1 Projeto de *software* de clientes.[1]

1.1 Introdução

Aplicações clientes são conceitualmente mais simples que aplicações servidoras pois, normalmente, não manipulam concorrência explicita com múltiplos servidores, nem necessitam de privilégios e mecanismos de proteção especiais.

1.2 Localizando o servidor

O software cliente pode usar vários métodos para encontrar o endereço IP e o número da porta do servidor.

O cliente pode:

- ter o nome do domínio do servidor ou o endereço IP especificado como uma constante quando o programa é compilado;
- pedir ao usuário que identifique o servidor quando invoca o programa;
- obter informação sobre o servidor de um armazenamento estável (arquivo em um disco local);
- usar um protocolo separado para encontrar o servidor (uma mensagem *broadcast* ou *multicast* à qual todos os servidores respondem).

A especificação do endereço do servidor como uma constante torna o software cliente mais rápido e menos dependente do ambiente computacional local. Entretanto, toda vez que o servidor for movido de máquina o cliente deve ser recompilado. Alguns clientes usam um nome para a máquina servidora, em vez de um endereço IP, deixando que a ligação seja feita em tempo de execução. O nome genérico do servidor será adicionado como um *alias* (pseudônimo) no sistema de nomes de domínios (DNS Domain Name System). Pseudônimos possibilitam a mudança de localização de um servidor pela alteração de um *alia*s, uma vez que os clientes procurarão um nome genérico em vez de uma máquina específica. Assim, o administrador da rede poderá mudar o servidor de lugar sem ter que recompilar todos os clientes.

Armazenar endereços de servidores em arquivos torna os clientes mais flexíveis. Porém, a execução do cliente dependerá da disponibilidade do arquivo, dificultando as alterações de máquinas onde rodam os clientes.

Usar protocolos de difusão (*broadcast*) pode funcionar bem em ambientes locais e/ou pequenos. Em ambientes de grande extensão, como na Internet, a difusão não é apropriada. Mecanismos de busca dinâmicos introduzem complexidade no software cliente e no software servidor, e geram tráfego adicional na rede.

Muitos clientes especificam o servidor de forma simples: o usuário fornece a identificação do servidor, pela especificação de um argumento, quando o cliente é invocado. Clientes que aceitam o endereço dos servidores como argumento são mais genéricos, menos complexos e menos dependentes do ambiente computacional.

Alguns serviços necessitam de um servidor explícito enquanto outros podem usar qualquer servidor disponível.

Exemplo:

Ao estabelecer uma conexão remota, o usuário tem em mente o nome da máquina com a qual deseja se conectar, e o estabelecimento da sessão só faz sentido naquela máquina específica. Entretanto, se ele deseja apenas consultar a hora corrente, qualquer servidor disponível lhe servirá, e o

[1] Elaborado por Magda Patrícia Caldeira Arantes e revisado por Juan Manuel Adán Coello.

cliente pode ser projetado para mudar o nome do servidor, dentro de um grupo, até conseguir uma resposta.

1.3 Analise gramatical (*parsing*) de um argumento de endereço.

Geralmente, um usuário pode especificar argumentos na linha de comando quando invoca um programa cliente. Na maioria dos sistemas o argumento é passado ao software como uma cadeia de caracteres (*string*). O cliente usa a sintaxe do argumento para interpretar o seu significado.

Por exemplo, o usuário fornece ao cliente o nome de domínio da máquina onde o servidor opera (merlin.cs.purdue.edu) ou o endereço IP na notação de campos decimais pontuados (128.10.2.3). Para determinar se o usuário especificou um nome ou um endereço, o cliente varre o argumento verificando se ele contém caracteres alfanuméricos. Caso contenha, o usuário especificou um nome, caso contrário, o cliente assume que a forma é a decimal pontuada e o analisa de forma apropriada.

Alguns clientes necessitam de outras informações além de endereços IP e nomes de máquinas. Os clientes completamente parametrizados permitem especificar também a porta do protocolo. Este argumento adicional poderá ser tratado em conjunto com o primeiro argumento, em uma única cadeia de caracteres, ou poderá ser tratado como outro argumento separado. Um grande número de utilitários nos sistemas Unix e Linux usa argumentos separados para especificar a máquina do servidor e a porta do protocolo. Entretanto, os clientes podem escolher a sintaxe dos argumentos de forma independente.

1.4 Procurando um nome de domínio

O cliente usa a estrutura *sockaddr_in* para especificar o endereço do servidor. O endereço IP na notação decimal pontuada ou o nome do domínio na notação textual devem ser convertidos para a representação binária de 32 bits.

A conversão da notação decimal para binária e vice-versa é trivial, porém a conversão de texto para binário requer um maior esforço. A API *socketes* inclui rotinas de biblioteca para efetuar estas conversões. A função *inet_addr* recebe uma cadeia ASCII que contém um endereço IP na notação decimal pontuada e devolve o endereço equivalente na representação binária. A função *inet_ntoa* recebe um endereço no formato binário e retorna um ponteiro para uma cadeia de caracteres na notação decimal pontuada.

As funções *gethostbyname* e *gethostbyaddr* retornam um ponteiro para a estrutura *hostent* que contém, entre outras informações, o endereço IP do hospedeiro na representação binária. A função *gethostbyname* recebe uma string ASCII com o nome do domínio. A função *gethostbyaddr* recebe três argumentos conforme mostrado abaixo:

*struct hostent *gethostbyaddr(char * addr, int len, int type)*

onde *addr é um ponteiro para o endereço no formato binário,*
len é o comprimento do endereço,
type é o tipo da família de protocolo.

A declaração da estrutura *hostent* é feita no arquivo *netdb.h*.

```
struct hostent {
     char  *h_name;          /* Nome oficial do host        */
     char  **h_aliases;      /* Outros pseudônimos (alises) */
     int   h_addrtype;       /* Tipo de endereços           */
     int   h_length;         /* Comprimento do endereço     */
     char  **h_addr_list;    /* Lista de endereço           */
};
#define h_addr h_addr_list[0]
```

Os campos que contém nomes e endereços devem ser listas porque os hospedeiros podem ter várias interfaces e vários nomes e endereços. Para garantir compatibilidade com versões anteriores, o arquivo também define o identificador `h_addr` para se referir à primeira posição na lista de endereços do hospedeiro.

Todos os endereços retornados pelas funções anteriores estão na ordem de bytes da rede.

Exemplo: Conversão de nome. Para obter o endereço IP do hospedeiro merlin.cs.purdue.edu, o cliente recebe o nome do domínio e chama gethostbyname.

```
struct hostent   *hptr;
char *hnome="merlin.cs.purdue.edu";

if (hptr=gethostbyname(hnome)) {
     /*    O endereço IP está em hptr->h_adrr  */
} else {
     /*    erro, trate-o     */
};
```

Se a chamada for bem sucedida, *gethostbyname* retorna um ponteiro para uma estrutura *hostent* válida. Se o nome de domínio passado não puder ser mapeado em um endereço IP o valor retornado será zero. Examinando o valor retornado o cliente poderá determinar a ocorrência de erro.

Exemplo: Conversão de endereço usando o gethostbyaddr

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
main(int argc, const char **argv){
    u_long addr;
    struct hostent *hp;
    char **p;
    if (argc != 2) {
        (void) printf("Uso: %s Endereco_IP\n", argv[0]);
        exit (1);
    }
    if ((int)(addr = inet_addr(argv[1])) == -1) {
        (void) printf("Forma do Endereco_IP: a.b.c.d\n");
        exit (2);
    }
    hp = gethostbyaddr((char *)&addr, sizeof (addr), AF_INET);
    if (hp == NULL) {
        (void) printf("Não foram encontradas informações do hospedeiro %s\n",
                      argv[1]);
        exit (3);
    }
    for (p = hp->h_addr_list; *p != 0; p++) {
        struct in_addr in;
        char **q;
        (void) memcpy(&in.s_addr, *p, sizeof (in.s_addr));
        (void) printf("%s\t%s", inet_ntoa(in), hp->h_name);
        for (q = hp->h_aliases; *q != 0; q++)
            (void) printf(" %s", *q);
        (void) putchar('\n');
    }
    exit (0);
}
```

1.5 Procurando o número de uma porta popular por nome

A maioria dos programas clientes deve procurar a porta de protocolo para o serviço que desejam chamar. Por exemplo, clientes do servidor de *e-mail* SMTP procuram a porta associada ao SMTP antes de invocá-lo.

A função de biblioteca *getservbyname* pode ser usada para esta finalidade. A função recebe como parâmetro duas cadeias de caracteres que especificam o serviço desejado e o protocolo usado, retornando um ponteiro para a estrutura *servent* que está definida no arquivo netdb.h.

```
struct servent {
     char  *s_name;          /* Nome oficial do serviço    */
     char  **s_aliases;      /* Outros pseudônimos (alises)*/
     int   s_port;           /* Porta para este serviço    */
     char  *s_protocol;      /* Protocolo a usar           */
};
```

Se um cliente TCP precisar do número de porta oficial para o protocolo SMTP, ele chama a função *gethostbyname,* conforme descrito a seguir:

```
struct servent   *sptr;

if (sptr=getservbyname("smtp","tcp")) {
     /*    O número da porta está em sptr->s_port    */
} else {
     /*    ocorreu erro, manipule-o      */
};
```

1.6 Números de portas e a ordenação de bytes

A função *getservbyname* retorna a porta do protocolo para o serviço na ordem de byte de rede, pois este é o modo usado na estrutura *sockaddr_in*. Esta representação pode ser diferente da usada na máquina local. Se o programa cliente precisar usar este valor (*para impressão, por exemplo*) ele deve chamar as funções de conversão (*htons, ntohs, htonl, ntohl*).

1.7 Procurando um protocolo pelo nome

A interface soquete fornece um mecanismo que permite a um cliente (ou servidor) mapear o nome de um protocolo em um inteiro associado ao protocolo. A função de biblioteca *getprotobyname* efetua a procura. A chamada a esta função passa o nome do protocolo em um argumento do tipo *string* e a função *getprotobyname* retorna o endereço de uma estrutura do tipo *protent*. Se a função não puder acessar a base de dados ou se o nome especificado não existir na base de dados, o valor zero é retornado. A base de dados de nomes de protocolos permite que um sítio defina *aliases* para cada nome. A estrutura *protoent* tem um campo para o nome oficial do protocolo e um campo para a lista de *aliases*. O arquivo netdb.h contém a declaração da estrutura:

```
struct protent {
     char  *p_name;          /* Nome oficial do protocolo   */
     char  **p_aliases;      /* Outros pseudônimos (alises) */
     int   p_proto;          /* Número oficial do protocolo */
};
```

Um cliente pode procurar o número da porta oficial para o protocolo UDP chamando a função *getprotobyname* conforme descrito a seguir:

```
struct protoent  *pptr;

if (pptr=getprotobyname("udp")) {
     /*    O número da porta está em
     pptr->p_proto     */
} else {
     /*ocorreu erro, manipule-o */
};
```

1.8 Um algoritmo para cliente TCP

Construir software cliente é mais fácil que construir software servidor. Construir software cliente TCP é a tarefa mais fácil da atividade de programação em rede, pois o TCP manipula todos os problemas de controle de fluxo e confiabilidade. O algoritmo 1 pode ser usado para o projeto de um cliente TCP que efetua conexão com um servidor e usa esta conexão para a comunicação.

Algoritmo 1: Cliente Orientado a Conexão

1. Encontre o endereço IP e o número de porta de protocolo do servidor com o qual a comunicação é desejada.
2. Aloque um soquete
3. Especifique que a conexão precisa de uma porta de protocolo arbitrária, não usada na máquina local, e permita ao TCP escolher uma
4. Conecte o soquete ao servidor
5. Comunique-se com o servidor usando o protocolo do nível de aplicação (isto normalmente envolve enviar requisições e esperar pelas respostas)
6. Feche a conexão.

1. 9 Alocando um soquete

Nas seções anteriores foi discutido como encontrar o endereço IP de um servidor e a função soquete foi usada para alocar um soquete de comunicação. Clientes que usam o TCP devem especificar a família de protocolos PF_INET e o serviço SOCK_STREAM. Um programa começa com comandos de inclusão dos arquivos que contém as definições das constantes simbólicas usadas nas chamadas de funções, seguidas das declarações das variáveis usadas para guardar o descritor do soquete. Se na família de protocolos especificada pelo primeiro argumento, mais de um protocolo oferecer o serviço especificado pelo segundo argumento, o terceiro argumento deve identificar o protocolo desejado. No conjunto de protocolos da Internet apenas o TCP oferece o serviço SOCK_STREAM, então o terceiro argumento é irrelevante.

Abaixo é dado um exemplo de chamada a *socket()*:

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

int  s;    /* descritor do soquete */

s = socket(PF_INET,SOCK_STREAM,0);
```

1.10 Escolhendo um número de porta para o protocolo local

Antes de usar o soquete a aplicação cliente deve especificar os endereços local e remoto. Um servidor opera em um endereço de porta popular, que o cliente deve conhecer. Entretanto, o cliente TCP não opera em uma porta pré-associada. O cliente deve selecionar uma porta para o protocolo local e usá-la como seu endereço final. Em geral o cliente pode usar qualquer porta desde que:

1. não exista conflito entre a porta usada pelo cliente e as portas sendo usadas por outros processos da máquina;
2. a porta usada não foi associada a um serviço conhecido;

Quando um cliente precisa de uma porta local para o protocolo ele pode escolher uma porta arbitrária e verificar se ela não está em uso, até encontrar uma que atenda o critério acima. Entretanto, a interface *socket* torna a escolha de uma porta para o cliente de forma mais simples, pois ela permite que o cliente deixe o TCP fazer a escolha. Essa escolha acontece como um efeito colateral da chamada a *connect*.

1.11 Um problema fundamental relacionado à escolha do endereço IP local

No estabelecimento da conexão o cliente deve escolher um endereço IP local e um número para a porta do protocolo local. Para um hospedeiro que está ligado a uma única rede isto é trivial. Entretanto, roteadores e hospedeiros *multi-homed* possuem vários endereços IP dificultando a escolha. Em geral, a dificuldade reside em que a escolha correta depende de questões ligadas ao roteamento de pacotes e as aplicações normalmente não tem acesso a esse tipo de informação.

Imagine, por exemplo, um computador com múltiplas interfaces de rede e portanto múltiplos endereços IP. Antes de uma aplicação usar o TCP, ela deve ter um endereço terminal para a conexão. Quando o TCP se comunica com um destino externo ele encapsula cada segmento do TCP em um datagrama IP. O IP usa o endereço do destino remoto e a tabela de roteamento para selecionar o endereço do próximo nó e uma interface de rede para alcançar o próximo nó. O problema é que o endereço IP da fonte, contido no datagrama de saída, deve ser igual ao endereço IP da interface de rede através da qual o datagrama IP será roteado. Se uma aplicação escolhe o endereço IP para a máquina de forma aleatória, ela pode escolher a interface errada, ou seja, a interface na qual o IP não será roteado. Na prática, um cliente pode funcionar mesmo se o programador escolher a interface errada, pois no trajeto do servidor para o cliente, a resposta pode percorrer uma rota diferente da usada para ir até o servidor. Entretanto, usar o endereço incorreto viola a especificação, torna a gerência da rede mais difícil e confusa e faz o programa menos confiável.

Para resolver esse problema, as chamadas às primitivas *sockets* aceitam o não preenchimento dos campos de endereço IP local e permitem ao software TCP/IP a escolha automática de um endereço no instante de conexão do cliente ao servidor. Resumindo*:*

> *Devido à necessidade de interação da aplicação com o software de roteamento IP para a escolha correta de um endereço IP local, o software cliente deixa esse campo sem preenchimento e permite ao software TCP/IP selecionar automaticamente o endereço IP local e um número de porta de protocolo local que não está sendo usado por outros processo.*

1.12 Conectando soquete TCP a um servidor

A chamada ao sistema *connect* permite ao cliente TCP iniciar uma conexão. Em termos do protocolo subjacente (TCP), *connect* força o início do protocolo *de apresentação em três vias* do TCP (*3-way handshake*). A chamada ao *connect* não retorna até que uma conexão TCP ter sido estabelecida ou até que o TCP atingir um tempo máximo tentando e desistir da conexão. A chamada retorna zero se a conexão for bem sucedida ou -1 se a conexão falhar. A chamada ao *connect* pede três argumentos:

```
retcode = connect(s,remaddr,remaddrlen)
```

onde:

```
s           é o descritor do soquete;
remaddr     é o endereço da estrutura sockaddr_in que especifica o
            endereço remoto com o qual a conexão é desejada;
remaddrlen é o comprimento (em bytes) do segundo argumento.
```

A função *connect* executa quatro tarefas:

1. Testa se o soquete é válido e se ele já está conectado;
2. Preenche o endereço remoto no soquete de acordo com o segundo argumento;
3. Escolhe o endereço local para a conexão (IP e número da porta do protocolo) se o soquete não tem um;
4. Inicia a conexão TCP e retorna um valor para informar ao remetente se a chamada foi bem sucedida.

1.13 Comunicando-se com o servidor usando o TCP

Assumindo que a chamada a *connect* estabeleceu a conexão, o cliente pode usá-la para se comunicar com o servidor. Normalmente, os protocolos de aplicação especificam uma interação do tipo requisição-resposta na qual os clientes enviam uma sequência de requisições e esperam uma resposta para cada pedido.

Em geral, o cliente chama *send* para transmitir uma requisição e *recv* para esperar por uma resposta. Nos protocolos mais simples, o cliente envia uma requisição e esperam por uma resposta antes de enviar um novo pedido. O código seguinte ilustra a interação requisição-resposta, mostrando como um programa envia uma requisição por uma conexão TCP e lê a resposta.

```
/* Segmento de código para exemplo */

#define BLEN = 120          /* Comprimento do buffer em uso */
char *req="pedido de alguma coisa";
char *buf[BLEN];            /*   Resposta ao pedido       */
char *bptr;                 /*   Apontador para o buffer  */
int  n;                     /*   Número de bytes lidos    */
int  buflen;                /*   Espaço usado no buffer   */

bptr = buf;
buflen = BLEN;

/* envio de um pedido */
send(s,req,strlen(req),0);

/* leitura da resposta (pode vir em vários pedaços) */
while ((n=recv(s,bptr,buflen,0)>0) {
     bptr += n;
     buflen -= n;
}
```

1.14 Recebendo uma resposta de uma conexão TCP

No exemplo anterior, um cliente envia uma pequena mensagem para o servidor e espera por uma pequena resposta (menor que 120 bytes). O código contém uma única chamada a *send,* mas faz repetidas chamadas a *recv*. Quando a chamada *recv* retorna os dados, o contador de espaço no buffer é decrementado e o ponteiro do buffer é movido para depois do espaço lido. Iterações são necessárias na leitura dos dados, mesmo para pequenas quantidades de dados, pois o TCP não é um protocolo orientado a blocos de dados. O TCP é orientado a fluxos de dados (*stream*): ele garante a entrega de uma sequência de bytes que o remetente envia, mas não garante que a entrega será feita com o mesmo agrupamento enviado pelo remetente. O TCP pode quebrar um bloco de dados em pedaços e transmitir cada pedaço em um segmento separado, ou, alternativamente, acumular muitos bytes no buffer de saída antes de enviar um segmento. A idéia é fundamental para a programação com o TCP é:

> *Como o TCP não preserva os limites dos registros, qualquer programa que lê de uma conexão TCP deve estar preparada para aceitar os dados de poucos em poucos bytes. A regra vale mesmo se a aplicação emissora transferir os dados em grandes blocos.*

1.15 Fechando uma conexão TCP

1.15.1 A necessidade do encerramento parcial

Quando uma aplicação termina o uso da conexão completamente, ela pode chamar a função *close* para terminar a conexão cordialmente e desalocar o soquete. Terminar uma conexão não é trivial, pois o TCP permite a comunicação nas duas vias. O término de uma conexão requer coordenação entre cliente e servidor.

Por exemplo, considere que um cliente e um servidor se comunicam através de interações requisição-resposta.. O software cliente envia pedidos aos quais o servidor responde. O servidor não pode terminar a conexão porque ele não sabe se o cliente enviará pedidos adicionais. O cliente sabe que não tem mais pedidos para enviar, entretanto, não sabe se todos os dados enviados pelo servidor já foram recebidos, especialmente se operação realizada for transferência de dados, em resposta a pedidos de consulta a base de dados.

1.15.2 A operação de encerramento parcial

Para resolver o problema do término da conexão, muitas implementações da interface *sockets* incluem a primitiva *shutdown* que permite que as aplicações derrubem a conexão TCP em uma direção. A chamada ao sistema *shutdown* recebe dois argumentos, um descritor do soquete e a especificação da direção, e encerra o soquete na direção especificada.

```
errcode = shutdown(s,direction);
```

O argumento *direction* é um inteiro. Se for 0 (zero) nenhuma entrada posterior é permitida, se for 1 nenhuma saída posterior é permitida e se for 2 a conexão é derrubada nas duas direções.

A vantagem do encerramento parcial deveria ser agora clara: quando um cliente termina o envio de pedidos ele usa a primitiva *shutdown* para especificar que não têm mais dados para enviar, sem desalocar o soquete. O protocolo subjacente reporta o fechamento à máquina remota, onde a aplicação servidora recebe um sinal de fim de arquivo. Uma vez que o servidor tenha detectado o fim de arquivo, ele sabe que nenhuma requisição chegará. Após enviar sua última resposta, o servidor poderá fechar a conexão. Resumindo:

> *O mecanismo de encerramento parcial remove a ambigüidade de protocolos de aplicação que transmitem quantidades arbitrárias de informação em resposta a uma requisição. Nesses casos, o cliente faz um encerramento da conexão após a sua última requisição; o servidor fecha a conexão após enviar sua última resposta.*

1.16 Programando um cliente UDP

À primeira vista, programar um cliente UDP parece fácil. O algoritmo 2 mostra um método básico para desenvolvimento de um cliente UDP, similar ao algoritmo usado para o cliente TCP.

Algoritmo 2: Cliente Sem Conexão

1. Encontre o endereço IP e o número de porta de protocolo do servidor com o qual a comunicação é desejada.
2. Aloque um soquete
3. Especifique que a conexão precisa de uma porta de protocolo arbitrária, não usada na máquina local, e permita ao UDP escolher uma
4. Especifique o servidor para o qual devem ser enviadas mensagens
5. Comunique-se com o servidor usando o protocolo do nível de aplicação (isto normalmente envolve enviar requisições e esperar pelas respostas)
6. Feche o soquete.

Os primeiros passos do cliente UDP são muito parecidos com os passos dos clientes TCP. Um cliente obtém o endereço do servidor e o número da porta do protocolo e então aloca um soquete para a comunicação.

1.17 Soquetes UDP conectado e não conectados

Aplicações clientes podem usar soquetes UDP de dois modos básicos: conectados e não conectados. No modo conectado, o cliente usa a chamada *connect* para especificar o endereço remoto (ou seja, o IP do servidor e o número da porta do protocolo). Uma vez especificado o endereço remoto, o cliente envia e recebe mensagens como um cliente TCP. A principal vantagens de soquetes UDP é a conveniência oferecida ao cliente quando este interage com um único servidor: a aplicação precisa especificar o servidor apenas uma vez, não importando o número de datagramas que lhe serão enviados.

No modo não-conectado, o cliente não conecta o soquete com um endereço remoto específico. Em vez disso, o destino remoto é especificado a cada vez que uma mensagem é enviada. A vantagem principal de um soquete desconectado é a flexibilidade: o cliente pode esperar para decidir que servidor contatar até o momento em que tiver um pedido para enviar, e pode usar servidores diferentes para cada pedido.

> *Um Soquete UPD pode ser conectados tornando conveniente a interação com um servidor específico, ou pode ser não-conectado tornando necessário que a aplicação especifique o endereço do servidor a cada envio de mensagem.*

1.18 Usando *connect* com o UDP

Embora um cliente possa conectar um soquete do tipo SOCK_DGRAM, a chamada *connect* não inicia qualquer troca de pacotes, nem testa a validade do endereço remoto. A chamada apenas guarda a informação do endereço remoto na estrutura de dados do soquete para uso posterior. Ou seja, quando aplicado a soquetes do tipo SOCK_DGRAM, *connect* apenas armazena um endereço. Mesmo que a chamada a *connect* seja bem sucedida, isso não significa que o endereço remoto seja válido ou que o servidor seja alcançável.

1.19 Comunicação com um servidor usando UDP

Após um cliente UDP chamar *connect*, ele podem usar *send* para enviar uma mensagem e *recv* para receber uma resposta. Diferentemente do TCP, o UDP fornece a transferência de mensagem. A cada chamada a *send*, o UDP envia uma mensagem ao servidor. A mensagem contém todos os dados passados à primitiva *send*. Da mesma forma, cada chamada a *recv* retorna uma mensagem completa. Se o cliente tiver especificado um buffer de tamanho suficiente a chamada a *recv* retornará todos os dados da mensagem. Portanto, o cliente UDP não necessita fazer repetidas chamadas *recv* para obter uma mensagem.

1.20 Fechando um soquete que usa UDP

O cliente UDP chama *close* para fechar um soquete e liberar os recursos associados a ele. Uma vez fechado o soquete, todas as mensagens que chegarem endereçados à porta de protocolos deste soquete serão rejeitadas pelo software UDP. Entretanto a máquina onde o soquete foi fechado não informa o fato à máquina remota. Assim, uma aplicação que usa transporte sem conexão deve ser projetada para saber quanto tempo deve reter um soquete antes de fechá-lo.

1.21 Encerramento parcial usando um soquete

A chamada *shutdown* pode ser usada com um soquete UDP para parar uma transmissão em uma dada direção. Infelizmente, quando o *shutdown* é aplicado a um soquete UDP, nenhuma mensagem é enviada ao outro lado. A chamada simplesmente marca o soquete para não transferir dados na(s) direção(ões) especificada(s). Então, se um cliente derruba a saída no seu soquete, o servidor não receberá qualquer mensagem para indicar que a comunicação cessará.

1.22 A falta de confiabilidade do UDP

O algoritmo para cliente UDP apresentado é simplista e ignora um aspecto fundamental do UDP: a semântica que ele oferece consiste em transferência não confiável (melhor esforço) de datagramas.

Embora um cliente UDP simplista possa funcionar bem em redes locais que exibem baixa perda, pequeno atraso, e não reordenam pacotes, este algoritmo não funciona bem em inter-redes (*internets*) complexas.

Para trabalhar em um ambiente inter-rede, um cliente deve implementar confiabilidade usando temporizadores (*timeouts*) e retransmissões. Ele deve também deve tratar dos os problemas da duplicação e reordenação de pacotes. Adicionar confiabilidade pode ser difícil e requer conhecimentos de projeto de protocolos.

O software cliente que usa UDP deve implementar confiabilidade usando técnicas como seqüenciamento de pacotes e reconhecimentos, *timeouts* e retransmissões. Projetar protocolos corretos, confiáveis e eficientes para ambientes inter-redes requer uma experiência considerável.

Bibliografia

COMER, D. E., STEVENS, D. L., Internetworking With TCP/IP Volume III: Client-Server Programming and Applications, Linux/POSIX Socket Version, Prentice-Hall International 2001, *Capítulo 6*.