INE5418 Computação Distribuída

2009

Trabalho 1 – Projeto 2

Parte 1: Fundamentos

1. Descrição Geral

A parte I (Fundamentos) da disciplina será avaliada através de um trabalho de implementação que esta dividido em 2 projetos, sendo que cada um deles corresponde a partes necessárias para a realização do seguinte trabalho.

O objetivo final é fazer um sistema nos moldes do *Dynamo*, um serviço de tabela contendo pares (chave, valor) (no melhor estilo das implementações da interface *java.util.Map* na API do Java) usado pela *amazon.com* para suportar as diversas aplicações que fazem com que sua infra-estrutura de comércio eletronico funcione [1]. Este nosso sistema será doravante chamado *TyDyn* (*Tyni Dynamo*).

No projeto 1 foram construídas uma série de funções para realizar a (1.) transformação de estruturas de dados em cadeias de bytes adequadamente formatadas para o envio pela rede e a recuperação das estruturas a partir destas cadeias; (2.) leitura de informações de um arquivo visando obter a lista de servidores disponíveis no sistema; e (3.) gestão de uma tabela local que suporte um subconjunto dos serviços definidos pelo *Dynamo*.

No projeto 2 vamos dar um passo a frente e implementar esta tabela num servidor, oferecendo ao cliente uma interface similar àquela implementada no projeto 1 para a tabela (definida pelo arquivo `tabela.h`). Isto implica que o cliente irá invocar uma série de operações que serão transformadas em mensagens e enviadas pela rede até o servidor, que interpretará essas invocações e realizará as operações correspondentes na tabela local implementada por ele, enviando a resposta em seguida. O objetivo final é fornecer as aplicações que usariam esta tabela um modelo de comunicação tipo RPC (*Remote Procedure Call*)[1], onde vários clientes acessam a uma mesma tabela partilhada.

O ponto inicial para a implementação do projeto 2 é a combinação do código desenvolvido no projeto 1 com as técnicas para comunicação por *sockets* TCP e suporte a múltiplos clientes simultaneamente pelo servidor através de multiplexação de I/O (*select* ou *poll*).

Para testar as funcionalidades do sistema, propõe-se a construção de uma aplicação cliente que armazena produtos (*struct product_t*, definida no projecto 1) no servidor *TyDyn*. Esta aplicação deve inserir, remover, consultar e listar os produtos armazenados no sistema.

Espera-se uma grande confiabilidade por parte do servidor, portanto não podem haver condições de erro não verificadas ou gestão de memória ineficiente a fim de evitar que este sofra um *crash*, o que deixaria todos os clientes sem a tabela de dados.

2. Programas a construir

Deve-se construir dois programas, com as seguintes linhas de comando:

- `tydyn-server <porta_servidor>`
  <porta_servidor> é a porta onde o servidor ficará à escuta;

---

[1] Ver aula teórica sobre RPC e capítulo 4 do livro texto da cadeira.

- `tydyn-client <arquivo_de_configuração>`
  <arquivo_de_configuração> é o nome de um arquivo, contendo endereços de servidores no formato `ip:porta` ou `hostname:porta`, um endereço por linha.

O **cliente** a construir usa as funcionalidades implementadas em `config.c` e `mensagem.c` no projeto 1, e é constituído por pelo menos três partes adicionais:

- Aplicação cliente interativa (ty`dyn-main.c`) com a função `main()`
- Rotina de adaptação do cliente, i.e., RPC *stub* (`tabela-stub.c`)
- Biblioteca de comunicação (`network.c`)

A aplicação interativa a realizar é um cliente simples que aceita um comando (uma linha) do usuário na `stdin`, invoca a respectiva função do *stub*, e imprime a resposta no monitor. Ao iniciar o cliente, este deve ler um arquivo de configuração (utilizando o `config.c` do projeto 1) e usar esta lista de servidores para inicializar o *stub*.

Cada comando vai ser inserido pelo usuário numa linha, havendo as seguintes alternativas:

```
put <key> <productname> <productprice>
get <key>
delete <key>
size
getkeys
quit
```

O cliente deve transformar o produto (representado pela *struct* `product_t`) numa cadeia de bytes e usar esta cadeia no campo `data`. O servidor pode simplesmente armazenar e retornar esta cadeia de bytes, sem ter que desempacotar o produto no campo `data`.

O **servidor** a realizar usa as funcionalidades de `mensagem.c` e `tabela.c` do projeto 1, e é composto por três partes adicionais:

- Aplicação servidor (ty`dyn-server.c`) com a função `main()`
- Rotina de adaptação do servidor, i.e., RPC *skeleton* (`tabela-skel.c`)
- Biblioteca de comunicação (`network.c`)

Os servidores deverão suportar vários clientes em simultâneo (mas atenderão apenas um pedido de cada vez). Como o servidor apenas necessita de tratar um pedido de cada vez, **não será** necessário recorrer à bifurcação do programa em *threads* ou processos filho (ex., através da chamada de sistema *fork*). No entanto, como o protocolo TCP é um protocolo com ligação, será necessário que o servidor seja capaz de escutar simultaneamente várias *sockets* (ver *Manual Pages* de `select` e `poll`). Estas funções permitem selecionar um conjunto de *file descriptors* (e.g., *sockets*) e esperar até que surja uma operação de I/O (e.g., dados disponíveis para leitura) num deles [3].

Uma boa estratégia para realizar o trabalho é fazer o sistema sem suportar a escuta simultânea de vários *sockets* e então, numa segunda etapa, incluir esta funcionalidade no servidor.

3. RPC stub

A interface a ser oferecida ao cliente é a seguinte:

```
#ifndef _TABELA_STUB_H
#define _TABELA_STUB_H
#include “tabela-types.h”
```

```
#define RPC_FAILURE (-5)

/* Função para se conectar com um dos servidores com endereços (host:port)
 * definidos no array de strings servers. O cliente deve se conectar com o
 * primeiro servidor do array que responda a conexão. */
int connect_table(char** servers);

/* Função para se desconectar do servidor de tabela. Desalocando todos os
 * recursos alocados para a comunicação (ex., descritores e memória). */
int disconnect_table();

/* Funcao para adicionar um elemento na tabela.  */
int puttab(struct key_t key, struct data_t data);

/* Funcao para obter um elemento da tabela.  */
int gettab(struct key_t key, struct data_t *data);

/* Função para remover um elemento da tabela. */
int removetab(struct key_t key);

/* Devolve número de elementos da tabela. */
int tabsize();

/* Devolve um array com a cópia de todas as keys da tabela
 * (função aloca memoria do array e guarda endereço em *keys),
 * e devolve o número de elementos na tabéla. */
int gettabkeys(struct key_t **keys);

/* Desaloca a memoria alocada por gettabkeys() no cliente  */
void freetabkeys(struct key_t *keys);

#endif
```

Os valores de retorno das funções no lado cliente são os mesmos definidos no projeto 1, a não ser pelo fato de que todas elas podem retornar `RPC_FAILURE` se foi impossível manter a comunicação com o servidor.

4. Biblioteca de comunicação

A biblioteca de comunicação, a implementar em `network.c`, é partilhado entre o cliente e o servidor. A interface a ser oferecida é a seguinte:

```
#ifndef _NETWORK_H
#define _NETWORK_H
#include "mensagem.h"

/* Funcao para initializar socket do servidor, retorna socket ou -1 */
int init_server_socket(int port);

/* Funcao para initializar socket do cliente, retorna socket ou -1 */
int init_client_socket(char *server);

/* Recebe uma mensagem de um socket, retorna NULL em caso de erro */
struct message_t *network_to_message(int sockfd);

/* Envia uma mensagem pela rede, retorna 0 (OK) ou -1 (error)  */
int message_to_network(struct message_t *message, int sockfd);
#endif
```

As funções `message_to_network` e `network_to_message` devem usar o formato serializado definido no projeto 1 (`mensagem.c`).

A função `message_to_network` serve para enviar pedidos do cliente ao servidor, e para enviar respostas do servidor ao cliente. Deve primeiro enviar o tamanho da mensagem serializado (num uint32_t em formato da rede), seguido pela mensagem serializada por `message_to_buffer()`.

A função `network_to_message` serve para receber pedidos do cliente no servidor, e para receber respostas do servidor no cliente. A função deve primeiro ler o tamanho recebido no socket, alocar memória para um buffer com o tamanho adequado, ler todo o buffer do socket, e depois transformar o buffer numa estrutura `struct message_t`. Nota-se que um servidor concorrente só chama a função depois de verificar que há dados disponíveis para a leitura (através das chamadas a *select* ou *poll*).

5. Skeleton

O *skeleton*, a implementar em `tabela-skel.c`, serve para transformar uma mensagem do cliente numa chamada da respectiva função da `tabela.c`. A interface é simples:

```
#ifndef _TABELA_SKEL_H
#define _TABLEA_SKEL_H
#include "mensagem.h"

int invoke(struct message_t *msg);
#endif
```

A função `invoke` interpreta o campo *opcode* de `msg` para seleccionar a função a chamar. Depois de executar essa função, coloca na mesma estrutura `msg` o resultado.

6. Formato das mensagens

Todas as mensagens enviadas do cliente para o servidor e vice-versa seguem o formato serializado da `struct message_t` definido no projeto 1. Se uma mensagem não utilizar o campo *key* (data), indicado por NULL na tabela a seguir, deve-se enviar o valor 0 no campo *keysize* (datasize). Os comandos a usar com os respectivos *opcodes* são os seguintes:

Operação (client->server)	conteúdo da mensagem (opcode, key, data):
`01: puttab`	`(01, key, data)`
`02: gettab`	`(02, key, size)`
`03: removetab`	`(03, key, NULL)`
`04: tabsize`	`(04, NULL, NULL)`
`05: gettabkeys`	`(05, NULL, NULL)`

Resposta (server->cliente)	conteúdo da mensagem (opcode, key, data):
`11: puttab-reply`	`(11, key, status)`
`12: gettab-reply`	`(12, key, data) [data==NULL se key não existe]`
`13: removetab-reply`	`(13, key, status)`
`14: tabsize-reply`	`(14, NULL, size)`
`15: gettabkeys-reply`	`(15, NULL, keydata)`

A parte de data tem o seguinte formato:

DATA: data	dados (contendo uma `struct product_t` serializada)
DATA: status	int32_t serializado em formato da rede, 0=OK, <0=ERROR
DATA: size	int32_t serializado em formato da rede, com tamanho da tabela ou tamanho do buffer do cliente
DATA: keydata	*array* de *struct key_t* serializado com o formato

KEY1SIZE KEY1 KEY2SIZE KEY2 KEY3SIZE KEY3 ...
(usa tipo `uint32_t` em formato da rede para o tamanho)

7. Exemplos de interação cliente - servidor

a) Usuário insere o comando "`put K123 Book 1095`"

O *stub* do cliente envia 30 bytes ao servidor (bytes da mensagem em notação hexadecimal): primeiro o tamanho da mensagem serializada (uint32_t em formato da rede), seguido pela mensagem (26 bytes)

```
00|00|00|1A|00|01|00|00|00|04|'K|'1|'2|'3|00|00|00|0C|00|00|00|04|'B|'o|'o|'k|00|00|04|47|
buflen=26  |op=1 |keysize=4  |key        |datasize=12|-------------data----------------|
                                                    |namelen=4  |name       |preço      |
```

O servidor responde com 22 bytes (mensagem serializada: 18 bytes) ao cliente:

```
00|00|00|12|00|0B|00|00|00|04|'K|'1|'2|'3|00|00|00|04|00|00|00|00|
buflen=18  |op=11|keysize=4  |key        |datasize=4 |----data---|
                                                    |status=ok  |
```

b) Usuário insere o comando "`get K123`"
(main() do cliente passa buffer com tamanho 1024 bytes a gettab())

O stub do cliente envia 18 bytes ao servidor:

```
00|00|00|0E|00|02|00|00|00|04|'K|'1|'2|'3|00|00|00|04|00|00|04|00|
buflen=14  |op=2 |keysize=4  |key        |datasize=4 | len=1024  |
```

O servidor responde com 30 bytes ao cliente:

```
00|00|00|1A|00|0C|00|00|00|04|'K|'1|'2|'3|00|00|00|0C|00|00|00|04|'B|'o|'o|'k|00|00|04|47|
buflen=26  |op=12|keysize=4  |key        |datasize=12|-------------data----------------|
                                                    |namelen=4  |name       |preço      |
```

8. Alterações nas partes do projeto 1

- Em `mensagem.c`, adicionar funções para empacotar/desempacotar um *array* de `struct key_t` (usadas para transferir a resposta de `gettabkeys`);
- Em `tabela.c`, adicionar uma função `inittab()`, que se deve chamar antes de usar a tabela;
- Em `tabela.c`, alterar a interface de `gettabkeys` (compatível com `tabela-stub.c`, devolvendo as *keys* no argumento e um inteiro no *retval* para poder indicar um erro no RPC);
- Em `config.c`, o arquivo de configuração vai conter endereços TCP (host:port) em vez de só endereços IP.

9. Makefile

Deve-se também escrever um `Makefile` que permita compilar os dois programas, com os seguintes *targets*:

- **tydyn-client-lib**: Compilar os arquivos `tabela-stub.c`, `network.c`, `mensagem.c` e `config.c` (e eventuais arquivos adicionais "`.c`" criados na implementação do cliente), criando uma biblioteca **tydyn-client.o** (Usa linker *ld* com opção *-r*: `ld -r in1.o in2.o [...] -o out.o`);
- **tydyn-client**: Compilar `simdyn-main.c` junto com a biblioteca `simdyn-client.o`, criando a aplicação cliente **tydyn-client**;
- **tydyn-server**: Compilar o servidor completo (composto ao menos por `tydyn-server.c`, `tabela-skel.c`, `tabela.c`, `mensagem.c`, e `network.c`);
- **clean**: Remover todos os arquivos criados pelos *targets* acima.

10. Entrega

- O trabalho pode ser feito em grupo (até 3 alunos).
- Os projetos deverão ser submetidos até a sua data de entrega, via e-mail (fernando@inf.ufsc.br), contendo :
  - Subject do e-mail : <ine5418+projeto> Ex.: ine5418projeto1, ine5418projeto2, ...
  - o arquivo com código fonte+ .h (em anexo),
  - descrição do trabalho, nome do(s) autor(es) e procedimentos de compilação/execução.

  A identificação dos arquivos deverá seguir o seguinte padrão, <trabalho + login [-login2] >, por exemplo: proj2_*fernando.c* , proj2_*fernando.h* .
- ***O prazo de entrega para o projeto 2 é dia 30/04/2009 até as 18:00hs***.

11. Bibliografia

[1] Giuseppe DeCandia et al. *Dynamo: Amazon's Highly Available Key-value Store*. Proc. of the 21st Symposium on Operating System Principles – SOSP'07. pp. 205-220. Out. de 2007.

[2] B. W. Kernighan, D. M. Ritchie, *C Programming Language*, 2nd Ed, Prentice-Hall, 1988.

[3] Linux Man pages. select, *etc*… http://www.linuxmanpages.com/man2/select.2.php. Consultado em 18 de Outubro de 2008.