamos os valores associados ao tipo da mensagem e a chave. Mais detalhes serão apresentados na seção 2.2.

Mensagem QUERY

Tamanho total: 55 bytes

QUERY	TTL	IP_CLIENTE	PORTA_CLIENTE	SEQ_NUM	CHAVE
2 bytes	2 bytes	4 bytes	2 bytes	4 bytes	41 bytes

Figura 2: *QUERY* = 2. A mensagem QUERY sempre tem origem em uma aplicação servent e tem como destino um outro servent.

O objetivo desta mensagem é repassar uma requisição enviada originalmente por um cliente a um servent. O campo TTL especifica o tempo de vida daquele pacote. O primeiro servent a receber uma mensagem CLIREQ de um cliente, seta TTL = 3. Este valor é então decrementado em cada servent que recebe esta mensagem e, caso o valor final seja superior a 0, o pacote é retransmitido a outros servents.

Mensagem RESPONSE

Tamanho total: 206 bytes

RESPONSE	CHAVE	‘\t’	VALOR	‘\0’
2 bytes	41 bytes	1 byte	161 bytes	1 byte

Figura 3: *RESPONSE* = 3. A mensagem RESPONSE sempre tem origem em uma aplicação servent e tem como destino um cliente.

O objetivo desta mensagem é responder a uma requisição realizada por um cliente. A mensagem, além de conter o *VALOR* associado a chave enviada pelo cliente, também contém esta chave em seu corpo. Note que um caractere de tabulação é inserido entre a chave e o valor associado, e que após o valor é inserido um caractere que caracteriza um byte nulo. Este último caractere pode ser redundante, uma vez que o campo *VALOR* pode finalizar com este mesmo byte, o que não é um problema para as aplicações.

2.2. Empacotamento de dados

Com a necessidade de manter todos os dados em NBO[1] (*Network-Byte-Order*) e empacotados em uma única cadeia de bytes para envio, realizamos empacotamento de todas as mensagens descritas na seção 2.2 em arrays com os mesmos tamanhos das mensagens. Cadeia de caracteres, como os dados presentes nos campos VALOR e CHAVE são inseridos na cadeia de bytes da maneira como estão presentes originalmente na memória, porém variáveis inteiras que identificam os campos TIPO, TTL, PORTA_CLIENTE (*`uint16_t`*), IP_CLIENTE e SEQ_NUM (*`uint32_t`*) são primeiramente convertidos para NBO com o uso das funções `htons` e `htonl` e, então, são copiadas para a cadeia de bytes a ser enviada.
A cópia dos dados é realizada por meio das funções `memcpy`. Abaixo temos o exemplo de como a mensagem do tipo QUERY é empacotada:

```
    memcpy(&(byte_array[0]),  &QUERY, sizeof(QUERY)); // 2 bytes
    memcpy(&(byte_array[2]),  &TTL,   sizeof(TTL)  ); // 2 bytes
    memcpy(&(byte_array[4]),  &IP,    sizeof(IP)   ); // 4 bytes
    memcpy(&(byte_array[8]),  &PORT,  sizeof(PORT) ); // 2 bytes
    memcpy(&(byte_array[10]), &SEQ,   sizeof(SEQ)  ); // 4 bytes
memcpy(&(byte_array[14]), KEY,    strlen(KEY) * sizeof(KEY[0]) );
```

2.3. Arquivos e Compilação

As aplicações foram escritas em linguagem C++ e fazem o uso de funcionalidades presentes na STL de C++, assim como em diferentes bibliotecas POSIX, sendo boa parte destas bibliotecas referente ao uso de sockets.

O código do trabalho é composto por 4 arquivos:

- `client.cc` : Arquivo com cabeçados e implementações das funções utilizadas pela aplicação cliente em nosso sistema P2P.
- `servent.cc` : Arquivo com cabeçalhos e implementações das funções utilizadas pela aplicação servent.

1 - *Quando dados são guardados em variáveis com armazenamento superior a 8 bits, por exemplo inteiros declarados com `uint16_t` ou `uint32_t`.*

- `utilitario.cc` : Implementações genéricas, compatíveis e utilizadas pela aplicações cliente e servant.
- `utilitario.h` : Cabeçalhos das funções implementadas no arquivo utilitario.cc

As aplicações podem ser compiladas com o uso do comando **`make`*** ou executando um a um os comandos listados no arquivo *Makefile* enviado junto com o código fonte.

As aplicações podem ser executadas com as seguintes linhas de comando:

1 - **Servent**
```
./servent <PORTA> <ARQUIVO-BASE-DADOS> <IP1:PORTA1>...<IPN:PORTAN>
```

2 - Emissor :
```
./client <IP:PORTA>
```

O trabalho foi desenvolvido e testado em um computador com processador x86, arquitetura de 64 bits. Sistema operacional GNU/Linux Ubuntu 14.04. O Compilador utilizado foi o G++ (GCC), versão 4.8.5.

3. Servent

O socket do servent é aberto com a configuração UDP:

```
socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)
```

Por se tratar de um socket UDP não precisamos usar a função connect, e por isso a comunicação é feita através das funções `sendto()` e `recvfrom()` que facilitam a captura do IP e Porto do interlocutor, seja ele um Cliente ou Servent.
Entretanto a função `bind()` se faz necessária, uma vez que o cliente precisa saber para onde ele vai enviar suas mensagens. Podemos considerar que este `bind()` é feito em uma porta publicamente conhecida do servent.
Para facilitar a localização do Servent na rede, decidimos utilizar a biblioteca `ifaddrs.h`*, parte da biblioteca C da GNU que nos permite listar as interfaces de rede e endereços IPs associados a elas:

```
Servent iniciado e escutando na porta 9992
As seguintes interfaces e enderecos IPs podem ser
utilizados para se comunicar com o servidor:
1 - Interface: lo     Endereco IP: 127.0.0.1
2 - Interface: wlan0  Endereco IP: 192.168.43.165
```

* Se por algum motivo o uso da biblioteca `ifaddrs.h` for um problema durante a compilação do trabalho, seu uso pode ser removido ao se compilar com a linha de comando **`make no-addr`**.

O servent implementa todos os requerimentos propostos na documentação do trabalho, ou seja, ele lê um arquivo de entrada, cria um dicionário, responde solicitações de clientes e comunica-se com outros servents. O dicionário foi implementado com o auxílio da estrutura **`std::map`** da STL de C++. Esta estrutura, **`std::map< string, string >`** armazena chaves e o valor associado. Como ela só permite a associação de uma chave a um único valor, esta estrutura sempre mantêm a versão mais atual de um valor para uma chave específica, o que atende um requerimento proposto na especificação do trabalho.

Cada servent também precisa guardar dados sobre todas as mensagens QUERY recebidas, a fim de identificar possíveis duplicações de mensagens. Atingimos esse objetivo o uso da estrutura **`std::set`** da STL de C++. Preenchemos esta estrutura com uma **`string`** composta pela combinação das substrings *`inet_ntoa(clientIP) + string(ntohs(PortaDoCliente)) + string(ntohl(inSeqNum)) + CHAVE`*. Essa string é gerada a cada QUERY e CLIREQ recebida e é inserida em nosso **`std::set <string>.`**

4. Cliente

Muito semelhante ao socket do Servent, com duas diferenças. O socket do Cliente não estabelece *bind* em nenhuma porta, portanto fica a cargo do Servant identificá-la. E é por esse motivo que o Cliente usa a função sendto() e o Servent recebe com a função recvfrom().

O único aspecto mais interessante do socket do cliente é que para receber mensagens ele não pode realizar uma operação blocante constante. Por isso tivemos que usufruir da estrutura timeval para limitar o tempo de 4 segundo até receber algo através do socket, conforme solicitado na especificação.

```
struct timeval tv;
tv.tv_sec = 4; // Timeout de 4 segundos para o recvfrom
tv.tv_usec = 0;
if (setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO,&tv,sizeof(tv)) < 0) {
    perr("setsockopt()");
}
```

4.1. Funcionamento

Após o início da execução, o cliente realiza a abertura do socket em modo UDP e define um *timeout* de 4 segundos para as operações recvfrom. Em seguida ele aguarda o usuário inserir uma chave de no máximo 40 caracteres.
Após o recebimento da chave, o cliente realiza o empacotamento dos dados em uma mensagem do tipo CLIREQ e a envia ao servent associado. Ele espera resposta por 4 segundos e caso nao receba nada, ele retransmite a mensagem **uma única vez**. Se ao retransmitir, ele ainda assim não receber uma mensagem, o programa volta a solicitar uma nova chave ao usuário.
No caso em que recebe uma mensagem RESPONSE, o cliente não sabe quantas outras podem chegar, por isso ele entra em *loop* aguardando todas as respostas chegarem. A Cada nova resposta, ele imprime

a chave na tela e o servent que o enviou. Cada tentativa de obtenção de resposta, persiste por 4 segundos. Assim que o socket tentar receber mais uma resposta e 4 segundos se passarem sem receber nada, o cliente pára de tentar receber mais respostas.

5. Testes

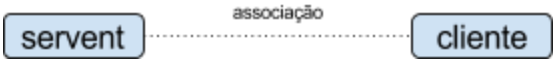
Os testes foram conduzidos em um único computador rodando o sistema operacional *GNU/Linux* Ubuntu 14.04, embora seja possível conectar clientes e servents localizados em máquinas distintas.

Os testes realizados são simples e tem como objetivo cobrir os seguintes pontos:

1. **Funcionamento simples de troca de mensagens CLIREQ e RESPONSE.**
2. **Propagação correta de mensagens QUERY.**
3. **Recebimento de QUERY duplicada em servent.**
4. **Respostas diferentes, por servents diferentes, para uma mesma CLIREQ.**

5.1. Funcionamento simples de troca de mensagens CLIREQ e RESPONSE.

Base de dados utilizada pelo servent: *teste1.input.txt*
Organização da rede:



Reprodução de telas:

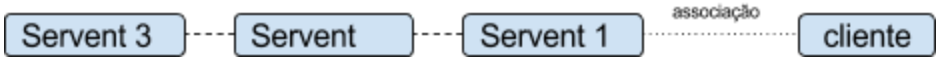
Servent	Cliente

`./servent 9999 teste1.input.txt`
`==========`
`Aguardando solicitacao...`
`Recebi CLIREQ do cliente 127.0.0.1:43103`
 	
`Chave consultada: < tcp >`
`Chave solicitada nao encontrada na base de dados.`
`==========`
`Aguardando solicitacao...`
`Recebi CLIREQ do cliente 127.0.0.1:43103`
 	
`Chave consultada: < tcp >`
`Chave solicitada nao encontrada na base de dados.`
`==========`
`Aguardando solicitacao...`
`Recebi CLIREQ do cliente 127.0.0.1:43103`
 	
`Chave consultada: < echo >`
`Chave encontrada no banco de dados com valor: ==>4/ddp # AppleTalk Echo`
	`./cliente 127.0.0.1:9999`
	
	`Insira uma chave para consulta, max 40 caracteres: tcp`
	
	`mandando para: 127.0.0.1:9999`
	`Não foi obtida uma resposta, tantando enviar mensagem novamente...`
	
	`mandando para: 127.0.0.1:9999`
	`Numero de tentativas de reenvio extrapolado, cancelando operacao.`
	
	`Insira uma chave para consulta, max 40 caracteres: echo`
	
	`mandando para: 127.0.0.1:9999`
	
	`Resposta recebida com ID = 3, enviada por 127.0.0.1:9999`
	`Dados: echo 4/ddp # AppleTalk Echo Protocol`

Protocol Response enviada ao cliente ========== Aguardando solicitacao...	Insira uma chave para consulta, max 40 caracteres:

5.2. Propagação correta de mensagens QUERY.

Base de dados utilizada pelo servent: *teste2.input.txt*
Organização da rede:



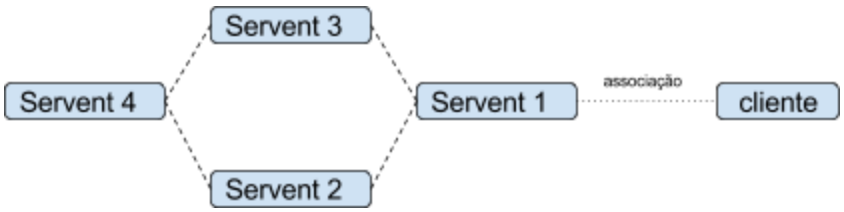
Reprodução de telas:

Cliente	Servent 1
./cliente 127.0.0.1:1111 Insira uma chave para consulta, max 40 caracteres: **rtsp** mandando para: 127.0.0.1:1111 Resposta recebida com ID = 3, enviada por 127.0.0.1:1111 Dados: rtsp 554/udp Resposta recebida com ID = 3, enviada por 127.0.0.1:2222 Dados: rtsp 554/udp Resposta recebida com ID = 3, enviada por 127.0.0.1:3333 Dados: rtsp 554/udp	./servent **1111** teste2.input.txt 127.0.0.1:**2222** ========== Aguardando solicitacao... Recebi CLIREQ do cliente 127.0.0.1:34860 Chave consultada: < **rtsp** > Enviando QUERY aos meus vizinhos Chave encontrada no banco de dados com valor: ==>554/udp RESPONSE enviada ao cliente ==========
Servent 2	**Servent 3**
./servent **2222** teste2.input.txt 127.0.0.1:**3333** ========== Recebi QUERY de um vizinho ==> Cliente original: 127.0.0.1:34860, SeqNum: 0, Chave: **rtsp** Identificacao do pacote: **127.0.0.1348600rtsp** Retransmitindo QUERY para meus vizinhos. Novo TTL: 2 Chave encontrada no banco de dados com valor:	./servent **3333** teste2.input.txt ========== Recebi QUERY de um vizinho ==> Cliente original: 127.0.0.1:34860, SeqNum: 0, Chave: rtsp Identificacao do pacote: 127.0.0.1348600rtsp Retransmitindo QUERY para meus vizinhos. Novo TTL: 1 **==> Nao ha vizinhos para retransmitir**

==>554/udp RESPONSE enviada ao cliente original ==========	Chave encontrada no banco de dados com valor: ==>554/udp RESPONSE enviada ao cliente original ==========

5.3. Recebimento de QUERY duplicada em servent.

Base de dados utilizada pelo servent: *teste3.input.txt*
Organização da rede:

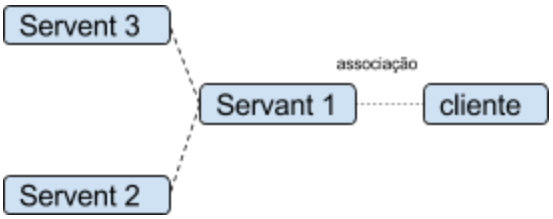


Reprodução de telas:
**Por simplificação, apenas reproduzimos as telas dos servents 1 e 4.*

Servent 1	Servent 4
./servent 1111 teste3.input.txt **127.0.0.1:2222 127.0.0.1:3333** ========== Aguardando solicitacao... Recebi CLIREQ do cliente 127.0.0.1:33741 Chave consultada: < udp > **Enviando QUERY aos meus vizinhos** Chave solicitada nao encontrada na base de dados. ========== Aguardando solicitacao... Recebi CLIREQ do cliente 127.0.0.1:33741 Chave consultada: < udp > **Enviando QUERY aos meus vizinhos** Chave solicitada nao encontrada na base de dados. ==========	./servent 4444 teste3.input.txt ========== Aguardando solicitacao... **Recebi QUERY de um vizinho** ==> Cliente original: 127.0.0.1:33741, SeqNum: 0, Chave: udp Identificacao do pacote: 127.0.0.1337410udp Retransmitindo QUERY para meus vizinhos. Novo TTL: 1 ==> Nao ha vizinhos para retransmitir Chave solicitada nao encontrada na base de dados. ========== Aguardando solicitacao... Recebi QUERY de um vizinho ==> Cliente original: 127.0.0.1:33741, SeqNum: 0, Chave: udp Identificacao do pacote: 127.0.0.1337410udp **Recebi solicitacao repetida, ignorando operacao.** ==========

5.4. Respostas diferentes, por servents diferentes, para uma mesma CLIREQ.

Base de dados utilizada pelo servent: *teste5.input.txt*
Organização da rede:



Reprodução de telas:

Servent 2	Servent 3
./servent 1111 teste4A.input.txt ========== Aguardando solicitacao... **Recebi QUERY de um vizinho** ==> Cliente original: 127.0.0.1:33741, SeqNum: 0, Chave: **tcpmux** Identificacao do pacote: **127.0.0.1337410tcpmux** Retransmitindo QUERY para meus vizinhos. Novo TTL: 2 **==> Nao ha vizinhos para retransmitir** Chave encontrada no banco de dados com valor: **==>9999 # TCP port service multiplexer** RESPONSE enviada ao cliente original ==========	./servent 4444 teste4B.input.txt ========== Aguardando solicitacao... **Recebi QUERY de um vizinho** ==> Cliente original: 127.0.0.1:33741, SeqNum: 0, Chave: **tcpmux** Identificacao do pacote: **127.0.0.1337410tcpmux** Retransmitindo QUERY para meus vizinhos. Novo TTL: 2 **==> Nao ha vizinhos para retransmitir** Chave encontrada no banco de dados com valor: **==>123 #new value maximal** RESPONSE enviada ao cliente original ==========
Cliente	
./cliente 127.0.0.1:1111 Insira uma chave para consulta, max 40 caracteres: tcpmux mandando para: 127.0.0.1:1111 Resposta recebida com ID = 3, enviada por 127.0.0.1:2222 Dados: **tcpmux 123 #new value maximal** Resposta recebida com ID = 3, enviada por 127.0.0.1:3333 Dados: **tcpmux 9999 # TCP port service multiplexer**	./servent 1111 arquivo_vazio.txt 127.0.0.1:2222 127.0.0.1:3333 ========== Aguardando solicitacao... Recebi CLIREQ do cliente 127.0.0.1:42676 Chave consultada: < **tcpmux** > **Enviando QUERY aos meus vizinhos** Chave solicitada nao encontrada na base de dados