Named pipes - Resumo e Exemplo Cliente/Servidor

1. Enquadramento

Este exemplo mostra como se pode construir um sistema de informação cliente-servidor utilizando *named pipes* (também referidos como FIFO) para transportar informação entre os clientes e o servidor. Neste exemplo, tal como acontece genericamente quando se fala em "cliente-servidor", tanto o servidor como o cliente enviam e recebem informação (ou seja, a informação flui nos dois sentidos).

1.1 Named pipes em Unix

Os *named pipes* em unix têm as seguintes características genéricas:

- Correspondem por alto a um tubo: escreve-se informação numa extremidade e retira-se essa informação pela outra extremidade. A informação é retirada pela mesma ordem com que foi enviada, daí o nome alternativo para este mecanismo de comunicação: FIFO – "*First In First Out*".
- O uso de um *named pipe* é semelhante ao de um ficheiro. Usam-se as mesmas funções para abrir, fechar, ler e escrever. A informação a escrever/ler é semelhante à noção de "ficheiro binário" no sentido em que o que é escrito é simplesmente *uma determinada quantidade de bytes* cujo significado é da responsabilidade dos processos que usam o *named pipe* e não do sistema operativo (nota: recordar que em unix não existe distinção entre ficheiros binários/texto).
- A estrutura interna da mensagem, e o tamanho é totalmente da responsabilidade dos processos, que devem conhecer à partida a estrutura e quantos bytes é suposto ler/escrever.
- A comunicação através de *named pipes* é, normalmente, unidireccional. Para se conseguir ter uma comunicação bidireccional entre dois processos serão usados, normalmente, dois *named pipes*, uma para cada direcção.
- Normalmente, num determinado *named pipe* só existirá um processo a ler. Podem haver um ou vários processos a escrever, dependendo do modelo de comunicação a ser implementado.

1.2 Named pipes para o modelo cliente-servidor

Os *named pipes* são independentes do modelo de comunicação (caixa de correio, cliente-servidor, difusão, peer-to-peer, etc.). No modelo cliente-servidor, tem-se normalmente a seguinte forma de uso de *named pipes*:

- Um *named pipe* "do servidor". Este *named pipe* serve para os clientes comunicarem algo ao servidor. Os vários clientes escrevem nesse *pipe* mas apenas o servidor lê.
- Um *named pipe* por cada cliente (*named pipe* "do cliente"). Estes *named pipe* servem para o servidor envia algo a cada cliente. Só o servidor escreve nestes *named pipes*, em, em cada um deles, apenas um cliente lê.
- O *named pipe* "do servidor" terá uma identificação pré-estabelecida, sendo conhecida em todos os processos envolvidos. A identificação dos *named pipes* "dos clientes" não é conhecida à partida pelo servidor, devendo cada cliente informar o servidor acerca da identificação do "seu" *named pipe*.
- O servidor cria o "seu" *named pipe* (função *mkfifo*). Cada cliente cria o "seu" *named pipe*. Os processos que criaram o *named pipe* são responsáveis por o remover (função *unlink*) quando terminam.

1.3 Persistência dos *handles* abertos para os *pipes* de outros processos

Normalmente, cada processo mantém o "seu" *named pipe* (onde lê) aberto para leitura durante toda a sua execução. Quanto aos *named pipes* dos outros processos, nos quais vai escrever, existem duas alternativas:

- Abrir e fechar o *named pipe* de cada vez que o vai usar (para escrita):
    - Mais lento em termos de processamento (mais operações *open* e *close*).
    - Mais simples em termos de estruturas de dados mantidas internamente pelo programa e pelo sistema operativo. Em sistemas como muitos clientes em simultâneo, é a solução mais viável (por exemplo, sistemas *web*).
    - Eventualmente mais simples de implementar para o programador.
    - Esta escolha será mais comum no servidor do que no cliente. **No entanto**, se o servidor precisar de *manter um conhecimento permanente ao longo do tempo acerca dos clientes* (por exemplo, para lhe enviar informação por iniciativa própria sem ser apenas para responder a um pedido[1]), então esta alternativa não é viável.
- Abrir o *named pipe* no primeiro uso, mante-lo aberto, e fechá-lo apenas quando determinar que não vai voltar a interagir com esse processo/*named pipe*.
    - Mais rápido em termos de processamento uma vez que se evita a repetição de *open/close*.
    - Obriga à manutenção em memória de *handles* (descritores de ficheiros) para ficheiros (*named pipes*) abertos. Normalmente existe um limite para o número de *handles* abertos e em sistemas reais com muitos clientes em simultâneo esta alternativa pode ser inviável.

[1] Tal como pode acontecer no trabalho prático.

- Eventualmente exige um pouco mais de esforço da parte do programador para manter a informação acerca de cada processo/*named pipe* conhecido. Se se estiver a interagir com apenas um outro processo, a diferença em termos de esforço para o programador é nula e o algoritmo acaba por ficar mais simples. Esta escolha faz todo o sentido no lado do cliente.
- Esta alternativa pressupõe que os processos (clientes) enviam informação uns aos outros (ao servidor) acerca de quando vão deixar de interagir com eles para que esse outro processo possa fechar os *handles* para os *named pipes* envolvidos.

1.4 Modos *bloqueante* e *não-bloqueantes* dos *named pipes* em Unix

Quanto à sincronização, os *named pipes* têm dois modos de funcionamento. **É importante recordar o material apresentado nas aulas e já disponibilizado** quanto a este aspecto, do qual apenas se vai aqui fazer um breve resumo. Os dois modos de funcionamento são:

- **Modo bloqueante / modo síncrono**. As operações *open*, *read*, *write* podem bloquear o processo que as invoca. Este comportamento simplifica bastante os algoritmos, pois têm-se a garantia que o processo só avança quanto existem condições para tal (por exemplo, um *read* só avança quando a informação pedida está realmente disponível / foi escrita por um outro processo). No entanto, existe o perigo dois ou mais processos entrarem em espera mútua uns pelos outros e ficarem permanentemente bloqueados.

- **Modo não-bloqueante / assíncrono**. As operações *open*, *read*, *write* não bloqueiam o processo que as invoca. Avançando logo. Este comportamento complica os algoritmos, pois é necessário verificar sempre se efectivamente se obteve o que se pretendia (por exemplo, dados num *read*), e, caso não se tenha obtido, o que fazer, como e quando (voltar a ler? Quando? O que se faz entretanto?). No entanto, este comportamento não coloca o perigo bloqueio mútuo e, em certos casos, pode ser necessário recorrer a uma operação assíncrona precisamente para evitar uma situação de espera mútua. Em príncipio, em situações moderadamente simples, não será necessário recorrer a comportamento assíncrono.

Normalmente o modo bloqueante é o que se pretende usar. No modo bloqueante existem dois aspectos mais salientes a ter em atenção:

- A abertura do *named pipes* bloqueia até que outro processo abra também o mesmo *pipe* para a operação inversa (*read* → *write, write* → *read*). Este é um dos pontos mais críticos quanto às situações de espera mútua e bloqueio permanente: o avanço de um processo fica logo dependente da existência de outro processo.
  Há duas formas para lidar com esta situação:
  - Planear cuidadosamente as operações de abertura em ambos os processos cliente e servidor para evitar situações de espera mútua.
  - Num dos processos, abrir um dos *pipes* em modo não-bloqueante e depois mudá-lo para modo bloqueante com a função *fcntl*. No entanto, essa operação de controlo pode afetar operações entretanto já desencadeadas nesse *pipe* por outros processos, quebrando a lógica de comunicação entre eles.

- Tendo um *named pipe* aberto para uma determinada operação (*read* ou *write*), quando deixa de existir outro processo com esse *named pipe* aberto para a operação inversa, o *pipe* reverte para o modo não bloqueante, complicando bastante o algoritmo. Esta situação ocorre, por exemplo quando o servidor se mantém em existência (como é normal) e, momentaneamente, deixa de haver clientes ligados a ele, ou seja, deixa de haver processos como named pipe do servidor aberto para escrita, fazendo com que as operações de leitura no servidor passem a ser não-bloqueante.
    - Uma maneira de lidar com esta situação corresponde ao próprio processo abrir o seu *named pipe* também para a operação inversa àquela que precisa para ter a garantia que existira sempre um processo (ele próprio) com o *named pipe* aberto "para a operação inversa", nunca revertendo este para o comportamento não-bloqueante. Bastará manter o *named pipe* aberto para a "operação inversa" não sendo necessário concretizar essas operações.

As duas situações descritas acima podem ser resolvidas simultaneamente de uma forma muito simples: cada processo (cliente e servidor) vai abrir o seu próprio *named pipe* para leitura e escrita em simultâneo (e em modo bloqueante). Desta forma:

- A função open não bloqueia – há logo um processo (ele próprio) com o *pipe* aberto para a operação inversa, e, portanto, a função retorna logo.
- O processo tem sempre a garantia da existência de um processo (ele próprio) com o *pipe* aberto para a "operação inversa

2. Contexto do exemplo e estratégia de implementação

O exemplo implementa uma situação cliente-servidor. O servidor tem a funcionalidade de traduzir palavras: recebe um pedido com uma palavra e responde com a tradução dessa palavra. Trata-se de uma situação simples, para servir de exemplo com as seguintes características:

- O protocolo de comunicação e interação entre cliente e servidor não exige a memorização dos clientes por parte do servidor – trata-se de um exemplo simples. Assim, o servidor abre e fecha imediatamente o *pipe* do cliente e não mantém informação acerca deles
- Tanto o cliente como o servidor abrem o seu próprio *named pipe* para leitura e para escrita. No entanto, apenas vão ler. Trata-se de seguir a solução mais simples para (descrita acima) para garantir que há sempre um processo com o *named pipe* aberto para a operação inversa, impedindo-o de entrar em modo não-bloqueante, e garantir que a operação *open* não bloqueia.

3. Linhas de trabalho sobre o exemplo

O exemplo pode ser usado como ponto de partida para diversos melhoramentos

- Manter no servidor a informação acerca de cada cliente.
- Prever um protocolo de comunicação (tipos de mensagens) mais complexo, que permita a um cliente informar que está a chegar, terminar, etc.
- Manter os descritores dos *named pipes* sempre abertos.
- Prever uma funcionalidade que envolva o servidor enviar informação aos clientes sem ser na situação de resposta a uma pergunta explícita.
- Permitir notificações assíncronas do servidor para o cliente com o auxílio de sinais
- Permitir que um cliente veja informação originária de outro cliente.
- Permitir que o servidor avise os clientes de que vai encerrar.
- Permitir encerrar o servidor através de um pedido explícito enviado por um cliente.
- Acrescentar a noção de utilizadores e de autenticação.
- Usar uma funcionalidade mais complexa em vez de um simples dicionário, nomeadamente algo que envolva dados que persistam de uns pedidos para outros.

4. Implementação e código dos programas

O código dos programas é apresentado a partir da página seguinte.

dicionario-cliente-servidor-codigo.txt

É suposto colocar o código numa máquina Unix para ***compilar e experimentar***, e nomeadamente, **seguir as indicações indicadas no ponto 3 deste documento**.

O código vai ser organizado em três ficheiros

- dict.h
- servidor.c
- cliente.c

Notas:

- O código do exemplo menciona FIFO – trata-se da designação alternativa dos *named pipes*.
- Existem algumas mensagens no ecrã para efeitos de *debug* (podem ser removidas). Também se podem executar os programas com redireccionamento do stderr para /dev/null para não ver essas mensagens.
- **Uso deste código no trabalho prático por mera colagem**: **desaconselhado**. Não é necessariamente o ideal para o trabalho + reutilização de código não é valorizado como código do próprio + qualquer código que aparece no trabalho tem sempre que ser explicado e defendido.

5. Código dos programas

dict.h - Definições comuns a incluir em ambos servidor e cliente

```
/* ficheiro header necessário aos clientes e servidor */

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <ctype.h>

/* nome do FIFO do servidor */
#define SERVER_FIFO "/tmp/dict_fifo"

/* nome do FIFO cada cliente. P %d será substituído pelo PID com sprintf */
#define CLIENT_FIFO "/tmp/resp_%d_fifo"

/* tamanho máximo de cada palavra */
#define TAM_MAX 50

/* estrutura da mensagem correspondente a um pedido cliente -> servidor*/
typedef struct {
   pid_t pid_cliente;
   char  palavra[TAM_MAX];
} pergunta_t;

/* estrutura da mensagem correspondente a uma resposta servidor -> cliente */
typedef struct {
   char palavra[TAM_MAX];
} resposta_t;
```

Código do servidor

```
#include "dict.h"
#include <signal.h>

#define NPALAVRAS 7       /* Número de palavras conhecidas */

char * dicionario[NPALAVRAS][2] = {  /*  O dicionário      */
   { "memory",    "memória" },       /*  é constituído     */
   { "computer",  "computador" },    /*  por uma matriz    */
   { "close",     "fechar" },        /*  bidimensional de  */
   { "open",      "abrir" },         /*  ponteiros para    */
   { "read",      "ler" },           /*  caracter.         */
   { "write",     "escrever" },      /*  [i][0] = palavra  */
   { "file",      "ficheiro" } };    /*  [i][1] = tradução */

int s_fifo_fd, c_fifo_fd; /* descritores de ficheiros (pipes) */

/* esta função vai atender o sinal SIGINT para terminar o servidor */
void trataSig(int i) {
   fprintf(stderr, "\nServidor de dicionario a terminar "
                   "(interrompido via teclado)\n\n");
   close(s_fifo_fd);
   unlink(SERVER_FIFO);
   exit(EXIT_SUCCESS);    /* para terminar o processo */
}

int main() {
   pergunta_t perg;   /* mensagem do "tipo" pergunta */
   resposta_t resp;   /* mensagem do "tipo" resposta */

   int i, res;
   char c_fifo_fname[50];
   char * aux;
   printf("\nServidor de dicionario");
   if (signal(SIGINT,  trataSig) == SIG_ERR) {
       perror("\nNao foi possivel configurar o sinal SIGINT\n");
       exit(EXIT_FAILURE);
   }
   fprintf(stderr, "\nSinal SIGINT configurado");
```

```
res = mkfifo(SERVER_FIFO, 0777);
if (res == -1) {
   perror("\nmkfifo do FIFO do servidor deu erro");
   exit(EXIT_FAILURE);
}
fprintf(stderr, "\nFIFO servidor criado");

/* prepara FIFO do servidor */
/* abertura read+write -> evita o comportamento de ficar */
/* bloqueado no open. a execução prossegue e as          */
/* operações read/write (neste caso APENAS READ)         */
/* continuam bloqueantes (mais fácil de gerir)           */
s_fifo_fd = open(SERVER_FIFO, O_RDWR);  /* bloqueante */
if (s_fifo_fd == -1) {
   perror("\nErro ao abrir o FIFO do servidor (RDWR/blocking)");
   exit(EXIT_FAILURE);
}
fprintf(stderr,"\nFIFO aberto para READ (+WRITE) bloqueante");

/* ciclo principal: read pedido -> write resposta -> repete */
while (1) {  /* sai via SIGINT */
   /* ---- OBTEM PERGUNTA ---- */
   res = read(s_fifo_fd, & perg, sizeof(perg));
   if (res < sizeof(perg)) {
         fprintf(stderr,"\nRecebida pergunta incompleta "
                 "[bytes lidos: %d]", res);
         continue; /* não responde a cliente (qual cliente?) */
   }
   fprintf(stderr, "\nRecebido [%s]", perg.palavra);

   /* ---- PROCURA TRADUCAO ---- */
   strcpy(resp.palavra, "DESCONHECIDO"); /* caso n encontre */
   for (i=0; i<NPALAVRAS; i++)
      if (!strcasecmp(perg.palavra,dicionario[i][0])) {
         strcpy(resp.palavra,dicionario[i][1]);
         break;
      }
   fprintf(stderr, "\nResposta = [%s]", resp.palavra);

   /* ---- OBTEM FILENAME DO FIFO PARA A RESPOSTA ---- */
   sprintf(c_fifo_fname, CLIENT_FIFO, perg.pid_cliente);
```

```
        /* ---- Abre FIFO do cliente p/ write ---- */
        c_fifo_fd = open(c_fifo_fname, O_WRONLY);
        if (c_fifo_fd == -1)
           perror("\nErro no open - Ninguem quis a resposta");
        else {
           fprintf(stderr,"\nFIFO cliente aberto para WRITE");

           /* ---- ENVIA RESPOSTA ---- */
           res = write(c_fifo_fd, & resp, sizeof(resp));
           if (res == sizeof(resp))
              fprintf(stderr,"\nescreveu a resposta");
           else
              perror("\nerro a escrever a resposta");

           close(c_fifo_fd); /* FECHA LOGO O FIFO DO CLIENTE! */
           fprintf(stderr,"\nFIFO cliente fechado");
        }
     } /* fim do ciclo principal do servidor */

     /* em principio não chega a este ponto - sai via SIGINT */
     close(s_fifo_fd);
     unlink(SERVER_FIFO);
     return 0;
} /* fim da função main do servidor */
```

(código do cliente na página seguinte)

Código do cliente

```
#include "dict.h"

int main() {
   int s_fifo_fd;          /* identificador do FIFO do servidor */
   int c_fifo_fd;          /* identificador do FIFO deste cliente */
   pergunta_t perg;        /* mensagem do "tipo" pergunta */
   resposta_t resp;        /* mensagem do "tipo" resposta */
   char c_fifo_fname[25];  /* nome do FIFO deste cliente */
   int read_res;

   /* cria o FIFO do cliente */
   perg.pid_cliente = getpid();
   sprintf(c_fifo_fname, CLIENT_FIFO, perg.pid_cliente);
   if (mkfifo(c_fifo_fname, 0777) == -1) {
      perror("\nmkfifo FIFO cliente deu erro");
      exit(EXIT_FAILURE);
   }
   fprintf(stderr,"\nFIFO do cliente criado");

   /* abre o FIFO do servidor p/ escrita */
   s_fifo_fd = open(SERVER_FIFO, O_WRONLY); /* bloqueante */
   if (s_fifo_fd == -1) {
      fprintf(stderr, "\nO servidor não está a correr\n");
      unlink(c_fifo_fname);
      exit(EXIT_FAILURE);
   }
   fprintf(stderr,"\nFIFO do servidor aberto WRITE / BLOCKING");

   /* abertura read+write -> evita o comportamento de ficar */
   /* bloqueado no open. a execução prossegue logo mas as   */
   /* operações read/write (neste caso APENAS READ)         */
   /* continuam bloqueantes (mais fácil)                    */
   c_fifo_fd = open(c_fifo_fname, O_RDWR); /* bloqueante */
   if (c_fifo_fd == -1) {
      perror("\nErro nao abrir o FIFO do cliente");
      close(s_fifo_fd);
      unlink(c_fifo_fname);
      exit(EXIT_FAILURE);
   }
```

```
    fprintf(stderr, "\nFIFO do cliente aberto para READ (+Write) BLOCK");

    memset(perg.palavra, '\0', TAM_MAX);

    while (1) {  /* "fim" para terminar cliente */
       /* ---- a) OBTEM PERGUNTA ---- */
       printf("\nPalavra a traduzir -> ");
       scanf("%s",perg.palavra);
       if (!strcasecmp(perg.palavra,"fim"))
          break;

       /* ---- b) ENVIA A PERGUNTA ---- */
       write(s_fifo_fd, & perg, sizeof(perg));
       /* ---- c) OBTEM A RESPOSTA ---- */
       read_res = read(c_fifo_fd, & resp, sizeof(resp));
       if (read_res == sizeof(resp))
          printf("\nTraducao -> %s", resp.palavra);
       else
          printf("\nSem resposta ou resposta incompreensivel"
                 "[bytes lidos: %d]", read_res);
    }

    close(c_fifo_fd);
    close(s_fifo_fd);
    unlink(c_fifo_fname);
    return 0;
} /* fim da função main do cliente*/
```