Comunicação em Rede

A comunicação em rede no ambiente UNIX é feita através do conceito de *sockets*. Um *socket* é uma interface de comunicação bidirecional entre processos. Sockets são representados como descritores de arquivos e permitem a comunicação entre processos distintos na mesma máquina ou em máquinas distintas, através de uma rede. Os sockets são a base da comunicação em redes TCP/IP e também são muito usados em comunicações entre processos no interior de um mesmo computador.

Estilos, nomes e protocolos

Ao criar um *socket*, deve ser especificada a forma de comunicação (*communication style*) a ser usada e que protocolo deverá realizar a comunicação nos níveis mais baixos do sistema. Em UNIX, três formas de comunicação são normalmente usadas:

- **stream** : esta forma de comunicação provê canais de comunicação bidirecional ponto-a-ponto (entre dois processos pré-definidos) sobre os quais pode-se enviar seqüências de bytes de qualquer tamanho, de maneira confiável (sem perdas, duplicações ou inversões de ordem nos dados).
- **datagram** : forma que provê canais de comunicação bidirecionais sobre os quais pode-se enviar pacotes de dados a qualquer processo que tenha um canal equivalente definido. Não há garantia na entrega dos pacotes, pois o sistema implementa uma política de *best-effort*. Além disso, como os pacotes são tratados de forma independente, podem haver inversões de ordem.
- **raw** : esta forma de comunicação provê acesso direto aos protocolos e interfaces de comunicação de baixo nível. Geralmente é usada por processos que gerenciam ou monitoram a infra-estrutura de rede.

Essas formas de comunicação são definidas através das constantes inteiras `SOCK_STREAM`, `SOCK_DGRAM` e `SOCK_RAW`, no arquivo `sys/socket.h`.

Além da forma de comunicação, também deve ser escolhido um espaço de nomes para nomear ou endereçar o socket criado. O nome de um socket sempre está relacionado a um espaço de nomes, também chamado de domínio (*socket domain*). Cada espaço de nomes é definido por uma macro na forma `PF_*` (que vem do termo *Protocol Family*). Os principais espaços de nomes em uso no UNIX são:

- `PF_LOCAL` : indica o espaço de nomes local, no qual os nomes de sockets são válidos somente no escopo do computador local. As macros `PF_UNIX` e `PF_FILE` são sinônimos desse espaço de nomes.
- `PF_INET` : indica o espaço de nomes IPv4 e seus protocolos associados.
- `PF_INET6` : indica o espaço de nomes IPv6 e seus protocolos associados.

Além dos acima, outros espaços de nome estão disponíveis, embora sejam de uso menos freqüente: `PF_NS` (protocolos Xerox NS), `PF_ISO` (protocolos OSI/ISO), `PF_CCITT` (protocolos do CCITT), `PF_IMPLINK` (Internet Message Processors), `PF_ROUTE` (protocolos de roteamento), etc. Para cada espaço de nomes, uma macro correspondente `AF_*` define o formato dos endereços para aquele espaço.

Por fim, deve ser escolhido o protocolo que irá efetuar a comunicação, ou seja, o mecanismo usado para a transferência dos dados. Cada protocolo é aplicável a um dado espaço de nomes e uma forma de comunicação. Algumas regras são importantes na escolha de um protocolo:

- Para ocorrer comunicação, os sockets de ambos os lados devem usar o mesmo protocolos e seus nomes devem estar no mesmo espaço de nomes.
- Cada protocolo é aplicável a uma combinação específica de forma de comunicação e espaço de nomes, e não pode ser usado em combinações inadequadas. Por exemplo, o protocolo TCP somente pode ser usado na forma de comunicação `SOCK_STREAM` e no espaço de nomes da Internet.
- Para cada combinação “forma de comunicação - espaço de nomes” existe um protocolo default, indicado pelo número 0 (zero), que é geralmente usado.

O modelo cliente/servidor

Geralmente a interação entre processos usa o modelo de comunicação Cliente/Servidor. Algumas características importantes desse modelo:

- O cliente conhece o endereço e forma de acesso ao servidor e toma a iniciativa da comunicação
- O servidor é uma entidade passiva, apenas recebendo pedidos dos cliente e respondendo aos mesmos.
- O servidor oferece um serviço específico a seus clientes
- O cliente envia uma requisição de serviço e aguarda uma resposta do servidor.
- As implementações do cliente e do servidor são independentes e autônomas; apenas as seqüências de mensagens trocadas durante a comunicação, que caracterizam o serviço, devem ser respeitadas.

Como servidor e cliente têm comportamentos distintos face à comunicação, suas implementações seguem também padrões distintos. Ambos criam sockets para se comunicar, mas operam de forma diferente.

Um cliente executa normalmente os seguintes passos para estabelecer uma comunicação com um servidor:

1. Cria um socket, usando a chamada de sistema `socket`
2. Conecta seu socket ao endereço do servidor, usando a chamada de sistema `connect`
3. Envia e recebe dados através do socket, usando as chamadas de sistema `read` e `write`
4. Encerra a comunicação, fechando o socket através da chamada `close`.

Um servidor normalmente executa os seguintes passos para oferecer serviço a seus clientes:

1. Cria um socket, usando a chamada de sistema `socket`
2. Associa um endereço a seu socket, usando a chamada de sistema `bind`.
3. Coloca o socket em modo de escuta, através da chamada de sistema `listen`.
4. Aguarda um pedido de conexão, através da chamada `accept` (que gera um descritor específico para a conexão recebida).
5. Envia e recebe dados através do socket, usando as chamadas de sistema `read` e `write`.
6. Encerra a comunicação com aquele cliente, fecha o descritor da conexão (chamada `close`).
7. Volta ao passo 4, ou encerra suas atividades fechando seu socket (chamada `close`).

Diversas variantes são possíveis no esquema acima: por exemplo, o servidor pode lançar *threads* ou processos filhos para tratar as conexões recebidas.

Chamadas de sistema

```
#include <sys/socket.h>
int socket (int namespace, int style, int protocol)
```

Cria um socket no `namespace` indicado, definindo o `style` e o `protocol` a ser usado. Retorna o descritor de arquivo do socket criado, ou -1 em caso de erro. Alguns exemplos:

```
fd = socket (PF_INET,  SOCK_DGRAM,  0) ; /* socket UDP */
fd = socket (PF_INET,  SOCK_STREAM, 0) ; /* socket TCP */
fd = socket (PF_LOCAL, SOCK_STREAM, 0) ; /* socket stream local */
```

```
#include <sys/socket.h>
int bind (int socket, struct sockaddr *addr, socklen_t length)
```

Associa um endereço ao `socket`. Os parâmetros `addr` e `length` especificam o endereço a usar; seu formato depende do espaço de nomes em uso.

```
#include <sys/socket.h.>
int connect (int socket, struct sockaddr *addr, socklen_t length)
```

Inicia uma conexão entre o socket local `socket` (lado cliente) e o socket cujo endereço está especificado nos campos `addr` and `length` (lado servidor). Bloqueia até obter uma resposta do servidor ou erro. Esta operação só é usada em sockets orientados a conexão (SOCK_STREAM).

```
#include <sys/socket.h>
int listen (int socket, unsigned int n)
```

Habilita o `socket` a aceitar pedidos de conexão, tornando-o um socket de servidor. O parâmetro n indica o tamanho da fila de pedidos de conexão pendentes (clientes excedendo a capacidade da fila terão seus pedidos de conexão recusados).

```
#include <sys/socket.h>
int accept (int socket, struct sockaddr *addr, socklen_t *length_ptr)
```

Recebe um pedido de conexão pendente (espera enquanto a fila estiver vazia). Os valores `addr` e `length_ptr` retornam informações sobre o cliente que efetuou o pedido de conexão.

Ao aceitar o pedido de conexão, a chamada `accept` cria um novo socket local para manter a conexão e retorna seu descritor (ou -1 em caso de erro). O socket original permanece disponível para atender novos pedidos de conexão, até ser fechado.

```
#include <sys/socket.h>
int getsockname (int socket, struct sockaddr *addr, socklen_t *length_ptr)
```

Retorna o endereço de `socket`, nos parâmetros `addr` e `length_ptr`. (observe que `lenght_ptr` é um ponteiro)

```
#include <sys/socket.h>
int getpeername (int socket, struct sockaddr *addr, socklen_t *length_ptr)
```

Retorna o endereço do socket ao qual `socket` está conectado, nos parâmetros `addr` e `length_ptr`.

```
#include <sys/socket.h>
int send (int socket, void *buffer, size_t size, int flags)
```

Envia os dados contidos no `buffer` de tamanho `size` através do `socket`. Alguns flags permitem modificar parâmetros do envio; se os flags forem 0 (zero), a chamada `write` pode ser usada. Retorna o número de bytes enviados, ou -1 (erro).

```
#include <sys/socket.h>
int recv (int socket, void *buffer, size_t size, int flags)
```

Recebe dados de `socket`, depositando-os no `buffer` de tamanho `size`. Alguns flags permitem modificar parâmetros do envio; se os flags forem 0 (zero), a chamada `read` pode ser usada. Retorna o número de bytes recebidos, ou -1 (erro).

```
#include <unistd.h>
int close (int filedes)
```

Fecha o socket representado por `filedes`. Antes de fechar, o sistema tenta enviar os dados pendentes (ainda não enviados).

```
#include <sys/socket.h>
int shutdown (int socket, int how)
```

Permite encerrar seletivamente um socket. O parâmetro how indica a ação a ser efetuada:

- 0 : parar de receber dados no socket. Novos dados serão descartados.
- 1 : parar de transmitir dados; descartar dados sendo enviados, não retransmitir dados perdidos e não aguardar confirmações de envio.
- 2 : parar envio e recepção de dados.

Estruturas de dados

Várias das chamadas de sistema relacionadas a sockets usam como parâmetro uma estrutura do tipo `struct sockaddr`, definida da seguinte forma:

```
#include <sys/socket.h>

struct sockaddr {
   unsigned short    sa_family;    // address family, AF_xxx
   char              sa_data[14];  // 14 bytes of protocol address
};
```

O campo `sa_family` indica o tipo de endereço a ser considerado (por exemplo, AF_LOCAL ou AF_INET).O campo `sa_data` contém o endereço do socket, e sua estrutura interna depende do espaço de nomes usado.

No caso da Internet (espaço de nomes PF_INET), o endereço de um socket é composto pelo endereço IP da interface de rede à qual ele está associado e do número da porta onde ele está conectado. Para definir esses parâmetros uma estrutura adicional struct `sockaddr_in` está definida:

```
#include <netinet/in.h>

struct sockaddr_in {
   short int          sin_family;  // Address family
   unsigned short int sin_port;    // Port number (in network byte order)
   struct in_addr     sin_addr;    // Internet address (in network byte order)
   unsigned char      sin_zero[8]; // just fill this with zeros
};
```

Como essa estrutura tem o mesmo tamanho de `sockaddr`, pode-se fazer *casting* de ponteiros de uma para outra, nos dois sentidos.

É importante observar que somente processos com UID = 0 (pertencentes ao usuário root) podem criar sockets AF_INET com número de porta inferior à constante IPPORT_RESERVED (geralmente igual a 1024).

A estrutura interna `in_addr` (Internet address) tem o seguinte formato em AF_INET:

```
#include <netinet/in.h>

struct in_addr {
   unsigned long s_addr; // 32-bit long (4 bytes)
};
```

O campo `s_addr` contém um endereço IP armazenado como 4 bytes (em *network byte order*). Para preenchê-lo pode ser usada a função `inet_aton` (vide abaixo).

Conversões de formatos

Para evitar problemas de incompatibilidade entre sistemas *big-endian* e sistemas *little-endian*, alguns campos da estrutura de endereço são armazenados em um formato padronizado conhecido como *Network Byte Order*, ou formato de rede. Quatro funções estão disponíveis para converter dados entre o formato local da máquina e o de rede:

- `short htons (short)`: converte *Host* ⇒ *Network Short*
- `short ntohs (short)`: converte *Network* ⇒ *Host Short*
- `long htonl (long)` : converte *Host* ⇒ *Network Long*
- `long ntohl (long)` : converte *Network* ⇒ *Host Long*

Além destas, as funções abaixo também são usadas:

```
#include <arpa/inet.h>
int inet_aton (const char *name, struct in_addr *addr)
```

Esta função converte um endereço IP no formato "`nnn.nnn.nnn.nnn`" para o formato binário exigido pela estrutura `in_addr` e o deposita no endereço indicado por `addr`. Além disso, algumas constantes pré-definidas podem ser usadas no campo `s_addr`:

- `INADDR_LOOPBACK`: *localhost*, ou endereço de *loopback*
- `INADDR_ANY`: *any incoming address*, usada em sockets de servidores
- `INADDR_BROADCAST`: endereço de broadcast da rede local
- `INADDR_NONE`: nenhum endereço, valor retornado como erro em algumas funções

```
#include <arpa/inet.h>
char *inet_ntoa (struct in_addr addr)
```

Efetua a conversão de um endereço no formato binário interno para seu equivalente no formato "`nnn.nnn.nnn.nnn`".

Exemplos: cliente TCP e servidor TCP.

Sockets UDP

A construção de sistemas comunicando através de sockets UDP é similar à dos exemplos apresentados em TCP, com pequenas alterações:

- O servidor simplesmente cria um socket; não é necessário colocá-lo em modo de escuta (`listen`) nem aguardar conexões (`accept`).
- Ao invés de usar as chamadas `send`/`write` e `recv`/`read`, normalmente são usadas as chamadas `sendto` e `recvfrom`, que especificam os parceiros de comunicação na hora do envio/recepção.
- Ao contrário do TCP, que transforma a comunicação em um fluxo contínuo de bytes, em UDP as fronteiras entre as mensagens são respeitadas. O receptor deve receber as mensagens com os mesmos tamanhos com que foram enviadas.
- Em UDP não há garantia de entrega de mensagens. Caso o servidor não esteja ativo, mensagens enviadas a ele são simplesmente descartadas.

As chamadas de sistema `sendto` e `recvfrom` têm a seguinte sintaxe:

```
#include <sys/socket.h>
int sendto (int socket, void *buffer, size_t size, int flags,
```

```
            struct sockaddr *addr, socklen_t length)
```

Esta chamada envia a mensagem presente em `buffer` através do `socket` para o destino especificado em `addr` e `length`. O parâmetro `size` indica o número de bytes da mensagem. Os flags permitem alterar parâmetros do envio (default em 0). O retorno é similar ao de `send`, mas somente erros locais são reportados.

É possível executar a chamada `connect` em um socket UDP, mas essa ação apenas define um destinatário *default* para todas as mensagens enviadas por aquele socket. Nesse caso, pode-se usar a chamada `send` ou mesmo `write` para enviar mensagens através desse socket UDP "conectado".

```
#include <sys/socket.h>
int recvfrom (int socket, void *buffer, size_t size, int flags,
              struct sockaddr *addr, socklen_t *length-ptr)
```

Esta chamada lê uma mensagem de `socket` e a deposita em `buffer`. O parâmetro `size` indica o tamanho do buffer (pacotes maiores que o buffer serão truncados). A identidade do remetente é retornada através dos parâmetros `addr` e `lenght_ptr`. Os erros possíveis são similares aos de `recv`.

Pode-se usar `recv` ou mesmo `read` caso não haja interesse em identificar o remetente da mensagem.

Exemplos: cliente UDP e servidor UDP.

Sockets no domínio UNIX

A criação de sockets no domínio AF_LOCAL, também conhecidos como sockets UNIX, é bastante similar à construção de sockets TCP ou UDP. As únicas diferenças perceptíveis estão na estrutura de dados que representa o endereço do socket, pois o endereço de um socket local é definido como um nome de arquivo. Nesse caso, uma estrutura adicional `struct sockaddr_un` (Unix Name) está definida:

```
#include <sys/un.h>

struct sockaddr_un {
   short int  sun_family;     // Address family
   char       sun_path[108];  // socket path in the filesystem
};
```

O campo `sun_path` deverá indicar o caminho completo para o socket no sistema de arquivos (por exemplo, "`/tmp/socket-423142`"). Isso significa que cliente e servidor devem estar acessando o mesmo sistema de arquivos local. Uma vez fechado, o socket permanece no sistema de arquivos e deverá ser explicitamente removido pelo servidor que o criou (através da chamada `unlink`).

Exemplos: cliente UNIX e servidor UNIX.

Daemons

A maior parte dos serviços de rede oferecidos por um computador é provida por processos com características especiais, denominados *daemons* (espíritos, em inglês). As principais características de um daemon são:

- têm um longo tempo de vida (podem ser relançados automaticamente caso terminem de forma inesperada);
- são lançados durante a inicialização do sistema operacional (*boot*);
- executam em *background* e não estão associados a um terminal;
- oferecem serviços de rede ou executam tarefas administrativas (*backup*, *logging*, etc);

- são de propriedade do usuário *root* ou de um usuário administrativo interno (*nobody*, *daemon*, etc).

Os *daemons* nascem como processos normais, que são em seguida convertidos em *daemons*. Os passos necessários para a transformação de um processo ordinário em um *daemon* são:

1. Duplicar o processo através de uma chamada `fork()` e encerrar o processo pai, para liberar o shell que lançou aquele processo.
2. Invocar a chamada `setsid()` para criar uma nova sessão de processos; isso faz com que o filho saia da sessão de seu pai (e do shell), sendo desvinculado do terminal e dos sinais porventura enviados ao shell.
3. Trocar o diretório atual pelo diretório de trabalho, que pode ser o diretório raiz (`/`) ou um diretório específico do daemon (por exemplo, o daemon `httpd` usa `/etc/httpd` como diretório de trabalho).
4. Fechar todos os descritores de arquivos não usados, para evitar manter aberto algum descritor herdado de seu pai ou do shell.
5. Por convenção, instalar um tratador para o sinal `SIGHUP`, que releia as configurações do daemon.

Como os daemons não estão associados a terminais, eles não podem gerar mensagens em um terminal. Duas abordagens são possíveis para a geração de mensagens:

- escrita em em um arquivo de mensagens específico do daemon (como fazem o Apache e o Samba)
- envio de mensagens para o serviço *syslog* (como faz a maioria dos daemons)

O serviço *syslog* é oferecido por um daemon, que recebe mensagens em um socket local em `/dev/log` e também em um socket UDP (porta 514). A partir das configurações descritas no arquivo `/etc/syslog.conf` ele decide o que fazer com cada mensagem recebida: enviar a um terminal, avisar o administrador, armazenar em um arquivo, ativar um script externo ou enviar a outro host são as principais possibilidades. As funcionalidades oferecidas pelo sistema *syslog* e sua forma de acesso estão definidas no manual da GLibC.

Resolução de nomes

Como visto acima, os programas que interagem através da rede usam nomes de sockets para localizar seus interlocutores. Na Internet, os sockets são nomeados através de um endereço IP e um número de porta, ambos definidos na estrutura sockaddr_in. O endereço IP de um socket é definido nessa estrutura por um inteiro long, e sua porta por um inteiro short.

Para os usuários, é muito mais fácil se referenciar a um servidor de rede através de um nome simbólico que usar seu endereço IP. Por isso, costumar ser associados nomes simbólicos aos endereços IP. Isso leva a várias representações para o mesmo endereço de socket, como mostra a tabela a seguir:

nome simbólico string	**endereço IP string**	**representação interna in_addr (long)**
espec.ppgia.pucpr.br	200.192.112.139	0xC8C0708B ou 0x8B70C0C8

Conforme visto anteriormente, as funções `inet_aton` e `inet_ntoa` permitem converter endereços IP em sua representação interna, ou vice-versa. Mas, como converter nomes simbólicos para endereços IP, ou vice-versa?

Diversos mecanismos de resolução de nomes podem ser usados por um sistema operacional para converter um nome simbólico em um endereço IP, ou o contrário. Os mecanismos mais usuais estão detalhados nesta página. Apesar da profusão de mecanismos, o sistema operacional oferece uma interface padronizada para resolver nomes, que é definida pela seguinte funções:

```
#include <netdb.h>
struct hostent * gethostbyname (const char *name)
```

Esta função retorna informações sobre o computador cujo nome é `name`, ou *null* se houver erro. Ver também `Gethostbyname2`.

```
#include <netdb.h>
struct hostent * gethostbyaddr (const char *addr, size_t length, int format)
```

Esta função retorna informações sobre o computador a partir de seu endereço. O campo `addr` é um ponteiro para um endereço em formato interno, com tamanho `size`. O campo `format` especifica o tipo de endereço (`AF_INET` para IPv4 ou `AF_INET6` para IPv6).

Essas funções podem falhar com um dos seguintes erros, indicados na variável `h_errno`:

- `HOST_NOT_FOUND`: o nome ou endereço não foi encontrado.
- `TRY_AGAIN`: um erro temporário (o servidor de nomes não pôde ser contactado, por exemplo).
- `NO_RECOVERY`: um erro não recuperável.
- `NO_ADDRESS`: existe uma entrada para o nome, mas não há endereço associado a ele.

As funções acima retornam um ponteiro para uma estrutura do tipo `hostent`, estaticamente alocada, que contém os seguintes campos:

- `char *h_name`: nome principal do host.
- `char **h_aliases`: aliases (nomes alternativos) do computador, na forma de um vetor de strings.
- `int h_addrtype`: tipo de endereço retornado (`AF_INET`, `AF_INET6`, etc).
- `int h_length`: tamanho em bytes dos endereços retornados (depende do tipo de endereço).
- `char **h_addr_list`: vetor de endereços associados ao host, terminado por *null*. Os endereços sempre estão no formato interno, em *Network Byte Order*.
- char *h_addr : sinônimo de h_addr_list[0], o primeiro endereço do host.

Como essa estrutura é alocada estaticamente, deve ser usada com muito cuidado em programas multi-thread.

Exemplos de resolução de nomes tanto direta quanto reversa.

Chamada de procedimento remoto

A chamada de procedimento remoto (RPC - *Remote Procedure Call*) é uma abstração construída sobre sockets. Ela simplifica a tarefa do programador, liberando-o das seguintes tarefas:

- criação e manutenção de sockets e seus descritores e buffers associados
- formatos de endereços e resolução de nomes e de endereços
- definição de um protocolo de comunicação para formatar dados e comandos
- preocupação com a ordem dos bytes e formato dos dados nas arquiteturas envolvidas

Nessa abstração, o cliente vê o servidor como uma interface constituída por uma ou mais funções e as estruturas de dados necessárias para definir os parâmetros das mesmas. Ao chamar uma função dessa interface, código de comunicação gerado automaticamente estabelece a comunicação com o servidor. O diagrama a seguir ilustra esse mecanismo:

Passo	cliente	biliotecas e runtime RPC	rede	bibliotecas e runtime RPC	servidor

1	`cliente()`
	`{`
	`x = 5 ;`
	`y = fatorial(x);`				

Passo	cliente	biliotecas e runtime RPC	rede	bibliotecas e runtime RPC	servidor
2		- recebe a chamada de função - prepara os parâmetros para envio - conecta com o servidor - envia o pedido com seus parâmetros			
3			⇒		
4				- recebe o pedido e parâmetros - extrai os parâmetros no formato local - invoca a função solicitada	

5					`int fatorial(int n)`
					`{`
					…
					`return(fat) ;`
					`}`

6				- recebe o valor de retorno - prepara o retorno para envio - devolve resposta ao cliente	
7			<=		
8		- recebe a resposta - extrai os valores de retorno no formato local - retorna a chamada da função			
9	`printf(“%d\n”,y);` `}`				

Somente o código do cliente e do servidor e a definição da interface devem ser escritos pelo programador da aplicação distribuída. Todo o código em rosa é gerado automaticamente pelas ferramentas que compõe o suporte a RPCs.

Em UNIX, o suporte a RPC é provido pelos seguintes componentes:

- biblioteca de RPC, que contém o suporte de comunicação e as funções de gerência de RPC.
- biblioteca XDR (*eXternal Data Representation*), que provê a codificação de todos os dados em um formato independente de plataforma.
- *daemon portmapper*, que gerencia os serviços ativos e informa aos clientes os serviços (interfaces) disponíveis e suas respectivas versões e portas.
- `rpcgen`, o compilador de descrições de interface e gerador de código de comunicação.

Um exemplo de RPC está disponível nesta página.

RPC é uma abstração muito usada para implementar diversos serviços do mundo UNIX, entre os quais podem ser citados o NFS (*Network File System*) e o NIS (*Network Information System*). O suporte a RPC é padrão no mundo UNIX, e as diversas implementações são compatíveis entre si (mas incompatíveis com as RPCs DCE e Microsoft).

Atividades

1. Compilar e executar os exemplos de clientes e servidores deste módulo, observando seus comportamentos e resultados.
2. Quando deseja obter um arquivo de um servidor Web, o browser envia a seguinte requisição ao servidor e fica aguardando uma resposta:

```
GET /path/file HTTP/1.0
(linha vazia)
```

Caso o arquivo seja encontrado, o servidor envia uma resposta com a seguinte forma e encerra a conexão:

```
HTTP/1.0 200 ok
Content-type: text/html
(linha vazia)
... (conteúdo do arquivo solicitado)
```

1. Construa um servidor Web que aceite requisições de arquivos HTML (`text/html`) e GIF (`image/gif`). Os arquivos oferecidos aos clientes devem estar em um sub-diretório `webfiles` no diretório corrente do servidor. Para mais informações sobre as mensagens do protocolo HTTP consulte esta página.

1. Construa um servidor TCP (e seu respectivo cliente) de operações matemáticas básicas. Em cada interação, o servidor recebe dois inteiros A e B e a definição de uma operação a ser realizada sobre eles (+, - * ou /), devolvendo ao cliente o resultado da operação.
2. Repita o exercício anterior usando sockets UDP.
3. Idem, usando sockets UNIX.
4. Vários servidores verificam a autenticidade de hosts através do DNS. O procedimento consiste em fazer duas consultas ao DNS e confrontar os resultados obtidos. Implemente uma ferramenta de verificação de nome que efetue esse procedimento.

```
address = gethostbyname (name1)
name2 = gethostbyaddr (address)
if (name1 equal name2)
  name1 is ok
else
  name is false
endif
```

1. Modifique o cliente Web do exemplo para informar o nome do servidor a acessar através da linha de comando.
2. Em seu servidor Web, use a chamada `gethostbyaddr` para encontrar o nome de cada cliente conectado.
3. Transformar o servidor Web construído em um *daemon*. Ele deve armazenar os registros de sua operação em um arquivo `web.log` localizado seu diretório de trabalho.