

Implementação e teste de protocolos com o x -kernel: passando da teoria à prática*

Minicurso apresentado durante o SBRC2000

Dorgival Olavo Guedes Neto
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais

Maio de 2000

Resumo

Protocolos de comunicação estão entre os módulos de *software* mais complexos encontrados em computadores modernos. O desenvolvimento de protocolos eficientes usualmente requer que se adotem técnicas de codificação deselegantes e altamente específicas para cada caso, o que torna o processo demorado e sujeito a erros. Pelos mesmos motivos, o desenvolvimento de trabalhos práticos em cursos de Redes de Computadores usualmente não permite aos alunos abordar detalhes de implementações reais.

Este curso visa apresentar os princípios básicos de desenvolvimento de protocolos de rede utilizando o x -kernel. Esse ambiente foi projetado para permitir o desenvolvimento de protocolos de forma padronizada, porém sem prejuízo para o desempenho final. As primitivas básicas do sistema são descritas utilizando-se um protocolo simples como exemplo, abordando todo o processo desde a obtenção do *software* até a execução de testes, passando pelas etapas de instalação e configuração do sistema.

*Este material é uma adaptação com extensões do tutorial distribuído com o x -kernel [PDB96].

Sumário

1	Introdução	3
2	Conceitos básicos	4
2.1	Modelos de processos	4
2.2	Especificação dos protocolos componentes da pilha	5
2.3	Representação das conexões	6
3	Interface entre os objetos	7
3.1	Abertura de canais de comunicação	7
3.2	Envio de mensagens	8
3.3	Recebimento de mensagens	9
3.4	Protocolos síncronos	10
4	Bibliotecas	10
4.1	Controle de execução e sincronização	10
4.2	Eventos	11
4.3	Manipulação de mensagens	11
4.4	Mapeamento de conexões	13
4.5	Listas de participantes	14
4.6	Suporte para depuração e instrumentação	14
5	Desenvolvimento de um protocolo simples	14
5.1	Os arquivos de definições	15
5.2	Inicialização	18
5.3	Abertura de conexão	19
5.3.1	Abertura ativa	20
5.3.2	Abertura passiva	22
5.4	Demultiplexação das mensagens recebidas	23
5.5	Envio e recebimento de mensagens	25
5.6	Fechamento de conexão	26
5.7	Operações de controle	27
6	Criação de uma versão executável de um protocolo	28
6.1	Obtenção	29
6.2	Configuração do ambiente	30
6.3	Compilação dos módulos do ambiente e protocolos	30
6.4	Especificação de um sistema executável	31
6.5	Configuração do grafo de protocolos	32
6.6	Configuração as “máquinas” de teste	32
6.7	Compilação e execução dos testes	33
7	Comparação com outros sistemas	34
7.1	Família BSD Unix	34
7.2	Linux	35
7.3	Nova interface FreeBSD	35
7.4	System V Streams	36

8	O <i>x</i>-kernel no ensino de Redes de Computadores	36
9	O <i>x</i>-kernel como ferramenta de pesquisa	36
9.1	Desenvolvimento de protocolos	37
9.2	Suporte do sistema operacional	37
9.3	Simulação de código real	38
10	Conclusão	38

1 Introdução

O *x*-kernel provê um ambiente baseado em objetos, com interfaces simples e elegantes, para o desenvolvimento de protocolos. Essa elegância é obtida sem prejuízo do desempenho final, tornando o sistema uma opção válida não apenas como ferramenta didática mas também como ambiente de pesquisa.

Desenvolvido originalmente no Departamento de Ciência da Computação da Universidade do Arizona, o sistema é utilizado hoje em inúmeras universidades e laboratórios de pesquisa, sendo inclusive parte de sistemas comerciais: o sub-sistema de rede do sistema operacional Digital Unix é exatamente o *x*-kernel, embutido no núcleo OSF/1 que compõe o sistema.

Segundo palavras do seu principal autor [PD96], a utilização do conceito de sistema baseado em objetos foi fundamental no desenvolvimento do ambiente, garantindo a consistência da interface. Entretanto, o sistema utiliza C como linguagem base e não C++, sendo a amarração entre os objetos provida por um sistema de tempo de execução próprio, por motivos de confiabilidade e desempenho.

Utilizando o conceito de objetos, cada protocolo é implementado como uma estrutura de dados que contém o estado do mesmo, associada a um conjunto de operações exportadas para serem utilizadas por outros objetos. Essas operações são mapeadas sobre uma interface padronizada que representa todas as principais operações de um protocolo tradicional, tais como envio e recepção de mensagens, abertura e fechamento de conexões, etc. Além dessa interface padronizada o sistema define também um conjunto de abstrações básicas necessárias em quase todo protocolo, tais como o conceito de mensagens e dicionários de dados para decisões sobre a demultiplexação de conexões, entre outras.

O restante deste texto é organizada da seguinte forma: inicialmente, a seção 2 apresenta os conceitos básicos de organização de sub-sistemas de rede em geral e do *x*-kernel em particular. Nas seções seguintes, os principais elementos da interface e do sistema de suporte à execução serão discutidos em detalhe nas seções 3 e 4, respectivamente. Esses elementos são então aplicados no desenvolvimento de um protocolo simples na seção 5, enquanto a seção 6 narra os passos necessários para instalação e execução do sistema. A seguir, a seção 7 descreve as relações entre o *x*-kernel e outros ambientes de desenvolvimento disponíveis e finalmente as seções 8 e 9 discutem aspectos de aplicação do sistema como ferramenta para ensino e pesquisa.

Considerando-se a natureza introdutória deste material, detalhes “interessantes” de implementação e pormenores de utilização do sistema não são discutidos aqui a fim de não confundir o leitor em um primeiro curso. Ao longo do texto, leitores que desejem maiores detalhes encontrarão referências a outros materiais, alguns deles distribuídos junto com o *x*-kernel, os quais podem ser consultados nesse caso.

Em particular, qualquer trabalho de desenvolvimento utilizando o *x*-kernel não deve ser iniciado sem que se obtenha uma cópia do manual do programador do sistema [Net96b], a referência definitiva em termos da interface do sistema e dos protocolos já implementados.

2 Conceitos básicos

Ao se implementar protocolos em um sistema operacional é preciso considerar-se vários aspectos do sistema e da arquitetura almejada que afetam tal implementação. Entre esses aspectos incluem-se a determinação do modelo de processos a ser utilizado, a descrição da pilha de protocolos propriamente dita e os elementos utilizados para se representar as conexões ativas.

2.1 Modelos de processos

O projetista do sub-sistema de redes deve determinar qual a organização da pilha de protocolos em termos de fluxos de execução, baseado nas primitivas oferecidas pelo sistema operacional, usualmente definidas em termos de processos ou *threads*. Há basicamente duas opções nesse caso: um processo por protocolos, ou um processo por mensagem, como ilustrado na figura 1.

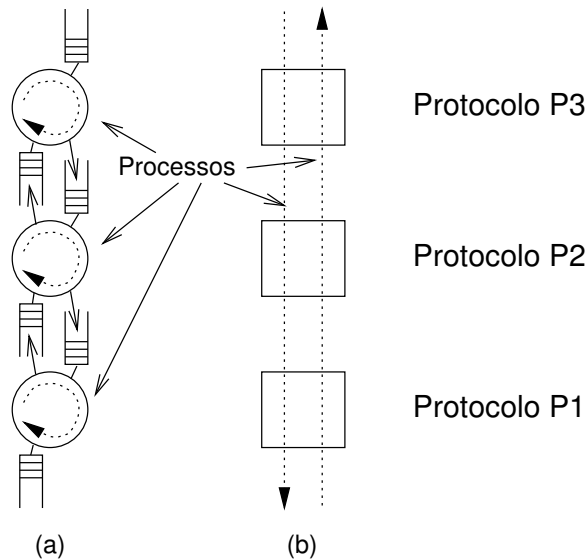


Figura 1: Modelos de processos: (a) por protocolo; (b) por mensagem

A implementação de cada protocolo por um processo (fig. 1.a) é às vezes mais simples em termos de projeto: cada protocolo pode ser definido como um processo individual, não sendo afetado pelas operações de outros protocolos. A relação direta entre processos e protocolos ajuda no desenvolvimento. Cada processo (protocolo) se comunica com os protocolos adjacentes da pilha de protocolos enviando e recebendo mensagens, as quais devem ser implementadas com base nas primitivas oferecidas pelo sistema operacional para comunicação entre processos. Apesar da aparente simplicidade, essa solução apresenta como desvantagem o custo operacional envolvido. Cada vez que uma mensagem é passada de um protocolo para outro ao longo da pilha, uma troca de contexto é necessária para suspender o processo que acaba de enviar a mensagem e permitir que o processo que recebe a mensagem tenha a oportunidade de desempenhar as tarefas associadas ao protocolo.

A outra opção nesse caso é implementar a pilha de protocolos associando-se fluxos de execução diretamente às mensagens (fig. 1.b). Dessa forma, todo o processamento relacionado ao recebimento ou envio de cada mensagem é executado sem que trocas de contexto sejam necessárias. O problema nesse caso é que um protocolo deve ser dividido de forma que diferentes fluxos de execução sejam responsáveis pela condução de mensagens “pilha acima” e “pilha abaixo”.

O *x*-kernel opta pela segunda opção, definindo protocolos como conjuntos de procedimentos a serem executados pelo processos que se movem pela pilha carregando as mensagens. Parte da complexidade envolvida nesse tipo de organização é evitada pela definição de uma interface comum a todos os níveis da pilha, de forma que cada nível se comporte de maneira semelhante em relação a seus vizinhos.

2.2 Especificação dos protocolos componentes da pilha

Outro fator a ser considerado é como especificar para o sistema quais protocolos devem ser executados, isto é, como representar a composição da pilha. Em uma organização com processos associados a protocolos isso é uma tarefa relativamente simples, pois adicionar um novo processo à pilha exige apenas que se inicie um novo processo para executar o código correspondente ao novo protocolo. Nas arquiteturas com processos associados a mensagens, entretanto, a amarração entre processos se dá pela ligação das rotinas a serem chamadas a cada nível.

Em sistemas monolíticos como o núcleo do sistema Unix BSD, a pilha é definida estaticamente na codificação do sistema. Para se adicionar, remover ou substituir um protocolo é necessário que o código da pilha seja alterado extensivamente para remover as referências ao código do protocolo e/ou adicionar as chamadas para o novo módulo. Não só o sistema não pode ser alterado durante a execução, mas mesmo alterações antes da ligação exigem que o código seja bastante modificado.

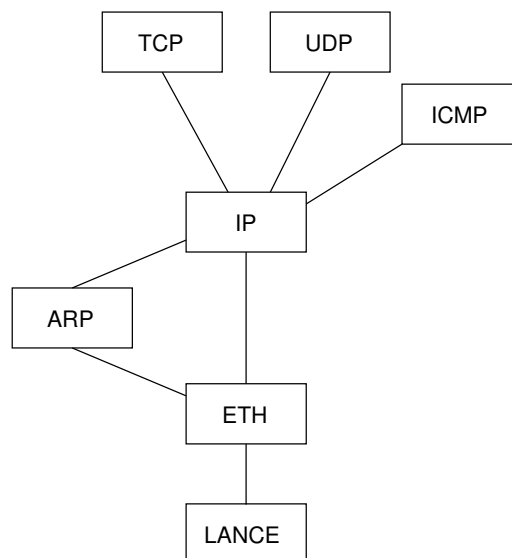


Figura 2: Um exemplo de grafo de protocolos

O *x*-kernel também utiliza uma ligação estática, porém a interface baseada em objetos oferece grande flexibilidade na configuração de novas pilhas. Uma encarnação do sistema é especificada em tempo de ligação, quando os protocolos a serem instanciados são definidos como formando um grafo de protocolos como o mostrado na figura 2 para TCP/IP. O uso da interface baseada em objetos garante que vários protocolos, uma vez definidos em termos da interface padronizada entre os objetos do sistema, possam ser combinados com grande flexibilidade. A adição de um novo protocolo a uma pilha seja feita sem que outros protocolos exijam alterações. Bastaria alterar-se o arquivo de definição do grafo, cujo formato é apresentado na figura 3. A princípio, se um novo protocolo fosse adicionado entre IP e ETH na figura, aqueles protocolos continuariam apenas acionando as funções exportadas na interface, que agora seriam ligadas ao novo protocolo pela infra-estrutura do sistema. Uma vez iniciado o sistema, entretanto, a pilha de protocolos se mantém fixa.

```
name=lance;  
name=eth protocols=lance;  
name=arp protocols=eth;  
name=ip protocols=eth,arp;  
name=icmp protocols=ip;  
name=udp protocols=ip;  
name=tcp protocols=ip;
```

Figura 3: Especificação do grafo da figura 2

O maior nível de versatilidade é atingido por sistemas que permitem que a pilha de protocolos seja alterada durante a execução do sistema. Nesses casos a aplicação poderia requerer ao sistema de execução que novos módulos sejam adicionados ao grafo dinamicamente. Alguns sistemas oferecem esta opção hoje sob certas condições, porém a definição de um sistema completamente geral para esse fim é um problema de grandes proporções. No caso específico do *x*-kernel há alguns estudos nessa área, porém todas as implementações hoje disponíveis são ainda definidas por um grafo estático.

2.3 Representação das conexões

Um protocolo define um conjunto de regras de comunicação, mas não representa um objeto ativo que personifique tal comunicação. Para que haja realmente a comunicação entre duas entidades é preciso que os sistemas envolvidos estabeleçam algum tipo de associação. No caso de protocolos orientados a conexões, é exatamente a noção da conexão que representa essa instanciação do protocolo. Em protocolos sem conexão a noção de uma instância do protocolo ainda existe, ainda que restrita a uma única mensagem ou a uma troca de mensagens limitada. O ponto importante aqui é que essa associação possui uma certa quantidade de informação associada, seja ela simplesmente a definição de um par de terminações para as mensagens ou algo mais elaborado, envolvendo a gerência de informações de estado, tais como controle de fluxo, congestionamento, etc.

Muitos sistemas não definem uma entidade específica para armazenar tais informações, que podem ser espalhadas por tabelas nos próprios protocolos, ou associadas a abstrações

a nível de aplicação, nas bordas do sub-sistema de rede, como é o caso usualmente das implementações baseadas na interface de *sockets*. Esses sistemas tendem a acumular toda a informação de estado de todos os protocolos percorridos por uma determinada mensagem associados a uma estrutura representando a extremidade do canal. Esse sistema exige que todos os protocolos envolvidos na comunicação sejam configurados de forma a compartilhar as estruturas de dados comuns corretamente.

O *x*-kernel define um objeto para representar cada associação origem-destino: a *sessão*. Uma sessão é a representação do estado daquela associação *em relação a um dado protocolo*. Por exemplo, em uma conexão TCP, o protocolo TCP terá uma sessão contendo números de portas, controle da janela deslizante, janela de congestionamento, etc. O protocolo IP por sua vez terá uma sessão contendo os endereços IP das duas máquinas envolvidas, etc. As várias sessões formadas por uma conexão são interrelacionadas de forma similar à relação entre seus protocolos.

3 Interface entre os objetos

Conforme discutido anteriormente, os objetos básicos que formam o *x*-kernel em execução são protocolos e sessões. Protocolos definem os métodos básicos envolvidos em uma comunicação, enquanto sessões fornecem uma referência para um determinado canal de comunicação aberto entre duas entidades. Estas entidades, por sua vez, podem ser representar uma aplicação ou um protocolo de nível mais alto.

Independente da identidade do elemento de nível superior, a interface definida pelo sistema se comporta da mesma forma. Entidades operam diretamente sobre os protocolos para abrir canais de comunicação com entidades de mesmo nível no destino, quando então sessões são criadas para representar esse canal de comunicação. Operações de envio e recebimento de dados são então executadas sobre as sessões criadas para esse fim.

3.1 Abertura de canais de comunicação

Uma sessão deve ser criada sempre que duas entidades desejam se comunicar, seja através de um serviço orientado a conexão, seja por um serviço sem conexão. Como em qualquer sistema de comunicação, a interface prevê tanto a abertura direta do canal, em que a entidade especifica a identidade exata do destino, quanto a abertura passiva, em que uma entidade apenas manifesta sua disponibilidade para aceitar mensagens de outras entidades que com ela desejem se comunicar. A abertura passiva corresponde ao comportamento usual de servidores, os quais disponibilizam um serviço sem identificar a identidade das entidades que com eles podem se comunicar. Por outro lado, a abertura ativa é normalmente utilizada por clientes, entidades que necessitam de algum serviço provido por outros elementos da rede e que normalmente possuem uma identificação direta ou indireta das entidades que oferecem tal serviço.

Aberturas ativas são implementadas através de uma única função:

```
Sessn xOpen(Protl hlp, Protl hlpType, Protl llp, Part *participants)
```

A utilizando esta função a entidade que deseja abrir um canal de comunicação se identifica como um protocolo de mais alto nível, `hlp`, que pretende abrir um canal de comunicação com uma entidade similar, canal este que deve ser identificado pelo conteúdo da estrutura `participants`, discutida posteriormente. Esse canal deve ser estabelecido pelo protocolo de nível inferior `llp`. A operação retorna a sessão recém-criada ou um indicador de erro, caso o canal não tenha sido criado.

Em certos casos, módulos intermediários que não representam protocolos de comunicação reais podem requisitar a abertura do canal em favor de um outro protocolo real localizado mais acima na pilha de protocolos. Nesse caso, `hlpType` identifica o protocolo real que utilizará o canal. Este artifício é utilizado por módulos denominados *protocols virtuais* que não adicionam cabeçalhos às mensagens nem geram mensagens de controle próprias, mas apenas existem no grafo de protocolos para controlar o fluxo da comunicação. Mais detalhes sobre a utilização desses módulos podem ser encontrados no manual do sistema [Net96b].

Por outro lado, aberturas passivas requerem duas operações, já que uma entidade que manifeste sua disponibilidade para aceitar a abertura de novos canais de comunicação devem posteriormente ser notificadas quando tais canais são abertos de fato:

```
XkReturn xOpenEnable(Protl hlp, Protl hlpType, Protl llp, Part *participant)
XkReturn xOpenDone(Protl hlp, Protl llp, Sessn session, Part *participants)
```

A primeira operação tem parâmetros similares aos usados para a abertura direta do canal e deve ser utilizada pela entidade desejando aceitar comunicações de outras máquinas. Já `xOpenDone` é uma operação definida por aquela entidade para ser executada pelo protocolo inferior quando do estabelecimento de um canal de comunicação. Os parâmetros nesse caso identificam o protocolo superior que deve ser notificado (`hlp`), o protocolo inferior que recebeu o pedido de estabelecimento do canal (`llp`), a sessão criada pelo protocolo inferior e finalmente uma identificação dos participantes no canal, a qual permite que o originador do pedido de abertura do canal, na sua outra extremidade, seja propriamente identificado. Ambas as operações retornam um valor que indica se a mesma pode ser concluída com sucesso.

3.2 Envio de mensagens

Uma vez aberto um canal de comunicação entre dois protocolos de mesmo nível estes podem se comunicar ativamente pelo envio de mensagens. Para isso, basta que o protocolo que deseja enviar uma mensagem requirite à sessão criada para identificar o canal no nível abaixo do seu que a mensagem seja passada pilha abaixo para que seja processada pelos níveis inferiores. A operação, ilustrada na figura 4, é executada pela operação `xPush`:

```
XkHandle xPush(Sessn lls, Msg *message)
```

Os únicos parâmetros são obviamente a mensagem a ser enviada e o identificador da sessão no nível inferior que deve providenciar a transferência. A operação retorna uma referência para a mensagem enviada que pode ser utilizada em uma posterior interação

entre o protocolo e a sessão envolvidos, por exemplo, caso o protocolo requirite uma retransmissão da mesma.

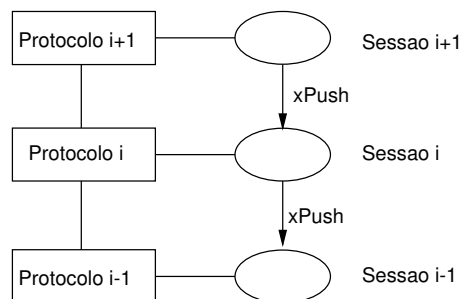


Figura 4: Enviando uma mensagem com xPush

3.3 Recebimento de mensagens

O processamento de mensagens recebidas por outro lado não é tão simples quando o envio. É preciso lembrar que sessões são criadas pelo *protocolo* de nível superior, ao qual as mesmas são associadas. A sessão inferior não dispõe de informação suficiente para identificar qual canal, entre os vários possivelmente existentes no nível superior, deve receber uma dada mensagem. Isto é devido ao fato de que a informação necessária para que essa decisão seja tomada encontra-se no cabeçalho da mensagem do protocolo superior. Para solucionar esse problema, uma sessão que tenha uma mensagem para ser entregue ao nível superior deve primeiro entregá-la ao protocolo do nível superior para que o canal apropriado seja identificado pelo protocolo. Apenas após esse passo a mensagem pode ser realmente entregue à sessão no nível superior que represente o canal correto. Tal processo é ilustrado na figura 5.

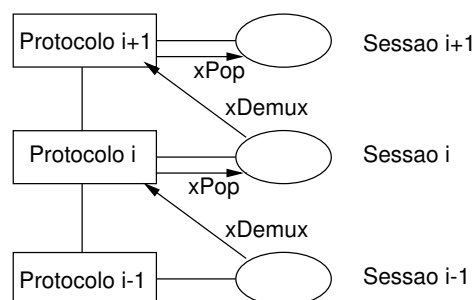


Figura 5: Procesamento de uma mensagem recebida

As operações envolvidas são então definidas como:

```
XkReturn xDemux(Protl hlp, Sessn lls, Msg *message)
```

```
XkReturn xPop(Sessn hls, Sessn lls, Msg *message, void *hdr)
```

`xDemux` é chamada pela sessão de nível inferior `lls` para entregar a mensagem indicada ao protocolo de nível superior `hlp`. É basicamente esta operação que deve remover o cabeçalho do protocolo do nível superior do início da mensagem e definir qual de suas sessões é o destino da mesma. Uma vez isto sendo determinado, `xPop` é chamada para entregar a mensagem à sessão do protocolo do nível superior. Para simplificar a implementação de operações como o envio de uma confirmação de recebimento (*acknowledgment*) para a mensagem, a sessão que recebeu a mensagem no nível anterior também é identificada. Finalmente, prevendo o caso em que o processamento da mensagem pela sessão do protocolo dependa de dados constantes do cabeçalho da mesma, este também é fornecido.

3.4 Protocolos síncronos

As operações de envio e recepção de mensagens acima são todas definidas para protocolos que enviam mensagens de assíncrona, isto é, que não suspendem qualquer processamento posterior até que uma resposta seja devolvida. Muitos protocolos, entretanto, tem exatamente esse processamento — por exemplo, os protocolos que implementam chamadas de procedimentos remotos (RPC). Para simplificar a implementação desse protocolos no *x*-kernel operações semelhantes às anteriores, porém voltadas para aplicações síncronas, também são definidas:

```
XkReturn xCall(Sessn session, Msg *request, Msg *reply)
```

```
XkReturn xCallPop(Sessn session, Msg *request, Msg *reply, void *hdr)
```

```
XkReturn xCallDemux(Protl hlp, Sessn session, Msg *request, Msg *reply)
```

Uma das principais alterações aqui é a adição de um campo para indicar a mensagem de resposta esperada/recebida. Como os protocolos de baixo nível são normalmente assíncronos, usualmente deve haver em um grafo que use protocolos síncronos um protocolo que exporte uma interface superior síncrona, porém se comunique com os protocolos inferiores de forma assíncrona [OP92].

4 Bibliotecas

A maior parte das interações entre protocolos de diferentes níveis de um grafo pode ser representada pelas operações descritas anteriormente. Para que isso seja possível o sistema de tempo de execução deve prover vários outros serviços necessários à implementação de protocolos. Entre eles podemos listar a gerência de memória para armazenamento de mensagens, primitivas para controlar o escalonamento de execução, tratamento de eventos e temporizações, entre outros. Nas seções a seguir os principais módulos fornecidos pelo ambiente de execução do *x*-kernel são apresentados.

4.1 Controle de execução e sincronização

Apesar da maior parte das operações de um protocolo poder ser especificada apenas como o processamento de uma sequência de operações para cada mensagem, em alguns

casos é preciso exprimir relações entre fluxos de instruções independentes, isto é, entre mensagens diferentes. Para esses casos o *x*-kernel provê semáforos como a primitiva básica de sincronização entre processos. As operações são as usualmente definidas para esse tipo de primitiva:

```
void semInit(Semaphore *s, int count)
void semSignal(Semaphore *s)
void semWait(Semaphore *s)
```

Um semáforo deve ser inicializado com um valor inicial (**count**) antes de ser utilizado. Uma vez inicializado, processos executando um **semWait** em um semáforo são bloqueados até que o valor do mesmo seja superior a zero, quando então o valor é decrementado e o processo continua. Um **semSignal** incrementa o valor do semáforo e libera um processos que esteja bloqueado caso o valor se torne maior que zero [HP91].

4.2 Eventos

Protocolos frequentemente definem operações a serem executadas caso um determinado evento não ocorra dentro de um certo intervalo de tempo, como por exemplo retransmissões caso um *acknowledgment* não seja recebido em tempo. O *x*-kernel permite que se definam eventos eventos como uma função para ser executada após um determinado tempo. As operações são:

```
Event evSchedule(EvFunc function, void *argument, int time)
EvCancelReturn evCancel(Event event)
void evDetach(Event event)
```

evSchedule agenda um evento para ser disparado mais tarde, quando então a função dada deve ser executada com **argument** como parâmetro. Caso o agendamento seja bem sucedido a operação retorna uma referência para o evento que pode ser manipulada pelas outras operações a fim de permitir que um evento possa ser cancelado caso não seja mais necessário, ou liberado para execução como um novo processo (**evDetach**).

4.3 Manipulação de mensagens

Protocolos tem como uma de suas operações básicas a manipulação de mensagens: cabeçalhos são adicionados a mensagens transmitidas, removidos de mensagens recebidas, mensagens são inspecionadas. Além disso, em certos casos mensagens podem ter que ser divididas em mensagens menores, bem como mensagens menores podem ter que ser agrupadas a fim de formar outras mais longas. Sendo operações frequentes, é imprescindível que sejam executadas de forma eficiente. Todas essas operações, bem como diversas formas de se criar uma mensagem, são previstas no *x*-kernel.

Protocolos podem precisar criar mensagens de diversas formas, e diferentes funções estão disponíveis para esse fim:

```

void msgConstructEmpty(Msg *message)
void msgConstructBuffer(Msg *message, char *buffer, int length)
char *msgConstructAllocate(Msg *message, int length)
void msgAssign(Msg *message_1, Msg *message_2)
void msgDestroy(Msg *message)

```

var `msgConstructEmpty` permite que uma mensagem seja criada inicialmente vazia, sem nenhuma alocação de espaço. Posteriormente esta mensagem pode vir a receber uma referência para outra, por exemplo através da operação `msgAssign`. Em outros casos, um protocolo pode desejar criar uma mensagem a partir de uma área de dados já conhecida, o que é um caso comum, por exemplo, em *device drivers*. Isso é feito utilizando-se `msgConstructBuffer`, enquanto `msgConstructAllocate` permite que se crie uma nova mensagem com um determinado espaço alocado dinamicamente para receber dados futuros. `msgDestroy` indica que um determinado processo não tem mais interesse em uma mensagem, liberando-a para ser removida caso nenhum outro processo esteja usando-a.

A manipulação de cabeçalhos é feita imaginando-se a área no início de uma mensagem existente como uma pilha: novos cabeçalhos podem ser inseridos empilhando-os à frente da mensagem atual, enquanto a remoção pode ser feita desempilhando-se um determinado número de bytes do início da mesma.

```

char *msgPush(Msg *message, int length)
char *msgPop(Msg *message, int length)

```

Ambas as operações retornam um apontador para o início da área (des)empilhada, onde o protocolo pode então ler ou escrever seus cabeçalhos.

A quebra de uma mensagem é feita fornecendo-se a mensagem original, uma nova mensagem vazia que deverá conter o novo fragmento e o tamanho desejado para o mesmo.

```

void msgBreak(Msg *original_message, Msg *fragment_message, int length)
void msgJoin(Msg *new_message, Msg *fragment1, Msg *fragment2)

```

De forma semelhante, a operação de concatenação de mensagens retorna uma referência para uma nova mensagem criada a partir dos dois fragmentos fornecidos.

Finalmente, o *x*-kernel simplifica o processo de manipulação de mensagens simplificando o tratamento de sequências de dados a serem inspecionados em uma mensagem que poderia, a princípio, ser composta de vários segmentos contendo cada um apenas parte dos dados como vetores. Para isso define-se o conceito de uma caminhada sobre os dados:

```

void msgWalkInit(MsgWalk cxt, Msg *message)
char *msgWalkNext(MsgWalk cxt, int *len)
void msgWalkDone(MsgWalk cxt)

```

`msgWalkInit` inicializa o contexto `cxt` que define o progresso da caminhada. `msgWalkNext` retorna um apontador para cada novo segmento de dados contíguos na mensagem, bem como o tamanho do segmento. Um protocolo que deseje inspecionar todos os bytes de

uma mensagem deve então executar um loop inspecionando cada segmento retornado pela função, até que a mesma retorne um apontador nulo, indicando o fim da mensagem. `msgWalkDone` finaliza o acesso, liberando o contexto.

Mensagens são implementadas no *x*-kernel como grafos acíclicos direcionados (DAGs) combinando vetores de dados que formam partes da mensagem [Mos96b]. A estrutura do DAG permite manipulação eficiente dos dados e controle sobre cópias e compartilhamento. Cada DAG mantém um contador de referências que é atualizado nas operações apropriadas, como criação, atribuição, concatenação, etc. Idealmente, cada mensagem deve ser destruída (com `msgDestroy`) pelo mesmo processo (idealmente, no mesmo procedimento) que a criou. Outros protocolos que necessitem armazenar uma cópia de uma mensagem, por exemplo para o caso de uma possível retransmissão futura, devem criar uma cópia da mesma (que basicamente incrementará o contador de referências) e depois destruir a cópia.

4.4 Mapeamento de conexões

Durante o processo de demultiplexação protocolos precisam inspecionar alguns campos do cabeçalho para aquele protocolo e determinar se aquele conjunto de campos já está associado a uma sessão existente, ou uma abertura passiva. Tal processo é tão comum que o ambiente oferece uma implementação otimizada de um tipo dicionário de dados para esse tipo de operação. Um dicionário de dados pode ser criado com a operação:

```
Map mapCreate(int number, int size)
```

Onde `number` indica o número de entrada a serem criadas no dicionário (na verdade, uma tabela de *hash* com uma cache de um acesso [Mos96a]) e `size` define o tamanho da chave a ser utilizada, o que permite que otimizações sejam aplicadas à comparação de chaves.

Uma vez criado um dicionário, as operações a seguir permitem que uma associação (chave,identificador) seja armazenada, bem como que uma consulta seja feita ao dicionário a fim de determinar se uma dada chave possui um valor associado a ela associado.

```
Binding mapBind(Map map, void *key, void *id)
```

```
XkReturn mapResolve(Map map, void *key, void **id)
```

Usualmente o identificador usado nesses casos é uma referência a uma sessão à qual mensagens podem ser entregues.

Associações podem vir a ser removidas do dicionário posteriormente:

```
XkReturn mapRemoveBinding(Map map, Binding binding)
```

```
XkReturn mapRemoveKey(Map map, void *key)
```

A primeira operação remove a associação diretamente, enquanto a segunda requer uma busca no dicionário para localizar a chave.

4.5 Listas de participantes

Os procedimentos para abertura de canais utilizam um objeto do tipo `Part` para identificar as entidades envolvidas em uma comunicação. Uma lista de participantes é implementada como uma pilha ou par de pilhas onde são empilhadas informações que sirvam para configurar os pares de protocolos em cada nível do grafo, em termos das terminações do canal. As operações a seguir permitem que se inicialize uma pilha de participantes, verifique-se o número de identificadores nela empilhados, insira-se e remova-se identificadores:

```
void partInit(Part *participants, int number)
int partLength(Part *participants)
void partPush(Part *participant, char *address, int length)
char *partPop(Part *participant)
```

Por exemplo, uma lista de participantes identificando uma conexão passiva de um servidor aceitando conexões TCP/IP em um porto `N`, originadas apenas de um endereço IP `XX.XX.XX.XX`, teria esses dois valores empilhados, com `N` no topo, uma vez que o primeiro processo a inspecionar a pilha seria TCP, seguido por IP, que removeria o último elemento da pilha.

4.6 Suporte para depuração e instrumentação

Cada protocolo deve obrigatoriamente definir uma variável global utilizada para definir diferentes níveis de instrumentação. Essa variável deve ser da forma `tracexp`, onde `x` deve ser substituído pelo nome do protocolo.

Ao longo do código do protocolo, comandos de instrumentação devem seguir a forma

```
xTracen(xp, TR_nivel, "string de formato", n parâmetros)
```

Macros definem os símbolos `xTrace0` até `xTrace6`. Novamente, `x` deve ser substituído pelo nome do protocolo. Esta macro causa uma chamada à função `printf` com a *string* de formato especificada e os `n` parâmetros que a seguem, sempre que a variável de depuração `tracexp` tiver um valor igual ou superior ao valor especificado por `TR_nivel`. O sistema define vários valores para essas macros, tais como `TR_ALWAYS`, `TR_ERRORS`, `TR_FULL_TRACE`, etc., sendo função do programador decidir qual nível usar para cada mensagem inserida no código.

O nível de depuração para cada protocolo do grafo utilizado pode ser definido nos arquivos de configuração do sistema no momento da geração do *kernel*, permitindo que se altere facilmente o nível de instrumentação do sistema. Macros semelhantes permitem também a execução condicional de código em função do nível de instrumentação..

5 Desenvolvimento de um protocolo simples

Para exemplificar melhor a utilização de cada elemento do ambiente, consideremos o desenvolvimento de um protocolo extremamente simples que implementa mensagens sem confirmação sobre IP. Conexões são identificadas por uma tupla contendo:

(IP local, IP remoto, porto local, porto remoto)

Logo, a demultiplexação no protocolo exemplo se fará em termos dos números de portos locais e remotos. Na verdade, tal protocolo já existe no sistema usualmente distribuído, sendo chamado ASP (*a simple protocol*).

Apesar da extrema simplicidade do protocolo, que praticamente não implementa nenhum serviço útil, a quantidade de código a ser desenvolvida é relativamente grande, como será visto nas seções a seguir. Isso se deve aos vários detalhes de implementação que devem ser considerados mesmo em um caso tão simples. A técnica básica de desenvolvimento no *x*-kernel é, então, desenvolver novos protocolos a partir de algum protocolo já existente que implemente algumas das funcionalidades básicas desejadas. Esse é na verdade mais um bom motivo para se estudar o protocolo ASP: ele é usualmente um bom ponto de partida para qualquer desenvolvimento.

5.1 Os arquivos de definições

A primeira coisa a se definir são as características do sistema que devem ser visíveis para outros protocolos dentro do *x*-kernel. No caso de ASP, a única informação sobre seu funcionamento interno que pode ser útil a outros protocolos é a definição do tipo utilizado como identificador de seus portos, uma vez que outros protocolos que operem sobre ASP precisam saber que informação esse novo protocolo espera em seu campo na lista de participantes. O conteúdo do arquivo `asp.h` é então aquele da figura 6.

```
typedef u_short ASPport;
```

Figura 6: Arquivo `asp.h`: configuração geral

Além das definições exportadas para outros protocolos, é necessário definir todos os componentes do protocolo que podem ser necessários para a operação do ambiente de execução e para o desenvolvimento do próprio protocolo. Neste caso, com um protocolo tão simples, todo o código pode ser colocado em um único arquivo, simplificando a compilação. No caso de protocolos mais complexos que poderiam estar divididos em vários arquivos, é importante identificar bem cada identificador exportado por um arquivo que pode ser útil a outros.

A figura 7 contém a primeira parte do arquivo, com definições de tipos e constantes. Algumas das características principais do protocolo estão definidas ali.

O cabeçalho de mensagens do protocolo contém o número dos portos local e remoto usados na comunicação, bem como o comprimento da mensagem. O tipo que define o estado do protocolo tem dois dicionários, um dos quais será usado para armazenar a informação sobre sessões abertas, enquanto o outro manterá identificadores para as aberturas passivas catalogadas. O estado de cada sessão contém apenas um cabeçalho do protocolo que será utilizado para preencher o cabeçalho de cada mensagem a ser enviada por uma dada sessão.

```

/*
 * asp_internal.h
 */
#include "xkernel.h"
#include "ip.h"
#include "asp.h"

/* ASP message header definition */

typedef struct header {
    ASPport sport;      /* source port */
    ASPport dport;     /* destination port */
    u_short ulen;      /* ASP length */
} ASPhdr;

#define HLEN (sizeof(ASPhdr))

/* protocol and session states */

typedef struct pstate {
    Map activemap;
    Map passivemap;
} ProtlState;

typedef struct sstate {
    ASPhdr hdr;
} SessnState;

/* active and passive maps */

typedef struct {
    Sessn lls;
    ASPport localport;
    ASPport remoteport;
} ActiveId;

typedef ASPport PassiveId;

#define ACTIVE_MAP_SIZE 101
#define PASSIVE_MAP_SIZE 23

```

Figura 7: Arquivo asp_internal.h: tipos e constantes

O dicionário de conexões abertas deve garantir um mapeamento baseado na tupla mencionada anteriormente, que inclui os endereços IP. Mas ASP não tem acesso a essa informação, interna ao protocolo inferior. Esse problema é solucionado acrescentando-se um apontador para a sessão IP, uma vez que IP define uma sessão para cada par de endereços (origem,destino). O dicionário de aberturas passivas mapeará apenas um número de porto local. Em ASP, um servidor simplesmente se cadastra como pronto para receber quaisquer mensagens para um porto X, logo esta é toda a informação necessária para catalogar aberturas passivas.

```

/*
 * asp_internal.h (continuacao)
 */

/* UPI function declarations */

void asp_init(Protl);
static Sessn aspOpen(Protl, Protl, Protl, Part *);
static XkReturn aspOpenEnable(Protl, Protl, Protl, Part *);
static XkReturn aspDemux(Protl, Sessn, Msg *);
static XkHandle aspPush(Sessn, Msg *);
static XkReturn aspPop(Sessn, Sessn, Msg *, void *);
static Sessn aspCreateSessn(Protl, Protl, Protl, ActiveId *);
static XkReturn aspClose(Sessn);
static int aspControlProtl(Protl, int, char *, int);
static int aspControlSessn(Sessn, int, char *, int);
static Part *aspGetParticipants(Sessn);

/* internal function declarations */

static void getproc_protl(Protl);
static void getproc_sessn(Sessn);
static long aspHdrLoad(ASPHdr *, char *, long);
static void aspHdrStore(ASPHdr *, char *, long);

/* trace variable used in conjunction with xTrace stmts */
int traceaspp;

```

Figura 8: Arquivo `asp_internal.h`: funções e variáveis

O restante do arquivo de definições é apresentado na figura 8. Para cada operação principal definida no *x*-kernel que ASP deseja definir o protocolo define uma função correspondente, substituindo o *x* do nome da operação pelo nome do protocolo (`asp`). (Não, não é coincidência: um dos motivos do nome *x*-kernel é que ele permite que se crie facilmente um *kernel* ASP, TCP/IP, NFS, etc., bastando trocar-se o *x* pelo ambiente desejado...)

Note-se que a maior parte dos símbolos são estáticos. Toda a inicialização é feita pela rotina `asp_init`, que monta a estrutura dos objetos para este protocolo. Além dela, apenas a variável de nível de instrumentação, `traceaspp`, é globalmente visível.

5.2 Inicialização

A rotina de inicialização, `asp_init`, é executada pelo sistema de tempo de execução durante a configuração do *kernel* a fim de inicializar o grafo de protocolos. Antes que qualquer operação possa ser levada a cabo pelos protocolos, todos os componentes do grafo terão suas rotinas de inicialização executadas. Conforme pode ser observado na figura 9, as primeiras operações da função visam inicializar o estado do protocolo, criando os dicionários.

```

void
asp_init(Protl self)
{
    ProtlState *pstate;
    Protl      llp;
    Part      part;

    getproc_prot1(self);

    /* create and initialize protocol state */
    pstate = X_NEW(ProtlState);
    bzero((char *) pstate, sizeof(ProtlState));
    self->state      = (void *) pstate;
    pstate->activemap = mapCreate(ACTIVE_MAP_SIZE, sizeof(ActiveId));
    pstate->passivemap = mapCreate(PASSIVE_MAP_SIZE, sizeof(PassiveId));

    /* find lower level protocol and do a passive open on it */
    llp = xGetProt1Down(self, 0);
    if (!xIsProt1(llp))
        Kabort("ASP could not get lower protocol");
    partInit(&part, 1);
    partPush(part, ANY_HOST, 0);
    if (xOpenEnable(self, self, llp, &part) == XK_FAILURE) {
        xTrace0(asp, TR_ALWAYS,
            "asp_init: openenable on lower protocol failed");
        xFree((char *) pstate);
        return;
    }
}

```

Figura 9: `asp_init`: inicialização do protocolo

Depois disso o protocolo verifica se há um outro protocolo corretamente instalado abaixo dele. Se isso é verdade, ASP inicializa uma lista de participantes apenas com a máscara indicando que aceita conexões de qualquer endereço IP e chama `xOpenEnable` para se registrar com IP.

A primeira operação da função de inicialização não foi descrita ainda. A figura 10 apresenta o código associado à função `getproc_prot1`.

```

static void
getproc_protl(Protl p)
{
    /* fill in the function pointers to implement protocol operations */
    p->open      = aspOpen;
    p->openenable = aspOpenEnable;
    p->demux     = aspDemux;
    p->controlprotl = aspControlProtl;
}

```

Figura 10: `getproc_protl`: inicialização do protocolo

O objetivo daquela função é exatamente inicializar a infra-estrutura de objetos do grafo, associando os campos do registro que define o protocolo às funções que implementam tais operações para o caso particular do protocolo ASP. Para um protocolo, as principais operações definidas são aquelas relacionadas à abertura de conexões e demultiplexação.

5.3 Abertura de conexão

Aproveitando a descrição da inicialização da infra-estrutura de objetos para um protocolo da seção anterior, iniciamos esta seção apresentando o código correspondente para o caso de uma sessão (figura 11). As operações principais nesse caso são aquelas associadas ao fluxo de uma conexão ativa: envio e recepção de dados, bem como o fechamento da mesma.

```

static void
getproc_sessn(Sessn s)
{
    /* fill in the function pointers to implement session operations */
    s->push = aspPush;
    s->pop = aspPop;
    s->controlsessn = aspControlSessn;
    s->getparticipants = aspGetParticipants;
    s->close = aspClose;
}

```

10

Figura 11: `getproc_sessn`: inicialização da sessão

A função de inicialização das operações da sessão será chamada toda vez que uma nova sessão for criada, o que é feito pela operação `aspCreateSessn`, mostrada na figura 12. A chamada de `getproc_protl`, entretanto, não é feita diretamente por aquela operação, mas indiretamente pela função `xCreateSessn`. Isto é feito para garantir que o *x*-kernel receba toda a informação de que necessita para inserir a sessão no grafo de protocolos e sessões.

Após a inicialização da estrutura o procedimento registra a nova sessão no dicionário de sessões ativas com a chave (recebida como parâmetro), preenche o cabeçalho que será

copiado para todas as mensagens enviadas.

```

static Sessn
aspCreateSessn(Protl self, Protl hlp, Protl hlpType, ActiveId * key)
{
    Sessn s;
    ProtlState *pstate = (ProtlState *) self->state;
    SessnState *sstate;
    ASPhdr *asph;

    /* create the session object and initialize it */
    s = xCreateSessn(getproc_sessn, hlp, hlpType, self, 1, &key->lls);
    s->binding = mapBind(pstate->activemap, key, s);
    sstate = X_NEW(SessnState);
    s->state = (char *) sstate;

    /* create an ASP header */
    asph = &(sstate->hdr);
    asph->sport = key->localport;
    asph->dport = key->remoteport;
    asph->ulen = 0;

    return s;
}

```

Figura 12: aspCreateSessn: inicialização da sessão

Essas rotinas de configuração de sessões serão utilizadas tanto nas aberturas ativas quanto na finalização de aberturas passivas.

5.3.1 Abertura ativa

No caso de aberturas ativas, o protocolo de nível superior a ASP deve fornecer duas estruturas de identificação de participantes, uma para a ponta local da conexão, outra para a ponta remota. Ao receber as estruturas a identificação dos portos local e remoto a serem usados por ASP para a conexão estão no topo das duas pilhas e são de lá removidos. Com a informação restante nas estruturas de participantes ASP tenta abrir uma conexão pelo protocolo inferior.

Se a conexão pelo protocolo inferior é completada com sucesso, ASP completa o campo da chave para o dicionário representando a tupla identificadora da conexão e tenta localizá-la no dicionário de conexões ativas. Se a busca é bem sucedida já há uma conexão aberta para a tupla dada e a nova conexão é descartada. Caso contrário a conexão pode ser aberta e aspCreateSessn é chamada para completar o preenchimento do registro da sessão e inserir a nova chave no dicionário.

```

static Sessn
aspOpen(Protl self, Protl hlp, Protl hlpType, Part *p)
{
    ActiveId    key;
    Sessn       asp_s, lls;
    ProtlState *pstate = (ProtlState *) self->state;

    bzero((char *) &key, sizeof(key));

    /* high level protocol must specify both local and remote ASP port */
    key.remoteport = *((ASPport *) partPop(p[0]));
    key.localport  = *((ASPport *) partPop(p[1]));

    /* attempt to open session on protocol below this one */
    lls = xOpen(self, self, xGetProtlDown(self, 0), p);
    if (lls != ERR_SESSN) {
        key.lls = lls;
        /* check for this session in the active map */
        if (mapResolve(pstate->activemap, &key, (void **) &asp_s) == XK_FAILURE) {
            /* session wasn't already in map, so initialize it */
            asp_s = aspCreateSessn(self, hlp, hlpType, &key);
            if (asp_s != ERR_SESSN) /* A successful open! */
                return asp_s;
        }
        /* if control makes it this far, an error has occurred */
        xClose(lls);
    }
    return ERR_SESSN;
}

```

Figura 13: aspOpen: abertura ativa de conexão

5.3.2 Abertura passiva

A abertura passiva exige uma manipulação de mais informação de estado, uma vez que a intenção de um protocolo de aceitar uma conexão deve ser registrada para utilização futura. Como pode ser visto na figura 14, a primeira providência é remover o número do porto a ser usado da lista de participantes. Se uma busca no dicionário de aberturas passivas utilizando o número do porto falha isso indica que uma abertura passiva para esse porto ainda não ocorreu e o procedimento prossegue para preencher um objeto do tipo **Enable** com a informação obtida. Esse objeto é definido pelo *x*-kernel exatamente para ser usado em dicionários de aberturas passivas. Ele registra o protocolo superior requisitando a abertura e a identificação da nova associação no dicionário.

```

static XkReturn
aspOpenEnable(Protl self, Protl hlp, Protl hlpType, Part * p)
{
    PassiveId key;
    ProtlState *pstate = (ProtlState *) self->state;
    Enable *e;

    key = *((ASPport *) partPop(*p));

    /* check if this port has already been openenabled */
    if (mapResolve(pstate->passivemap, &key, (void **) &e) != XK_FAILURE) {
        if (e->hlp == hlp) {
            /* this port was openenabled previously by the same hlp */
            e->rcnt++;
            return XK_SUCCESS;
        }
        /* this port was openenabled previously by a different hlp - error */
        return XK_FAILURE;
    }
    /* this will be a new enabling, so create and initialize Enable object, */
    /* and enter the binding of port/enable object in the passive map */
    e = X_NEW(Enable);
    e->hlp = hlp;
    e->hlpType = hlpType;
    e->rcnt = 1;
    e->binding = mapBind(pstate->passivemap, &key, e);
    if (e->binding == ERR_BIND) {
        xFree((char *) e);
        return XK_FAILURE;
    }
    return XK_SUCCESS;
}

```

Figura 14: aspOpenEnable: abertura passiva de conexão

Se ao testar a presença da associação no dicionário no início do procedimento uma

associação é encontrada, os outros campos do objeto **Enable** são verificados para verificar se uma nova abertura passiva pelo mesmo protocolo foi feita. Se esse é o caso, a nova abertura é contabilizada na associação e o procedimento retorna com sucesso. Se a associação indicava que o porto já havia sido aberto por outro protocolo a função retorna com erro.

5.4 Demultiplexação das mensagens recebidas

Toda vez que o protocolo abaixo de ASP recebe uma mensagem que ele identifica como sendo direcionada para o mesmo, aquele protocolo ativa a função `xDemux` do sistema para o protocolo superior, o que neste caso causa a ativação de `aspDemux` para demultiplexação da mensagem. A primeira tarefa da função é remover o cabeçalho da mensagem, que é transformado em um buffer isolado e processado pela função `aspHdrLoad` (fig. 15).

```

static long
aspHdrLoad(ASPHdr * hdr, char *src, long len)
{
    /* copy from src to hdr, then convert network byte order to hnot order */
    bcopy(src, (char *) hdr, HLEN);
    hdr->ulen = ntohs(hdr->ulen);
    hdr->sport = ntohs(hdr->sport);
    hdr->dport = ntohs(hdr->dport);
    return HLEN;
}

```

10

Figura 15: `aspHdrLoad`: manipulação de cabeçalhos

O processamento do cabeçalho envolve não só a cópia do buffer da mensagem (possivelmente desalinhado) para um registro do tipo correto como também a conversão do formato de representação de rede para o formato interno do computador, o que em alguns casos pode resultar em uma troca de posição dos bytes. Isso é feito transparentemente pelas macros `ntoh`.

O processamento por `aspDemux` prossegue como ilustrado na figura 16. De posse do cabeçalho bem formado monta-se a chave para procurar-se pela conexão (identificada pela tupla descrita anteriormente) no dicionário de conexões ativas. Caso a busca é bem sucedida uma conexão já existe e basta entregar a mensagem e o cabeçalho à função `aspPop` para processamento, o que é feito através da função do ambiente `xPop`.

O processamento fica mais complicado se uma associação com uma conexão ativa não é encontrado. Nesse caso, duas coisas podem ocorrer: não há nenhuma conexão passiva à espera de mensagens naquele porto, ou há alguma abertura passiva. No primeiro caso a mensagem é sumariamente descartada e a função retorna. Caso contrário uma nova sessão é criada e `xDuplicate` é chamado para indicar que a sessão inferior agora é referenciada por mais uma sessão neste protocolo. Em seguida, `xOpenDone` é chamado para o protocolo superior, notificando-o de que uma abertura passiva foi completada e quando essa notificação é completada a nova mensagem é entregue para a nova sessão para ser processada.

```

static XkReturn
aspDemux(Protl self, Sessn lls, Msg * dg)
{
    char      *buf;
    ASPhdr    h;
    ActiveId   activeid;
    ProtlState *pstate = (ProtlState *) self->state;
    Sessn      s;
    PassiveId  passiveid;
    Enable     *e;
    10

    /* extract the header from the message */
    buf = msgPop(dg, HLEN);
    if (buf == NULL)
        return XK_FAILURE;
    aspHdrLoad(&h, buf, HLEN);

    /* construct a demux key from the header */
    bzero((char *) &activeid, sizeof(activeid));
    activeid.lls      = lls;
    20
    activeid.localport = h.dport;
    activeid.remoteport = h.sport;

    /* see if demux key is in the active map */
    if (mapResolve(pstate->activemap, &activeid, (void **) &s) == XK_FAILURE) {
        /* didn't find an active session, so check passive map */
        passiveid = h.dport;
        if (mapResolve(pstate->passivemap, &passiveid, (void **) &e) ==
            XK_FAILURE) {
            return XK_SUCCESS; /* drop the message */
            30
        }
        /* port was enabled, so create a new session and inform hlp */
        s = aspCreateSessn(self, e->hlp, e->hlpType, &activeid);
        if (s == ERR_SESSN)
            return XK_SUCCESS;
        xDuplicate(lls);
        xOpenDone(e->hlp, self, s);
    }
    /* found (or created) an appropriate session, so pop to it */
    return xPop(s, lls, dg, &h);
    40
}

```

Figura 16: aspDemux: demultiplexação de mensagens recebidas

5.5 Envio e recebimento de mensagens

Continuando o processamento de pacotes recebidos iniciado na seção anterior, a figura 17 mostra o código do procedimento `aspPop`, disparado por `aspDemux` no caso de conexões existentes. Em um protocolo tão simples quanto ASP não há muito a ser feito: a função apenas usa a informação do cabeçalho para truncar a mensagem para o tamanho correto.

```

static XkReturn
aspPop(Sessn self, Sessn lls, Msg * msg, void *hdr)
{
    ASPhdr *h = (ASPhdr *) hdr;

    /* truncate message to length shown in header */
    if (h->ulen - HLEN < msgLength(msg))
        msgTruncate(msg, (int) (h->ulen-HLEN));

    /* pass the message to the next protocol up the stack */
    return xDemux(xGetUp(self), self, msg);
}

```

Figura 17: Recepção de mensagens

Uma vez a mensagem tendo sido colocada em seu tamanho correto ela é entregue ao protocolo superior a ASP pela ativação da operação `xDemux` sobre aquele protocolo.

O envio de mensagens é implementado pela função `aspPush` como ilustrado na figura 18. Um novo cabeçalho é criado como uma cópia daquele armazenado no estado da sessão, reduzindo a montagem do mesmo ao preenchimento do campo de tamanho.

```

static XkHandle
aspPush(Sessn self, Msg * msg)
{
    SessnState *sstate = (SessnState *) self->state;
    ASPhdr    hdr;
    char      *buf;

    /* create a header by inserting length into header template */
    hdr      = sstate->hdr;
    hdr.ulen = msgLength(msg) + HLEN;

    /* attach header to message and pass it on down the stack */
    buf = msgPush(msg, HLEN);
    aspHdrStore(&hdr, buf, HLEN);
    return xPush(xGetSessnDown(self, 0), msg);
}

```

Figura 18: Envio de mensagens

A função `msgPush` abre espaço à frente da mensagem para que o novo cabeçalho seja inserido, o que é feito pela função `aspHdrStore` (fig. 19), de comportamento semelhante à função `aspHdrLoad`, discutida anteriormente.

```

static void
aspHdrStore(ASPHdr * hdr, char *dst, long len)
{
    /* convert host byte order to network order, then copy from hdr to dst */
    /* (note: argument 'hdr' is changed by the following code) */
    hdr->ulen = htons(hdr->ulen);
    hdr->sport = htons(hdr->sport);
    hdr->dport = htons(hdr->dport);
    bcopy((char *) hdr, dst, HLEN);
}

```

Figura 19: `aspHdrStore`: manipulação de cabeçalhos

5.6 Fechamento de conexão

Finalmente, entre as operações utilizadas durante o fluxo normal de uma conexão, `aspClose` (fig. 20) completa o conjunto. Quando o protocolo superior requisita que uma conexão seja fechada ASP remove a associação da sessão de seu dicionário de sessões ativas, requisita o fechamento de sessões inferiores que porventura existam apenas em função da sessão que está sendo fechada e finalmente libera a sessão.

```

static XkReturn
aspClose(Sessn s)
{
    ProtlState *pstate = (ProtlState *) xMyProtl(s)->state;

    /* remove this session from the active map */
    mapRemoveBinding(pstate->activemap, s->binding);

    /* close the lower level session on which it depends */
    xClose(xGetSessnDown(s, 0));

    /* de-allocate the session object itself */
    xDestroySessn(s);

    return XK_SUCCESS;
}

```

Figura 20: `aspClose`: fechamento de conexão

A chamada a `xDestroySessn` não destrói a sessão obrigatoriamente. Se outras referências existem para a mesma em protocolos superiores a destruição real pode ser adiada

até que outros protocolos liberem a sessão.

5.7 Operações de controle

O *x*-kernel permite que protocolos superiores executem certas operações sobre protocolos inferiores a fim de obter informações sobre o estado de protocolos e sessões a eles relacionadas ou para controlar detalhes de operação dos protocolos ou sessões.

```

static int
aspControlSessn(Sessn self, int opcode, char *buf, int len)
{
    SessnState *sstate = (SessnState *) self->state;
    ASPhdr    *hdr;

    hdr = &(sstate->hdr);
    switch (opcode) {
    case GETMYPROTO:
        checkLen(len, sizeof(long));
        *(long *) buf = sstate->hdr.sport;
        return sizeof(long);
    case GETPEERPROTO:
        checkLen(len, sizeof(long));
        *(long *) buf = sstate->hdr.dport;
        return sizeof(long);
    case GETMAXPACKET:
    case GETOPTPACKET:
        checkLen(len, sizeof(int));
        if (xControlSessn(xGetSessnDown(self, 0), opcode, buf, len) <
            sizeof(int))
            return -1;
        *(int *) buf -= HLEN;
        return sizeof(int);
    default:
        return xControlSessn(xGetSessnDown(self, 0), opcode, buf, len);
    }
}

```

Figura 21: Operações de controle: `aspControlSessn`

Conforme pode ser visto na figura 21, sessões ASP reconhecem códigos de controle para retornar alguns valores de operação, em particular os números de porto locais e remotos. Outros comandos requerem a determinação do mesmo parâmetro para o nível inferior a fim de que o mesmo seja adequadamente ajustado em função das necessidades do protocolo em questão.

Finalmente, quaisquer comandos não reconhecidos em um nível são repassados para processamento pelo nível inferior, até que um nível seja atingido onde o comando seja reconhecido, ou não hajam outros níveis válidos. Um operação similar é definida a nível

de protocolo. No caso de ASP, devido à sua simplicidade, não há grandes operações possíveis, apenas um sub-conjunto daquelas definidas para as sessões.

```

static Part *
aspGetParticipants(Sessn self)
{
    Part      *p;
    int      numParts;
    SessnState *sstate = (SessnState *) self->state;
    long     localPort, remotePort;

    p = xGetParticipants(xGetSessnDown(self, 0));
    if (!p)
        return NULL;
    numParts = partLength(p);
    if (numParts > 0 && numParts <= 2) {
        if (numParts == 2) {
            localPort = (long) sstate->hdr.sport;
            partPush(p[1], (void *) &localPort, sizeof(long));
        }
        remotePort = (long) sstate->hdr.dport;
        partPush(p[0], (void *) &remotePort, sizeof(long));
        return p;
    } else
        /* Bad number of participants */
        return NULL;
}

```

Figura 22: Manipulação de *participantes*

Além das operações de controle apresentadas, protocolos no *x*-kernel devem também ser capazes de satisfazer consultas sobre as listas de participantes que identificam uma dada sessão. A implementação da operação (fig. 22) mostra como isso pode ser feito. Primeiramente os níveis inferiores são consultados a fim de obter-se as listas de participantes para os mesmos. Uma vez de posse dessa informação o procedimento apenas empilha nas listas a informação local obtida do estado da sessão.

6 Criação de uma versão executável de um protocolo

Consideremos agora o processo de instalação e execução do *x*-kernel. Esse processo envolve duas etapas principais: obtenção e instalação do ambiente básico e configuração de um *kernel* a ser executado [Net96a]. Neste estudo consideramos que o ambiente será instalado em diretórios do sistema, para ser utilizado por vários usuários, o que constitui o caso mais comum. O sistema pode também ser instalado em um diretório pessoal para uso individual, o que basicamente exige alterações apenas na definição do diretório alvo.

O processo de instalação exige aproximadamente 25 MB de espaço em disco disponível. O total após instalação pode ser um pouco menor.

6.1 Obtenção

Suponhamos que se deseje instalar o ambiente em um diretório global, como por exemplo, `/pkg/xkernel`. Uma vez criado o diretório, deve-se armazenar nele o arquivo de distribuição do *x*-kernel, que pode ser obtido diretamente dos servidores da Univerdidade do Arizona através da url:

```
ftp://ftp.cs.arizona.edu/xkernel/xkernel.tar.Z (aproximadamente 7 MB)
```

Uma vez o arquivo estando instalado deve-se prosseguir inicialmente à sua descompressão e extração dos arquivos. Supondo que o arquivo acima foi armazenado no diretório onde se deseja criar o sistema, isso seria feito pela sequência de comandos:

```
cd /pkg/xkernel uncompress /tmp/xkernel.tar.Z tar xf /tmp/xkernel.tar
```

Após esses passos, o diretório `/pkg/xkernel` conterà todos os arquivos fonte do sistema e o arquivo `tar` pode ser removido. A hierarquia recém-construída deve ocupar aproximadamente 16 MB.

A atual distribuição do ambiente permite sua instalação para execução como processos a nível de usuário, onde cada “máquina” de uma configuração de rede é representada por um processo de usuário executando a pilha de protocolos da mesma, bem como um simulador. No primeiro caso, processos representando cada máquina podem realmente ser executados em máquinas separadas, utilizando a rede física para se comunicar. No segundo caso, uma arquitetura genérica pode ser criada com várias máquinas e uma topologia de rede qualquer como um único processo simulador, o qual então executa a simulação completamente no espaço do processo, sem acesso à rede. Nesta discussão abordamos principalmente o ambiente de execução no espaço do usuário. O simulador é discutido na seção 9.

Entre os sub-diretórios criados no diretório de instalação, os seguintes merecem destaque:

- **bin**: contém algumas ferramentas específicas do sistema, necessárias na compilação dos módulos;
- **doc**: os principais documentos sobre o *x*-kernel, distribuídos em L^AT_EX, Postscript e HTML;
- **include**: arquivos de definição de interfaces;
- **pi**: arquivos independentes da arquitetura, comuns ao simulador; e ao ambiente a nível de processos do usuário;
- **protocols**: a definição de todos os protocolos distribuídos com o sistema, que incluem todo o conjunto TCP/IP, Sprite RPC e outros;
- **simulator**: o ambiente do simulador;
- **user_level**: o ambiente de execução como processos do usuário.

As próximas etapas cuidarão da configuração e compilação dos módulos do sistema contidos na distribuição. Como mencionado anteriormente, consideramos aqui que o ambiente de processos será a arquitetura instalada.

6.2 Configuração do ambiente

Para proceder a configuração do ambiente deve-se criar um diretório que servirá de base para o processo de compilação, o qual deve ser colocado abaixo do diretório `user_level/build`. O sistema pode ser compilado e configurado para ser usado por máquinas de arquiteturas diferentes, por isso a estrutura de diretórios deve diferenciar entre as opções possíveis. Duas das opções mais comuns são hoje a utilização em estações Sun SPARC com o sistema operacional Solaris e PCs de arquitetura Intel rodando Linux. No primeiro caso o diretório criado deve ser `user_level/build/solaris`, enquanto no segundo deve ser `user_level/build/linux-x86`. Nesta discussão adotamos o segundo caso como exemplo.

Deve-se então criar o novo diretório e nele instalar os arquivos usados no processo de compilação, como se segue:

```
mkdir /pkg/xkernel/user_level/build/linux-x86
cd /pkg/xkernel/user_level/build/linux-x86
cp ../Template/Makefile.linux-x86 Makefile
chmod u+w Makefile
```

Em seguida é necessário configurar corretamente o `Makefile` para utilizar a hierarquia de diretórios criada. Basta editar o arquivo e substituir a linha

```
XRT = ../../..
```

Esta linha define a raiz dos diretórios do x-kernel e funciona corretamente para diretórios de trabalho dentro do diretório `build`, como criados acima. Porém, para permitir que o sistema seja também configurado e compilado em diretórios de usuários, fora da hierarquia `/pkg/xkernel`, o ideal é que a referência seja substituída por um nome absoluto. No caso em questão, a linha deve ser substituída por:

```
XRT = /pkg/xkernel
```

6.3 Compilação dos módulos do ambiente e protocolos

Em geral, é necessário adicionar os diretórios `/pkg/xkernel/bin` e `/pkg/xkernel/bin/linux-x86` (ou o correspondente para outras arquiteturas) ao início do seu conjunto de diretórios com executáveis para que as versões de utilitários distribuídos com o *x*-kernel sejam utilizados ao invés das versões do sistema. Isso pode ser feito adicionando-se esses diretórios ao conteúdo de sua variável de ambiente `PATH`. A forma de fazer isso depende da interface de comandos utilizada. Para shells derivadas da Bourne shell isso pode ser feito pela sequência:

```
PATH=/pkg/xkernel/bin:/pkg/xkernel/bin/linux-x86:$PATH export PATH
```

Obviamente o segundo diretório deve ser escolhido de acordo com a arquitetura sendo considerada.

As versões mais novas de Linux tem apresentado problemas com esse sistema. Como muitos dos utilitários necessários já estão disponíveis por definição naquele sistema, apenas

nesse caso tem se mostrado mais vantajoso adicionar-se os diretórios acima ao final da variável `PATH`. Em alguns sistemas (por exemplo, SuSE Linux 6.1) esta é a única opção válida. Alguns dos utilitários distribuídos com o *x*-kernel não funcionam corretamente (gerando até `segmentation fault`) e as versões nativas do Linux são suficientes. Nesses casos a configuração acima deve ser substituída por:

```
PATH=$PATH:/pkg/xkernel/bin:/pkg/xkernel/bin/linux-x86 export PATH
```

Uma vez tendo acrescentado os diretórios de utilitários, pode-se finalmente proceder à compilação do kernel.

```
cd /pkg/xkernel/user_level/build/linux-x86
make system
```

O primeiro passo é obviamente redundante caso a sequência de comandos anteriores tenha sido executada, pois esse o diretório corrente já será o desejado. Dependendo da máquina utilizada o processo de compilação pode exigir desde alguns segundos até vários minutos.

O processo de compilação pode gerar várias mensagens de advertência sobre arquivos não encontrados nesse ponto, as quais podem ser seguramente ignoradas, desde que nenhuma mensagem de erro (que interrompa o processo de compilação) aconteça.

Caso a compilação termine sem erros, o sistema estará completamente configurado, pronto para ser utilizado.

6.4 Especificação de um sistema executável

Como exemplo de geração de um sistema executável, tomemos como base o protocolo TCP. A pilha de protocolos nesse caso deverá ser da forma descrita na figura 2, porém acrescida de um protocolo de teste sobre TCP que se comportará como uma aplicação. Em uma máquina o protocolo de teste se comportará como um servidor, recebendo mensagens e contabilizando-as, enquanto em outra ele se comportará como um cliente, enviando mensagens para o servidor. Para isso, é preciso definir o grafo de protocolos e criar os executáveis para as duas máquinas. Para simplificar o controle dos arquivos, cada executável será gerado em um diretório separado.

Como o sistema será executado como processos a nível de usuário, sem acesso direto ao dispositivo de rede, o tráfego entre as duas “máquinas” definidas é feito sobre uma conexão UDP criada junto ao sistema operacional. O protocolo inferior no grafo, que em outros casos seria um *device driver* real, neste caso é substituído por um pseudo-dispositivo, `SIMETH`, responsável por prover uma interface idêntica a uma interface Ethernet sobre a conexão UDP.

Consideremos agora o caso onde os executáveis serão gerados em um diretório do usuário, por exemplo, `/home/dorgival/work/xk`, que consideraremos como já existente. O primeiro passo requer que os arquivos de configuração sejam copiados para o diretório de trabalho:

```

cd /home/dorgival/work/xk
cp /pkg/xkernel/user_level/build/linux-x86/Makefile Makefile
cp /pkg/xkernel/user_level/build/Template/graph.comp .
mkdir client server
cp /pkg/xkernel/user_level/build/Template/rom.client client/rom
cp /pkg/xkernel/user_level/build/Template/rom.server server/rom
chmod 664 Makefile graph.comp client/rom server/rom

```

6.5 Configuração do grafo de protocolos

O arquivo `graph.comp` distribuído no diretório `Template` já descreve exatamente o grafo de protocolos necessário neste caso. É preciso editá-lo para alterar o diretório referenciado na sua última linha, que deve ser alterado de

```
prottbl=/cs/x33/etc/prottbl.std; para prottbl=/pkg/xkernel/etc/prottbl.std;
```

O arquivo `prottbl.std` possui identificadores para todos os arquivos reconhecidos no núcleo, os quais são necessários para a demultiplexação de protocolos como IP e para controle do ambiente de execução durante a montagem do grafo. Cada novo protocolo desenvolvido deve ser adicionado àquela tabela, ou a uma cópia local da mesma (desde que o *path* seja alterado no arquivo `graph.comp`). No caso de TCP e do protocolo de teste, como ele foi distribuído junto com o sistema, identificadores já existem para os mesmos. Outros protocolos deveriam receber entradas de forma similar ao que é feito no arquivo para o protocolo ASP.

6.6 Configuração as “máquinas” de teste

Os arquivos `rom` contém parâmetros de configuração específicos de cada “máquina” a ser criada, neste caso cliente e servidor. A única configuração necessária é a definição dos endereços UDP a serem usados pelas duas instâncias de `SIMETH` (uma em cada processo) para se comportarem como interfaces ethernet. Tal informação será inserida como dados da tabela do protocolo ARP em cada processo. Cada configuração é da forma

```
simeth 1234 # simulated Real Real # IP address IP address Port number arp
128.1.2.3 192.12.69.186 1234 arp 128.1.2.4 192.12.69.35 9876
```

A primeira linha indica a `SIMETH` que ele deve se conectar ao porto UDP de número 1234. Essa informação será usada juntamente com as linhas seguintes para determinar o endereço IP associado a essa “máquina”.

Em cada linha seguinte, o primeiro número identifica o endereço IP *virtual* visto pelos processos executando em cada máquina, enquanto os campos seguintes indicam um porto em uma máquina real onde um processo *x-kernel* utilizando `SIMETH` estará conectado, recebendo mensagens. O que isso significa é que cada vez que um processo em uma instância no *x-kernel* desejar comunicar-se com a máquina (virtual) 128.1.2.3 no exemplo acima, `SIMETH` deve enviar mensagens UDP para o porto 1234 da máquina 192.12.69.186. Os endereços reais em cada arquivo `rom` devem ser alterados para conter os endereços reais

das máquinas onde os processos serão executados. Esses endereços IP podem inclusive ser iguais, indicando que os dois processos serão executados em uma mesma máquina, desde que os números de porto sejam distintos. Em uma execução real, por exemplo, os endereços utilizados foram:

```
arp 128.1.2.3 192.168.0.1 1234 arp 128.1.2.4 192.168.0.1 9876
```

Caso os números de portos não sejam alterados, a configuração de SIMETH em cada arquivo pode permanecer inalterada.

6.7 Compilação e execução dos testes

Uma vez completadas todas as alterações pode-se gerar o executável para o kernel, que será o mesmo utilizado por cliente e servidor (já que ambos possuem o mesmo grafo de protocolos):

```
cd //home/dorgival/work/xk
make compose
make depend
make
```

Alguns avisos sobre arquivos não encontrados podem ocorrer durante o `make compose`, o que é normal.

Uma vez gerado o executável, pode-se iniciar a execução dos dois processos para simular cliente e servidor. O servidor deve ser iniciado primeiro, já que ele é o responsável pela abertura passiva. O único parâmetro necessário neste caso é a indicação de que o protocolo de teste deve se comportar como o servidor, o que é indicado pela opção `-s`:

```
cd /home/dorgival/work/xk/server
../xkernel -s
```

Uma vez que o servidor esteja no ar o cliente pode ser iniciado. O parâmetro neste caso indica ao

```
cd /home/dorgival/work/xk/client
../xkernel -c128.1.2.3
```

O protocolo de testes neste caso está configurado para abrir uma conexão TCP do cliente para o servidor, por onde o cliente enviará sequências de mensagens de diferentes tamanhos para o servidor, o qual simplesmente as ecoará de volta para o cliente. O tempo gasto para trocar 100 mensagens de cada tamanho é exibido na tela. Uma possível saída seria da forma:

```
Protocol: TCP
Time: Fri Jun 6 16:16:11 1997
Host: cicada.CS.Arizona.EDU
Participant: client
Round Trips: 100
```

```
Message Length (bytes): 1
Times (sec):
    0.114381
    0.090574
    0.320991
```

```
Message Length (bytes): 200
Times (sec):
    0.101033
    0.100527
```

Neste caso, o protocolo de testes levou 0.114381 segundos para enviar as primeira 100 mensagens de 1 byte, 0.090574 para o segundo conjunto, etc., depois iniciou a transmissão de mensagens de 200 bytes, com tempos na casa de 0.101033 segundos. O teste prossegue testando mensagens maiores até o tamanho de 16 KB, ou até ser interrompido pelo usuário.

7 Comparação com outros sistemas

Como mencionado anteriormente, o estudo do *x*-kernel é útil não só para o uso da ferramenta em si, como também para facilitar o entendimento da operação de outros sub-sistemas de rede em sistemas operacionais com código aberto, como Linux e a família BSD, bem como ambientes que permitem o desenvolvimento de protocolos de rede em sistemas comerciais.

7.1 Família BSD Unix

Sendo um sistema de código aberto e gratuito, versões de Unix derivadas do sistema operacional BSD, como netBSD e freeBSD oferecem uma boa opção para trabalhos na área de sub-sistemas de rede, sendo largamente utilizados em pesquisa e ensino. A interface de instalação e configuração do sistema requer maior conhecimento sobre sistemas Unix que o seu mais direto competidor, Linux, o que justifica sua menor aceitação.

Entretando, do ponto de vista do sub-sistema de rede esse sistema operacional tem poucos rivais entre os sistemas em produção. A implementação de TCP/IP da família BSD tem sido considerada por anos como o padrão de referência na indústria. O desempenho é dos melhores registrados e vários livros documentam todos os detalhes da implementação.

Sendo um sistema de origem mais antiga, a implementação é bem distante da encontrada no *x*-kernel. Cada pilha é implementada como um conjunto de código interrelacionado, onde cada protocolo precisa conhecer exatamente as operações definidas por seus

vizinhos. Mesmo assim é um código muito bem planejado, sendo razoavelmente elegante. A única parte do sistema que se assemelha mais diretamente ao *x*-kernel é o módulo de manipulação de mensagens, denominadas *mbufs*, que tem princípios semelhantes (se bem que mais limitados) que os do *x*-kernel.

Apesar dessas limitações, o domínio dos conceitos de modularidade do *x*-kernel com certeza simplificam as tarefas de análise e desenvolvimento de protocolos no BSD Unix.

7.2 Linux

O enorme sucesso do sistema operacional Linux, sendo também um sistema com código aberto e gratuito, tem levado pesquisadores e professores a utilizá-lo como ferramenta de trabalho. Apesar de eficiente e relativamente bem projetado, o código do sistema é muitas vezes confuso e mal comentado. A implementação da pilha TCP/IP contém alguns bugs e erros de implementação de menor importância e é extremamente confusa.

Entretanto, o sub-sistema de rede em si, é de relativa elegância, aparentemente tendo utilizado vários conceitos criados pelo *x*-kernel. A interface entre protocolos é definida através de uma camada baseada em objetos, onde cada protocolo deve definir um conjunto de operações padrão que deve ser identificado através de pontos de acesso em um registro que define as características de cada objeto. A interface de manipulação de mensagens se assemelha aos *mbufs* dos sistemas BSD, porém com diferenças a nível de implementação.

Em suma, apesar dos protocolos existentes no núcleo Linux não serem particularmente claros e bem escritos, a interface de desenvolvimento de protocolos em si tem algumas idéias interessantes que podem facilitar o desenvolvimento de protocolos seguindo os princípios do *x*-kernel.

7.3 Nova interface FreeBSD

A nova versão do sistema freeBSD, (freeBSD 4.0, recentemente lançada) parece incluir uma nova infra-estrutura para desenvolvimento de protocolos de rede, aparentemente com várias características semelhantes ao ambiente do *x*-kernel. Entre elas está a definição de protocolos como objetos com interfaces bem definidas que podem ser acessadas pelo protocolos superiores ou inferiores sem necessidade de conhecimento exato dos detalhes de implementação internos a cada nível.

Aparentemente, entretanto, a interface entre objetos não é fixada *a priori*, como no *x*-kernel, mas sim definida de forma modular entre protocolos que desejem compartilhar uma certa funcionalidade. Apesar de restringir a capacidade de combinação de objetos devido à exigência de que os mesmos tenha interfaces comuns, esse enfoque pode simplificar certas tarefas na interface.

Infelizmente até a edição deste material não foi possível obter-se maiores detalhes sobre os recursos dessa nova interface, mas a perspectiva é de que a inclusão de princípios de modularização baseados em interface de objetos possa vir a facilitar o desenvolvimento de protocolos no freeBSD, tornando a tarefa mais simples, nos moldes do que é possível no *x*-kernel.

7.4 System V Streams

O sistema Unix System V e seus derivados (Sun Solaris, inclusive) oferecem a abstração de *streams* que são basicamente cadeias de fluxo de execução dentro do núcleo, associadas a mensagens, que podem ser manipuladas pelos usuários. Dada uma certa cadeia, programadores podem inserir módulos na mesma, os quais são ativados pela passagem de mensagens, de forma bastante semelhante ao que é implementado no *x*-kernel com *xPush* e *xDemux/xPop*.

A interface entretanto não é tão flexível quanto aquela definida no *x*-kernel e por limitações de segurança muitos dos recursos da interface não estão ao alcance do programador em um sistema em operação, exigindo privilégios de super-usuário para serem acessados.

8 O *x*-kernel no ensino de Redes de Computadores

Como demonstrado anteriormente, o desenvolvimento de protocolos reais com desempenho competitivo no *x*-kernel é uma realidade, reduzindo o problema de implementação e interfaceamento com um ambiente de tempo de execução. O código desenvolvido, entretanto, ainda possui uma complexidade inerente que dificulta sua utilização como ferramenta para desenvolvimento de trabalho, especialmente em disciplinas a nível de graduação.

O que se faz para reduzir esse impacto inicial de utilização do sistema, como mencionado anteriormente, é a definição de novos protocolos em função de outros módulos mais simples já disponíveis. Um desses módulos é exatamente o protocolo ASP discutido na seção 5. A partir de um protocolo mais simples pode-se definir trabalhos onde os alunos devam adicionar novas características ao mesmo. Um exemplo disso já utilizado na disciplina de Redes de Computadores na Universidade do Arizona no passado é a definição de um trabalho a ser desenvolvido ao longo do semestre que se inicia com a simples execução do protocolo ASP, sendo estendido aos poucos para incluir:

- entrega com controle de fluxo com confirmação simples, transformando ASP em um protocolo do tipo *stop-and-go*;
- extensão do protocolo para incluir mecanismos de retransmissão, transformando-o em um protocolo confiável;
- substituição do *stop-and-go* por um mecanismo de janela deslizante;
- inclusão de mecanismos de controle e/ou prevenção de congestionamento.

Outros projetos possíveis incluem a implementação de protocolos de mais alto nível, tais como RPC, HTTP, XTP, protocolos para vídeo, etc.

9 O *x*-kernel como ferramenta de pesquisa

Como ferramenta de pesquisa o *x*-kernel oferece um ambiente flexível e elegante para implementação de protocolos, porém sem prejuízo do desempenho final, permitindo até a

validação de protocolos em ambientes de produção. As principais vertentes de pesquisa que se beneficiam do uso do sistema são principalmente o desenvolvimento e validação de novos protocolos, pesquisas na área de desenvolvimento de novos recursos do sistema operacional para o suporte a sub-sistemas de rede e a análise de comportamento de protocolos reais por simulação.

9.1 Desenvolvimento de protocolos

Existem módulos de definição de dispositivos no *x*-kernel que permitem que protocolos sejam desenvolvidos com acesso direto à interface de rede real, como por exemplo o protocolo `ethpkt` no *x*-kernel rodando sobre Linux e o protocolo `nit` sobre Solaris/SunOS. Com tais protocolos é possível evitar o processamento pela pilha de protocolos convencional do sistema, permitindo que protocolos desenvolvidos no *x*-kernel interajam com outras implementações em outras máquinas. O sistema ainda pode ser desenvolvido e depurado utilizando-se um ambiente mais controlado como aquele a nível de usuário com SIMETH, sendo depois colocado em execução na rede real sem nenhuma alteração além da troca do módulo do dispositivo.

O *x*-kernel teve papel importante por exemplo na definição do protocolo IPSEC: uma das duas implementações de referência exigidas pela IETF (*Internet Engineering Task Force*) foi implementada utilizando-se o *x*-kernel na Universidade do Arizona. O processo de implementação do protocolo, bastante complexo devido a protocolos de criptografia e gerência de chaves, foi completado em menos de dois meses por dois pesquisadores em tempo parcial.

9.2 Suporte do sistema operacional

Muita atenção tem sido dispensada ao estudo de formas de se melhorar o desempenho dos protocolos e serviços definidos para redes modernas através da melhoria das abstrações definidas pelo sistema operacional para serem utilizadas pelo sub-sistema de rede. O *x*-kernel foi em si um grande passo nessa área quando de seu lançamento, provando que protocolos podem ser implementados eficientemente em um sistema real sem perda da modularidade e organização. Vários trabalhos já utilizaram o sistema como ponto de partida para o estudo de novas abstrações a nível do ambiente em tempo de execução ou da organização do sistema operacional como um todo.

Uma limitação (e ao mesmo tempo uma vantagem) do *x*-kernel é que a interface entre todos os níveis torna explícita a operação em termos de mensagens recebidas ou enviadas, em contraste com a interface de `sockets` oferecida por todos os sistemas comerciais. O uso da interface do *x*-kernel permite que novos conceitos e abstrações possam ser criados sem as restrições impostas por `sockets`. Porém, por outro lado, a falta de uma interface de `sockets` completa sobre o *x*-kernel evita que o mesmo seja facilmente combinado com aplicações tradicionais.

Por exemplo, é possível se desenvolver novas soluções para o problema de servidores HTTP escaláveis utilizando o *x*-kernel como ambiente de desenvolvimento, porém é difícil

reaproveitar soluções já testadas e implementadas para as interfaces de sistemas operacionais tradicionais.

9.3 Simulação de código real

O crescimento do tráfego na Internet e o surgimento de novas aplicações com novas exigências de serviços tem chamado a atenção da comunidade para os problemas de comportamento da arquitetura que se tornou um padrão de fato para redes de computadores, TCP/IP. Hoje já é claro que TCP, por exemplo, tem grandes problemas em certos tipos de ambientes e aplicações. Por exemplo, TCP não funciona bem em redes sem fio, nem em redes locais muito rápidas com muitas conexões simultâneas.

O *x*-kernel oferece também, além do ambiente de execução a nível de usuário, um simulador detalhado, *x-sim* [BBP96], capaz de executar o código real desenvolvido para um dado protocolo. Com isso pode-se estudar a implementação real de do protocolo em questão, ao invés de ter que ser recorrer a descrições aproximadas na linguagem do simulador. Redes de topologias variadas podem ser expressas com simplicidade e o comportamento dos protocolos pode ser acompanhado com um nível de detalhe que não é igualado por nenhum outro ambiente.

Esse simulador já foi empregado no desenvolvimento de novos mecanismos de controle de congestionamento em redes TCP/IP e vem sendo utilizado na análise de técnicas de *traffic shaping* aplicadas a múltiplas conexões e no estudo do comportamento de TCP/IP em redes sem fio.

10 Conclusão

Bibliografia

- [BBP96] Lawrence Brakmo, Andy Bavier, and Larry Peterson. *x-sim User's Manual (Version 1.0)*. Network Systems Research Group, Department of Computer Science, University of Arizona, Tucson, AZ, 1996.
- [HP91] Norman Hutchinson and Larry Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, janeiro 1991.
- [Mos96a] David Mosberger. *Map Library Design Notes*. Network Systems Research Group, Department of Computer Science, University of Arizona, Tucson, AZ, 1996.
- [Mos96b] David Mosberger. *Message Library Design Notes*. Network Systems Research Group, Department of Computer Science, University of Arizona, Tucson, AZ, 1996.
- [Net96a] Network Systems Research Group, Department of Computer Science, University of Arizona, Tucson, AZ. *Getting Started with the x-kernel*, 1996.

- [Net96b] Network Systems Research Group, Department of Computer Science, University of Arizona, Tucson, AZ. *The x-kernel Programmer's Manual (Version 3.3)*, 1996.
- [OP92] Sean O'Malley and Larry Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, maio 1992.
- [PD96] Larry Peterson and Bruce Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann, 1996.
- [PDB96] Larry Peterson, Bruce Davie, and Andy Bavier. *x-kernel Tutorial*. Network Systems Research Group, Department of Computer Science, University of Arizona, Tucson, AZ, 1996.