UNIVERSIDADE DO VALE DO RIO DOS SINOS

ALAN BARONIO MENEGOTTO

Implementação e Avaliação de um Protocolo Multicast no Kernel GNU/Linux

Monografia apresentada como requisito parcial
para a obtenção do grau de
Bacharel em Ciência da Computação

Prof. Dr. Marinho Pilla Barcelos
Orientador

São Leopoldo, novembro de 2006

“If I have seen farther than others,
it is because I stood on the shoulders of giants.”
— Sir Isaac Newton

AGRADECIMENTOS

Agradeço primeiramente ao orientador deste trabalho, Marinho Pilla Barcellos, pelo tempo despendido em reuniões, conselhos e revisões de texto e código. Aprendi muito com ele durante este período de convivência. Agradeço também a profa. Tatiane Ghedini pelas sugestões e revisões de texto, assim como a profa. Ana Paula Lüdtke Ferreira e o ex-colega (já graduado) Felipe W. Damásio pelo formato Latex criado. Finalizando, agradeço aos amigos "lá da Canoas", "lá da Unisinos", o pessoal do trabalho, a namorada e os familiares pela compreensão, paciência e apoio fornecidos.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ACK	ACKnowledgment
AFS	Andrew File System
ATM	Asynchronous Transfer Mode
ATO	Acknowledgment TimeOut
BSD	Berkeley Software Distribution
DCCP	Datagram Congestion Control Protocol
DMA	Direct Access Memory
ECN	Explicit Congestion Notification
FACK	Forward Acknowledgment
FIN	FINalize
FTP	File Transfer Protocol
HFSC	Hierarchical Fair Service Curve
IEEE	Institute of Electrical and Electronics Engineers
IGMP	Internet Group Management Protocol
IP	Internet Protocol
IPSEC	IP SECurity
IPX	Internetwork Packet eXchange
IRDA	Infra-Red Data Association
LAPB	Link Access Procedure Balanced
MD4	Message-Digest Algorithm 4
MD5	Message-Digest Algorithm 5
MFTP	Multisource File Transfer Protocol
MIB	Management Information Base
MSS	Maximum Segment Size
MTU	Maximum Tranmission Unit
NFS	Network File System

PAWS	Protect Against Wrapped Sequence Numbers
PPP	Point-to-Point Protocol
PCB	Process Control Block
POSIX	Portable Operating System Interface for uniX
RMP	Reliable Multicast Protocol
RMTP	Reliable Multicast Transport Protocol
RPC	Remote Procedure Call
RST	ReSeT
RTT	Round Trip Time
RTO	Retransmission Timeout
SACK	Selective Acknowledgment
SCTP	Stream Control Transmission Protocol
SCE	Single Connection Emulation
SFQ	Stochastic Fairness Queueing
SKB	Socket Buffer
SMP	Symmetric Multiprocessing
SMTP	Simple Mail Transport Protocol
SRM	Scalable Reliable Multicast
SYN	SYNchronize
TBF	Token Bucket Filter
TSO	TCP Segmentation Offload
TCP	Transport Control Protocol
UDP	User Datagram Protocol
URG	URGency
XNS	Xerox Network System
XTP	Xpress Transfer Protocol

LISTA DE FIGURAS

RESUMO

Multicast permite a transmissão $1 \rightarrow N$ de dados de maneira eficiente. Apesar de multicast no IP existir desde o início da década passada, poucos protocolos para transmissão confiável de dados foram realmente implementados. Não existe até hoje nenhum protocolo multicast confiável nativamente implementado no kernel GNU/Linux mainline. Este trabalho tem como objetivo descrever os passos realizados para implementação do protocolo TCP-XM em nível de kernel. Protocolos multicast confiável são inerentemente complexos. Neste trabalho apresenta-se um estudo prévio sobre o protocolo TCP-XM e sobre o subsistema de rede do kernel GNU/Linux. A implementação do protocolo TCP-XM é abordada tanto no contexto do kernel quanto no contexto de usuário, juntamente o ferramental utilizadao para seu desenvolvimento.

Palavras-chave: Protocolos de rede, multicast, TCP-XM, kernel GNU/Linux.

A Multicast Protocol Development and Evaluation on GNU/Linux Kernel

ABSTRACT

Multicast allows the efficient transmission of data from one machine to many. Although IP multicast exists since early 90s, few reliable multicast protocols have been actually implemented. Up to date, no reliable multicast protocol has been implemented in the mainline kernel of a GNU/Linux system. The main goal of this work is to describe the steps toward a kernel-level implementation of the TCP-XM protocol. Reliable multicast protocols are inherently complex. We first present a review on TCP-XM and the Linux network subsystem. The protocol implementation is addressed on both kernel and user levels, as well as the tools employed during implementation.

Keywords: network protocols, multicast, TCP-XM, GNU/Linux Kernel.

1 INTRODUÇÃO

O TCP, principal protocolo de transporte da internet, oferece uma forma de comunicação confiável, orientada à conexões [Braden 1989] e com entrega seqüencial de octetos entre um transmissor e um receptor [DARPA 1981]. Exemplos de aplicações que utilizam o TCP como protocolo de transporte são o SMTP [Postal 1982], utilizado para envio de e-mails e o FTP [Reynolds e Postel 1985], utilizado para transferência fim-a-fim de dados entre nodos da rede.

A transmissão simultânea dos mesmos dados para várias máquina utilizando o protocolo TCP é feita através da transmissão de múltiplas cópias destes dados, uma para cada máquina receptora. Tal método não é eficiente quando temos um grande volume de dados para transmitir. IP Multicast [Deering 1989] é o método de transmissão eficiente para transferências $1 \rightarrow N$ que não necessitam garantia na entrega dos pacotes.

Multicast confiável [Mankin 1998] é a forma de transmissão utilizada para garantir a entrega dos pacotes enviados de um transmissor para vários receptores de maneira eficiente. Existem na literatura diversos protocolos que implementam multicast confiável. Como exemplo pode-se citar o RMTP [Lin e Paul 1996], o SRM [Floyd et al. 1997], o M/TCP [Visoottiviseth et al. 2001] e o RMP [12].

O suporte a multicast não costuma ser popular nos ambientes atuais de produção. Apesar da grande quantidade de protocolos de multicast confiável existentes, não é possível utilizar tal protocolo quando não há conectividade IP multicast entre transmissor e receptores, pois os pacotes não chegariam até os mesmos.

Ao contrário da maioria dos protocolos multicast confiável da literatura, o TCP-XM [Jeacle et al. 2005] é um protocolo de multicast confiável *sender initiated*. Isto significa que o transmissor sinaliza o início da transmissão e garante a entrega dos pacotes [8]. O TCP-XM utiliza um método híbrido de transmissão multicast e unicast [Jeacle e Crowcroft 2005]. A transmissão multicast é utilizada para garantir o desempenho necessário, mas caso multicast não esteja disponível é possível enviar dados para múltiplos receptores através de múltiplas conexões unicast [Jeacle e Crowcroft 2004]. O TCP-XM foi desenvolvido para aplicações *push* (aplicações com transmissão unidirecional e $1 \rightarrow N$), para um pequeno número de receptores [Jeacle et al. 2005].

Atualmente a implementação do protocolo TCP-XM é feita através da modificação da pilha TCP/IP lwIP [Dunkels 2001], desenvolvida em nível de usuário [Jeacle et al. 2005]. Apesar das facilidades fornecidas por uma pilha TCP/IP em nível de usuário, uma sobrecarga é adicionada durante a transmissão de segmentos. Outro problema são as limitações impostas pela implementação do lwIP (como por exemplo a falta de janelas deslizantes [Jeacle et al. 2005]). Uma melhoria para aumento no desempenho do protocolo TCP-XM seria implementá-lo nativamente no kernel de sistema operacional [Jeacle 2005].

O kernel GNU/Linux [Pick 2006] é um kernel clone do sistema operacional Unix

[OpenGroup 2006] escrito por Linus Torvalds e mantido pela comunidade de software livre. O kernel GNU/Linux possui implementações em diversas plataformas, possui código fonte sob licença GPL e grande disseminação no mercado de servidores corporativos.

Uma implementação e avaliação do TCP-XM dentro de um kernel são de grande importância para quantificar o ganho real do protocolo TCP-XM sobre os demais protocolos de multicast confiável da mesma categoria e mesmo sobre o próprio protocolo TCP utilizando múltiplas conexões unicast.

Pelas razões citadas acima, este trabalho objetiva desenvolver uma versão do TCP-XM em nível de kernel, considerando o GNU/Linux como sistema alvo. Além disso, acredita-se que este trabalho ofereça uma implementação base do TCP-XM em kernel que, futuramente, possa ser melhorada e incorporada à árvore do kernel GNU/Linux, beneficiando um grande número de usuários.

O restante deste documento está organizado da seguinte maneira. O Capítulo 2 apresenta o referencial teórico, contendo uma revisão dos conceitos fundamentais necessários ao entendimento deste trabalho. O Capítulo 3 traz uma descrição sobre a arquitetura da implementação, enquanto que uma descrição sobre a implementação realizada e uma sugestão de experimentos para avaliação do protocolo TCP-XM são oferecidos nos Capítulos 3 e 5, respectivamente. O Capítulo 6 encerra o trabalho com conclusões e trabalhos futuros.

2 REFERENCIAL TEÓRICO

Este Capítulo revisa os principais conceitos relacionados à implementação de um protocolo no subsistema de rede do kernel GNU/Linux, tratando apenas dos elementos mais centrais ao trabalho. Não é objetivo deste Capítulo abordar temas abrangentes de natureza introdutória, como por exemplo os fundamentos do kernel GNU/Linux, funcionamento de protocolos multicast ou TCP. Maiores informações sobre o kernel GNU/Linux podem ser obtidas em [Pick 2006; Hemminger 2006; 4; Sarolahti e Kuznetsov 2002; Beck 2002]. Maiores informações sobre fundamentos de multicast confiável podem ser obtidas em [Lin e Paul 1996; Mankin 1998; Deering 1989; Comer 1998; Paul 1998]. Maiores informações sobre fundamentos do protocolo TCP podem ser obtidas em [Braden 1989; Stevens 1994; Comer 1998; 3; 6; 7].

Neste Capítulo, primeiramente apresenta-se na Seção 2.1 uma visão geral da arquitetura do subsistema de rede do kernel GNU/Linux, aonde será realizada a implementação do protocolo TCP-XM. O desenvolvimento de um protocolo confiável em nível de kernel é uma tarefa que exige o domínio de diversas ferramentas. A Seção 2.2 trata do ferramental empregado e ao permitir seu melhor entendimento contribui para que interessados possam dar continuidade a este trabalho. Por fim a Seção 2.3 descreve o funcionamento do protocolo TCP-XM apontando e resumindo as principais questões definidas em [Jeacle et al. 2005; Jeacle e Crowcroft 2004; Jeacle e Crowcroft 2005; Jeacle 2005]. O entendimento da arquitetura do subsistema de rede e do funcionamento do protocolo TCP-XM são fundamentais para o entendimento do restante do texto.

2.1 Subsistema de Rede do Kernel GNU/Linux

O kernel GNU/Linux é um kernel clone do sistema operacional Unix escrito por Linus Torvalds e mantido pela comunidade Open Source [Pick 2006]. É um kernel monolítico híbrido, pois apesar de possuir um design monolítico permite que drivers e outras facilidades possam ser carregadas e removidas da memória sob demanda.

A implementação do protocolo TCP-XM desenvolvida neste trabalho é realizada no subsistema de rede do kernel GNU/Linux. Nesta Seção apresenta-se um resumo sobre a história do subsistema de rede e as facilidades fornecidas pelo kernel GNU/Linux para comunicação entre nodos de rede. A comunicação entre camadas do subsistema de rede ocorre através de uma estrutura conhecida como `sk_buff`, que representa um segmento de rede no kernel GNU/Linux. Esta estrutura é descrita na Subseção 2.1.2.1. Ao final dessa Seção é apresentado um resumo sobre o ciclo de vida de um pacote de rede, desde sua criação até sua recepção. Maiores informações sobre o subsistema de rede do kernel GNU/Linux e projetos relacionados podem ser encontradas em [Hemminger 2006].

2.1.1 História

O voluntário que iniciou o desenvolvimento do subsistema de rede do kernel GNU/Linux foi *Ross Biro*. Ross produziu um conjunto incompleto de rotinas e um driver para placa de rede WD-8003. Este foi o suporte inicial à redes fornecido pelo kernel GNU/Linux. Ross foi o primeiro mantenedor do subsistema de rede deste sistema operacional.

Orest Zborowski reescreveu a interface de programação de sockets BSD para o kernel GNU/Linux, adicionando novas facilidades ao código de Ross Biro. Foi nesta época que Laurence Culhane iniciou o desenvolvimento dos primeiros drivers de interfaces de rede no kernel GNU/Linux.

Devido ao acúmulo de compromissos, Ross Biro entregou o cargo de mantenedor oficial do subsistema de rede à Fred van Kempen. Fred reorganizou o código de Ross criando a então denominada NET-2. Fred foi o responsável por grandes inovações no subsistema de rede do kernel GNU/Linux como a implementação do protocolo de rádio amador AX.25. Apesar dos esforços o subsistema de rede não era muito estável, sendo necessário dedicação integral ao projeto para produção de um código mais seguro e estável.

Alan Cox resolveu melhorar o código de Fred, tornando-o confiável e estável. Foi nesta época que Alan tornou-se o novo mantenedor do subsistema de rede do kernel GNU/Linux. Nesta época, Alan Cox e Donald Becker adicionaram diversos drivers de interfaces de rede no kernel GNU/Linux. O código-fonte produzido ficou conhecido como NET-2D.

Alan Cox continuou melhorando significativamente a estrutura do subsistema de rede, adicionando suporte a protocolos como o IPX. Foi nesta época que o protocolo PPP foi adicionado por Michael Callahan e Al Longyear.

Com a lançamento do kernel 2.0, todo o subsistema de rede foi reescrito por Alan Cox em conjunto com membros da comunidade OpenSource. O código produzido ficou conhecido como INET3. Desde então foram realizadas melhorias como a adição de novas facilidades, protocolos e *drivers*. Entretanto, o *core* do código continua o mesmo até os dias de hoje.

Após o lançamento do kernel 2.6, Alan Cox entregou o cargo de mantenedor para uma equipe de experientes desenvolvedores liderados por David S. Miller, que permanece como mantenedor até os dias de hoje.

2.1.2 Estruturas de Dados

Existem dezenas de estruturas de dados utilizadas na implementação das facilidades de rede encontradas no subsistema de rede do kernel GNU/Linux. Dentre elas, são explanadas as estruturas-chave para a implementação do subsistema de rede. Os Linux Socket Buffers são utilizados para comunicação entre as camadas da pilha de protocolos. A estrutura `Socket` representa uma conexão através da abstração BSD e a estrutura `Sock` representa uma conexão da família AF_INET.

A Figura 2.1 ilustra a ligação entre as principais estruturas de dados do subsistema de rede do kernel GNU/Linux:

As próximas subseções fornecem uma visão geral destas estruturas. Maiores informações podem ser obtidas diretamente nos comentários do código-fonte dos arquivos, que podem ser visualizados em [Menegotto 2006].

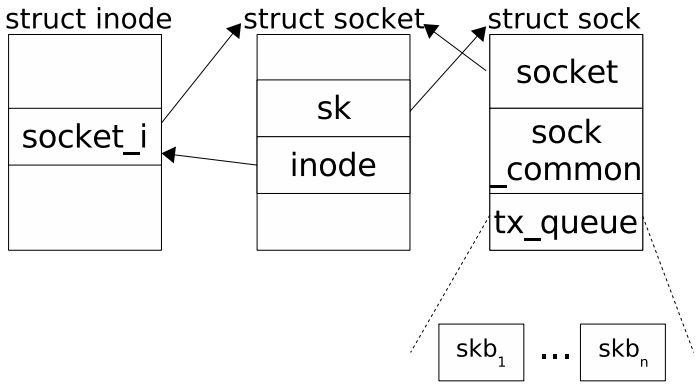

Figura 2.1: Relação entre principais estruturas de dados

2.1.2.1 Linux Socket Buffer

O Linux Socket Buffer (também conhecido como `sk_buff` ou `skb`) é uma estrutura de dados utilizada por todas as camadas de rede do kernel GNU/Linux. Sua função é facilitar, padronizar e otimizar o acesso necessário aos dados encapsulados pelas diversas camadas do subsistema de rede no kernel GNU/Linux. Um `skb` não representa diretamente um pacote, pois os dados de um `skb` podem ser fragmentados em um ou mais segmentos de rede dependendo da família de protocolos utilizada pela aplicação.

A maior parte da complexidade em escrever facilidades de rede no kernel GNU/Linux reside na correta manipulação dos `skb`s [Cox 1996]. Para facilitar esta tarefa existem quatro tipos distintos de facilidades fornecidas pela implementação dos `skb`s no *core* do subsistema de redes do kernel GNU/Linux [Welte 1996]. São elas:

- Funções para manipulação dos buffers (criação, cópia, destruição, etc...).
- Funções para manipulação de listas de `skb`s.
- Funções para manipulação dos dados contidos em `skb`s.
- Funções para identificação do tipo de `skb`s.

A implementação destas facilidades pode ser encontrada em `net/core/skbuff.c` e `include/linux/skbuff.h`. A estrutura atual de um `skb` é a seguinte:

- `next`: ponteiro para o próximo `skb` da lista.
- `prev`: ponteiro para o `skb` anterior da lista.
- `list`: ponteiro para lista encadeada em que o `skb` está inserido.
- `sk`: ponteiro para a estrutura sock a que o `skb` está associado.
- `tstamp`: timestamp da criação do `skb`.
- `dev`: interface de rede utilizada na transmissão do `skb`.
- `input_dev`: interface de rede utilizada na recepção do `skb`.
- `h`: ponteiro para o cabeçalho da camada de transporte.

- `nh`: ponteiro para o cabeçalho da camada de rede.
- `mac`: ponteiro para o cabeçalho da camada de enlace de dados.
- `dst`: ponteiro para estrutura que armazena informações sobre a rota de saída desse `skb`.
- `sp`: *security path* utilizado na implementação do XFRM. Maiores informações sobre este campo podem ser obtidas em `net/xfrm/xfrm.c` e [Kent e Atkinson 1998].
- `cb`: buffer de controle reservado para uso de qualquer camada. Variáveis privadas normalmente são colocadas aqui.
- `len`: tamanho em bytes do `skb`.
- `data_len`: tamanho em bytes dos dados encapsulados no `skb`.
- `mac_len`: tamanho em bytes do cabeçalho da camada de enlace de dados.
- `csum`: soma de verificação do pacote utilizado pelo protocolo de transporte.
- `local_df`: flag que permite a fragmentação local.
- `cloned`: flag que indica se cabeçalho do `skb` foi criado ou clonado.
- `nohdr`: tamanho em bytes da carga (*payload*) desse `skb`.
- `pkt_type`: flag que indica o tipo do pacote armazenado no `skb`.
- `ip_summed`: soma de verificação do protocolo IP.
- `priority`: identificador da prioridade do `skb` em decisões de escalonamento.
- `users`: contador de usuários do `skb`. Esta variável não é mais utilizada em versões recentes do kernel GNU/Linux.
- `protocol`: flag que indica o tipo de protocolo informado pelo driver da interface de rede para o `skb`.
- `truesize`: tamanho real do `skb` (cabeçalho e área de dados).
- `head`: ponteiro para o cabeçalho do `skb`.
- `data`: ponteiro para a região de dados do `skb`.
- `tail`: ponteiro para última seção do `skb`.
- `end`: ponteiro para o final do `skb`.
- `destructor`: ponteiro para função destrutora do `skb`.

Existem campos da estrutura `skb` definidos exclusivamente para facilidades do subsistema de rede. São exemplos de facilidades que possuem campos privados no `skb`:

- **Netfilter**: define os campos `nfmark`, `nfct`, `nfctinfo` e `nf_bridge`. Maiores informações sobre a utilização destes campos podem ser obtidas em `net/netfilter/netfilter.c` e [Russel e Welte 2002].

- **Traffic Shaper**: define os campos `tc_index` e `tc_verd`. Estes campos são amplamente utilizados pelos escalonadores de pacotes implementados pelo kernel GNU/Linux. Maiores informações podem ser obtidas em `net/sched` e [Bennett e Zhang 1997; Ng 1999; McKenney 1990].

A implementação do protocolo TCP-XM adiciona mais um campo que identifica o tipo do `skb` (multicast ou unicast). Esta flag indica o tipo do `skb` as diversas funções percorridas pelo `skb` no seu caminho de execução.

2.1.2.2 Socket

A estrutura `socket` forma a base da abstração de sockets BSD implementada no kernel GNU/Linux [Beck 2002]. A declaração da estrutura `socket` é feita no arquivo `include/net/net.h`. Os campos a seguir fazem parte da estrutura `socket`:

- `socket_state`: define o estado do socket. Os estados podem ser `SS_FREE` (em que o socket não está alocado), `SS_UNCONNECTED` (em que o socket não está conectado), `SS_CONNECTING` (em que o socket está em processo de conexão), `SS_CONNECTED` (em que o socket está conectado) e `SS_DISCONNECTING` (em que o socket está em processo de desconexão).

- `flags`: *bitmask* que identifica características do socket. As informações armazenadas neste bitmask são `SOCK_ASYNC_NOSPACE`, `SOCK_ASYNC_WAITDATA`, `SOCK_NOSPACE` e `SOCK_PASSCRED`.

- `ops`: ponteiro para a estrutura `proto_ops` do protocolo de transporte utilizado no socket. A estrutura `proto_ops` armazena ponteiros para os *callbacks* do protocolo conforme explicado na Seção 4.1.

- `fasync_list`: ponteiro para lista de processos do contexto de usuário que utilizam o socket. Esta lista serve para sinalização de eventos assíncronos ocorridos com o socket aos processos associados.

- `file`: ponteiro para o sistema de arquivos (*inode*) utilizado para armazenamento do socket.

- `sk`: ponteiro para a estrutura `sock` associada ao `socket`. Maiores informações sobre a estrutura `sock` podem ser obtidos na Subseção 2.1.2.4

- `wait`: fila de ponteiros para os processos do contexto de usuário que estão aguardando por eventos do `socket`.

- `type`: tipo do socket. Os tipos permitidos são `SOCK_STREAM` (utilizado em transmissões orientadas à conexão de streams de bytes), `SOCK_DGRAM` (utilizado em transmissões não-orientadas a conexão de datagramas), `SOCK_RAW` (utilizado em transmissões que necessitam que a aplicação construa os cabeçalhos dos segmentos), `SOCK_RDM` (utilizado para comunicação confiável de datagramas sem ordenação), `SOCK_SEQPACKET` (utilizado para transmissão confiável e seqüencial

de datagramas com um tamanho máximo fixo), SOCK_DCCP (utilizado exclusivamente pelo protocolo de transporte DCCP [2]) e SOCK_PACKET (utilizado para comunicação entre contexto de usuário e espaço do kernel).

2.1.2.3 Sock_common

A estrutura sock_common é uma representação mínima da abstração de sockets BSD para a camada de rede. Ela foi criada para compartilhar campos comuns aos diversos tipos de sockets existentes. A declaração da estrutura sock_common é realizada dentro do arquivo include/net/sock.h. Os campos a seguir fazem parte de sua estrutura:

- skc_family: família do endereço de rede. As famílias de endereço suportadas pelo kernel GNU/Linux são: AF_INET, AF_INET6, AF_UNIX, AF_LOCAL, AF_AX25, AF_APPLETALK, AF_NETROM, AF_BRIDGE, AF_ATMPVC, AF_X25, AF_ROSE, AF_DECNET, AF_NETBEUI, AF_SECURITY, AF_KEY, AF_NETLINK, AF_ROUTE, AF_PACKET, AF_ASH, AF_ECONET, AF_ATMSVC, AF_SNA, AF_IRDA, AF_PPOX, AF_WANPIPE, AF_LLC e AF_BLUETOOTH.
- skc_state: estado da conexão.
- skc_reuse: configuração do parâmetro SO_REUSEADDR (que permite a reutilização de endereços de sockets que estão no estado TCPXM_TIME_WAIT.
- skc_bound_dev_if: número da interface de saída associada a esta sessão.
- skc_node: ponteiro para o próximo nodo da tabela hash.
- skc_bind_node: ponteiro para o próximo nodo da hash de portas alocadas.
- skc_refcnt: número de processos do contexto de usuário associados ao socket.
- skc_hash: chave hash utilizado para pesquisa em tabelas hash.
- skc_prot: ponteiro para estrutura que armazena as rotinas de tratamento de eventos desta família de endereços de rede.

2.1.2.4 Sock

A estrutura sock é uma representação da estrutura socket com dados específicos da camada de rede. Esta é a estrutura utilizada pelas funções e procedimentos da implementação do subsistema de rede (incluindo o protocolo TCP-XM) para representar uma conexão. A partir da estrutura sock é possível manipular todas as demais estruturas referentes à sessão através de macros como tcpxm_sk e inet_sk. A declaração da estrutura sock é realizada dentro do arquivo include/net/sock.h. Os campos a seguir compõe a estrutura sock:

- __sk_common: ponteiro para estrutura sock_common do sock.
- sk_shutdown: bitmask de SEND_SHUTDOWN e/ou RCV_SHUTDOWN
- sk_userlocks: bitmask de SO_SNDBUF (que manipula o tamanho em bytes do buffer de transmissão) e SO_RCVBUF (que manipula o tamanho em bytes do buffer de recepção).

- `sk_lock`: variável de trava utilizada para sincronização entre processos.
- `sk_rcvbuf`: tamanho em bytes do buffer de recepção.
- `sk_sleep`: fila de ponteiros para processos no contexto de usuário que estão no estado *sleep* aguardando por eventos do `sock`.
- `sk_dst_cache`: ponteiro para estrutura `dst_entry`, que armazena informações sobre a rota de saída do `sock`.
- `sk_dst_lock`: lock utilizado para acesso à estrutura `dst_entry`.
- `sk_policy`: política de transmissão utilizada para implementação da API XFRM.
- `sk_rmem_alloc`: tamanho em bytes da memória alocada para o buffer de recepção do `sock`.
- `sk_receive_queue`: fila que representa o buffer de recepção do `sock`.
- `sk_wmem_alloc`: tamanho em bytes da memória alocada para o buffer de transmissão do `sock`.
- `sk_write_queue`: fila que representa o buffer de transmissão do `sock`.
- `sk_omem_alloc`: tamanho em bytes da memória alocado para os buffers de recepção (`prequeue`, `out_of_order_queue`, `sk_backlog` e `sk_error_queue`) do `sock`.
- `sk_wmem_queued`: tamanho mínimo em bytes do buffer de transmissão do `sock`.
- `sk_forward_alloc`: tamanho em bytes da memória alocado para o roteamento de segmentos.
- `sk_allocation`: modo de alocação de memória utilizado. Os modos possíveis são `GFP_KERNEL` (a alocação é feita conforme disponibilidade do kernel) e `GFP_ATOMIC` (a alocação é feita da forma atômica).
- `sk_sndbuf`: tamanho em bytes do buffer de transmissão.
- `sk_flags`: bitmask de `SO_LINGER` (que habilita o envio em background de segmentos não transmitidos antes da finalização do `sock`), `SO_BROADCAST` (que permite o envio de datagramas broadcast), `SO_KEEPALIVE` (habilita o envio de mensagens *keep-alive*) e `SO_OOBINLINE` (que coloca segmentos destinados à `out_of_order_queue` diretamente no buffer de recepção).
- `sk_no_check`: bitmask de `SO_NO_CHECK` (que habilita a verificação de integridade dos pacotes do `sock`).
- `sk_route_caps`: configuração das facilidades de roteamento presentes na interface do `sock`.
- `sk_lingertime`: configuração do tempo máximo de espera antes da destruição do `sock` caso a opção `SO_LINGER` esteja habilitada.

- sk_backlog: fila em que são armazenados segmentos de uma sessão antes de seu estabelecimento. Como o acesso ao buffer sk_backlog é feito somente com o uso de spinlocks (para diminuir a latência de acesso) a implementação da fila sk_backlog foi realizada de forma diferenciada.
- sk_callback_lock: lock utilizado pelos callbacks sk_write_pending, sk_state_change, sk_data_ready, sk_write_space, sk_error_report, sk_backlog_rcv e sk_destruct do sock.
- sk_error_queue: fila que representa o buffer de recepção de pacotes corrompidos.
- sk_prot_creator: ponteiro para função de criação do sock.
- sk_err: último erro ocorrido com o sock.
- sk_err_soft: último erro que não causou uma falha persistente no sock.
- sk_ack_backlog: número de sessões que estão na fila sk_backlog esperando pelo estabelecimetno da sessão.
- sk_max_ack_backlog: número máximo de conexões que podem ficar na fila sk_backlog esperando pelo estabelecimento da sessão.
- sk_priority: configuração do parâmetro SO_PRIORITY (que configura uma prioridade padrão a todos segmentos originados do sock).
- sk_type: tipo do socket associado a esta estrutura sock (SOCK_STREAM, SOCK_DGRAM, SOCK_RAW, SOCK_RDM, SOCK_SEQPACKET, SOCK_DCCP ou SOCK_PACKET).
- sk_protocol: número do protocolo associado ao sock.
- sk_peercred: configuração do parâmetro SO_PEERCRED (que retorna as credenciais do processo remoto associado ao sock).
- sk_rcvlowat: configuração do parâmetro SO_RCVLOWAT (que configura o número mínimo de bytes armazenados no buffer de recepção antes de envia-los ao contexto do usuário).
- sk_rcvtimeo: configuração do parâmetro SO_RCVTIMEO (que configura o timeout de recepção).
- sk_sndtimeo: configuração do parâmetro SO_SNDTIMEO (que configura o timeout de transmissão).
- sk_filter: campo privado utilizado pelo filtro de pacotes (netfilter).
- sk_protinfo: campo privado da camada de rede utilizando quando o sock não está sendo armazenado no *slab cache*.
- sk_timer: temporizador do sock.
- sk_stamp: timestamp do último segmento recebido nesta estrutura sock.

- `sk_socket`: ponteiro para estrutura `socket` associada ao `sock`.
- `sk_user_data`: campo privado da camada RPC.
- `sk_sndmsg_page`: ponteiro para página da cache utilizada na função de transmissão de segmentos do `sock`.
- `sk_sndmsg_off`: offset da página de cache da função de transmissão de segmentos apontada por `sk_sendmsg_page`.
- `sk_send_head`: ponteiro para o primeiro `sk_buff` da fila de transmissão.
- `sk_security`: campo privado utilizado pelos mecanismos de segurança do kernel (SElinux [NSA 2006]).
- `sk_write_pending`: callback que indica a existência de dados para transmissão na fila `sk_write_queue`.
- `sk_state_change`: callback que indica mudança no estado do `sock`.
- `sk_data_ready`: callback que indica a existência de dados recebidos e ainda não processados.
- `sk_write_space`: callback que indica a existência de espaço disponível no buffer de transmissão.
- `sk_error_report`: callback que indica a ocorrência de erros na sessão.
- `sk_backlog_rcv`: callback que realiza o processamento de pacotes da fila `sk_backlog`.
- `sk_destruct`: callback utilizado para destruição do `sock` (quando $refcnt == 0$).

2.1.3 Arquitetura do Subsistema de Rede

A arquitetura do subsistema de rede do kernel GNU/Linux é orientada à interrupções de forma que o processamento dos pacotes recebidos é priorizado pelo sistema operacional. Esta estratégia produz um maior *throughput* de rede mas pode levar ao congelamento do sistema operacional em situações de tráfego acima do suportado [Druschel e Banga 1996].

A Figura 2.2 ilustra a arquitetura do subsistema de rede do kernel GNU/Linux.

Durante a inicialização do sistema operacional, o subsistema de rede é inicializado, juntamente com suas facilidades. Após a inicialização, sua única tarefa é transmitir, receber e encaminhar segmentos da rede para o contexto de usuário. Estas tarefas são descritas nas Subseções 2.1.3.1, 2.1.3.2 e 2.1.3.3.

2.1.3.1 Inicialização do Subsistema de Rede

Durante a inicialização do kernel GNU/Linux, as interfaces de rede são ativadas (caso habilitadas). Após, são inicializadas as estruturas de dados utilizadas na implementação da abstração de sockets BSD e as estruturas de dados utilizadas para manipulação de `sk_buffs`. Somente durante a execução da função `do_initcalls` ocorre a inicialização dos protocolos habilitados durante a compilação do kernel. O envio e recepção de pacotes começa somente após a inicialização do firewall e de todos os protocolos de rede habilitados.

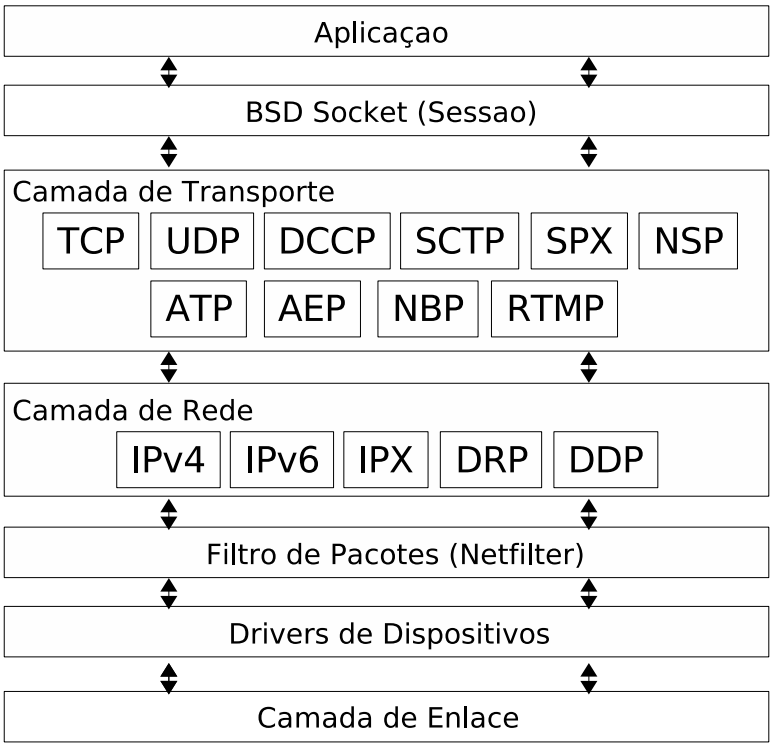

Figura 2.2: Arquitetura do Subsistema de Rede do kernel GNU/Linux

2.1.3.2 Transmissão de Pacotes

A transmissão de dados no kernel GNU/Linux inicia no contexto do usuário quando uma chamada `send`, `sendmsg` ou `sendto` é executada. Esta chamada é interpretada pela função `sys_socketcall`, que executa a função correspondente `sys_send`, `sys_sendmsg` ou `sys_sendto`. Estas funções passam parâmetros para a função `sock_sendmsg`, que executa a função `__sock_sendmsg` e copia os parâmetros do contexto de usuário para o kernel, invocando as funções de transmissão de pacotes do protocolo utilizado pela aplicação.

No kernel, a função `.sendmsg` da estrutura `proto_ops` do protocolo inicia o ciclo de vida do pacote, até a sua transmissão pela interface de rede. Primeiro o dado é encapsulado pelo protocolo de transporte (TCP, UDP, DCCP ou SCTP) dentro de um `sk_buff` e enviado para a camada de rede.

Na camada de rede o cabeçalho de rede é preenchido e o pacote é endereçado. Protocolos como o IP ou o IPX são exemplos de protocolos utilizados para esta tarefa. Após o endereçamento o pacote é encaminhado ao driver da interface de rede, que encarrega-se de transmitir o segmento assim que possível.

2.1.3.3 Recepção de Pacotes

A recepção de pacotes percorre o caminho inverso da transmissão no subsistema de rede. Quando um pacote chega ele é repassado pelo driver da interface de rede para a camada de rede, que encarrega-se da validação do segmento recebido. Após a validação, a camada de rede localiza na tabela hash de protocolos o protocolo associado ao segmento recebido. A camada de rede entrega então o `sk_buff` recebido para a camada de transporte, que após fazer o processamento necessário envia os dados para o contexto de

usuário, para processamento pela aplicação.

2.1.4 Código-Fonte do Subsistema de Rede

O kernel é um grande projeto de desenvolvimento que mistura linguagens de programação como C e Assembly. O tamanho atual do kernel GNU/Linux mainline versão 2.6.15.4 comprimido através do algoritmo bzip2 é de 38 MB.

Uma explanação sobre as facilidades encontradas nos diretórios do subsistema de rede pode fornecer uma visão geral da arquitetura de implementação utilizada no kernel GNU/Linux. O código-fonte do subsistema de rede do kernel GNU/Linux está estruturado da seguinte forma:

- **802**: família de protocolos IEEE 802.3, que definem um mecanismo de comunicação entre o meio físico e a camada de enlace em redes Ethernet. Maiores informações sobre o protocolo IEEE 802.3 podem ser obtidas em [Law 2006].

- **8021q**: protocolo 802.1Q, que permite a implementação de VLANs gerenciadas pelo subsistema de rede do kernel GNU/Linux. Maiores informações sobre o protocolo IEEE 802.1Q podem ser obtidas em [Greear 2005].

- **appletalk**: appletalk é um conjunto de protocolos desenvolvido pela Apple no final dos anos 80 para compartilhamento de recursos de um computador Apple. Maiores informações sobre este conjunto de protocolos podem ser encontradas em [11].

- **atm**: suporte à redes ATM no kernel GNU/Linux. Maiores informações sobre redes ATM podem ser obtidas em [Comer 1998].

- **ax25**: protocolo de comunicação de rádio-amador AX.25. Maiores informações sobre o protocolo AX.25 podem ser obtidas em [Tranter 2001].

- **bluetooth**: comunicação com dispositivos bluetooth no kernel GNU/Linux. Maiores informações sobre esta implementação podem ser encontradas em [Bossek 2006].

- **bridge**: serviço de bridge no kernel GNU/Linux. Maiores informações sobre *bridging* em redes Ethernet no kernel GNU/Linux podem ser obtidas em [Hemminger 2006]

- **core**: cerne do subsistema de rede do kernel GNU/Linux. Todas as funções de rede que são independentes de protocolo e/ou arquitetura encontram-se neste subdiretório. Exemplos de rotinas são: funções para manipulação de interfaces, funções para manipulação de `sk_buffs`, funções para configuração de velocidade de transmissão e funções para manipulação de datagramas e streams.

- **decnet**: protocolos da pilha DECNET, uma pilha de protocolos similar ao TCP/IP utilizada em computadores fabricados pela Digital (conhecida atualmente como Compaq) nos anos 80. Maiores informações sobre redes DECNET podem ser obtidas em [Cisco 2002].

- **dccp**: protocolo DCCP, que utiliza datagramas associados à técnicas de controle de congestionamento para melhorar o *throughput* da rede. Maiores informações sobre o protocolo DCCP podem ser obtidas em [2].

- **econet**: protocolo ECONET, utilizado por computadores Accorn para compartilhamento de recursos (disco e impressoras). Maiores informações sobre o protocolo ECONET podem ser obtidas em [Murray 1999].

- **ethernet**: rotinas de manipulação de frames Ethernet no kernel GNU/Linux. Maiores informações sobre frames Ethernet podem ser obtidas em [Cisco 2005].

- **ieee80211**: rotinas de manipulação de frames Ethernet 802.11 (utilizados em redes wireless). Maiores informações sobre frames Ethernet 802.11 podem ser obtidas em [Ketrenos 2006]

- **ipv4**: protocolos que compõe a pilha TCP/IP do kernel GNU/Linux na versão 4 do protocolo IP. São eles: IPv4, IGMPv3, UDP e TCP. É neste diretório que é implementado o protocolo TCP-XM. Maiores informações sobre os protocolos da pilha TCP/IP podem ser encontrados em [Braden 1989; Murai 2006; DARPA 1981; Bieringer 2006]

- **ipv6**: protocolo IPv6 no kernel GNU/Linux. Facilidades como suporte à multicast e anycast em redes IPv6 também são implementadas neste diretório. Maiores detalhes sobre o protocolo IPv6 podem ser obtidos em [Bieringer 2006; Murai 2006]

- **ipx**: IPX é um protocolo da camada de rede do modelo OSI desenvolvido pela Novell para comunicação em redes Netware. Maiores informações sobre o protocolo IPX podem ser obtidas em [Novell 1989].

- **irda**: suporte à comunicação com dispositivos IRDA no kernel GNU/Linux. Dispositivos IRDA utilizam raios infra-vermelhos para comunicação e são geralmente encontrados em dispositivos móveis como celulares, PALMs e notebooks. Maiores informações sobre dispositivos IR podem ser encontradas em [Heuser 2003].

- **key**: protocolo PF_KEY (Key Management API), utilizado por aplicações que desejam comunicar-se com o mecanismo de gerenciamento de chaves do sistema operacional. Maiores informações sobre o protocolo PF_KEY podem ser obtidas em [5].

- **lapb**: protocolo LAPB, utilizado em redes AX.25 para comunicação entre rádios-amadores. Maiores informações sobre o protocolo LAPB podem ser obtidas em [Tranter 2001].

- **llc**: protocolo LLC, utilizado para manipulação de frames Ethernet 802.2. Frames Ethernet 802.2 são utilizados por diversos protocolos adjacentes como protocolo da camada de enlace. Os protocolos IPX, AppleTalk e NETBEUI são exemplos de protocolos que utilizam Frames Ethernet 802.2. Maiores informações sobre o protocolo LLC podem ser obtidas em [Fairhust 2006; Cisco 2005]

- **netfilter**: firewall Netfilter, escrito por Rusty Russel e mantido pela comunidade de software livre. Maiores informações sobre o Netfilter podem ser obtidas em [Russel e Welte 2002]

- **netlink**: protocolo NETLINK, desenvolvido por Alan Cox para comunicação entre o contexto de usuário e o kernel. Maiores informações sobre o protocolo Netlink podem ser obtidas em [Dhandapani 1999]

- **netrom**: protocolo de comunicação de rádio-amador NETROM. Maiores informações sobre o protocolo NETROM podem ser obtidas em [Tranter 2001]

- **packet**: abstração *Packet Socket*, utilizada para envio e recepção de pacotes encapsulados diretamente pela aplicação através de API de sockets BSD.

- **rose**: protocolo de comunicação de rádio-amador ROSE. Maiores detalhes sobre o protocolo ROSE podem ser obtidos em [Tranter 2001].

- **rxrpc**: protocolo Rx Protocol RPC (RxRPC), utilizado para comunicação entre nodos na implementação do sistema de arquivos distribuído AFS [Tanembaum 1994] no kernel GNU/Linux. O RxRPC foi desenvolvido pela equipe mantenedora do AFS dentro do kernel [Hutzelman 2006]. Maiores informações sobre este protocolo podem ser obtidas em [Zeldovich 2002].

- **sched**: escalonadores de pacotes de rede. Exemplos destes escalonadores são: TBF, SFQ e HFSC. A escolha do escalonador de pacotes adequado pode aumentar significativamente o desempenho da comunicação entre nodos de rede que utilizam o kernel GNU/Linux. Maiores informações sobre estes algoritmos podem ser obtidas em [Bennett e Zhang 1997; Ng 1999; McKenney 1990].

- **sctp**: protocolo SCTP, um protocolo de transporte confiável muito similar ao TCP mas que possui suporte para comunicação *multi-homed*, na qual os sockets do contexto de usuário (da aplicação) podem ser alcançados através de mais de um endereço IP. Maiores informações sobre o protocolo SCTP podem ser obtidas em [IBM 2004; Stewart et al. 2000]

- **sunrpc**: protocolo Sun RPC (também conhecido como ONC RPC), utilizado para execução de chamadas remotas por serviços do mundo Unix como o NFS. Maiores informações sobre o protocolo SunRPC podem ser obtidas em [Srinivasam 1995].

- **unix**: abstração *Unix Domain Sockets* utilizada para comunicação inter-processos em sistemas POSIX. Este tipo de socket é conhecido também como POSIX Local IPC Sockets e é utilizado em diversas aplicações do mundo Unix como por exemplo o X. Maiores detalhes sobre o Unix Domain Sockets podem ser obtidas em [Leffler et al. 1993].

- **wanrouter**: WANPIPE-S é um conjunto de protocolos utilizado para comunicação WAN/ADSL que pode atingir picos de 8 Mbps. O WANPIPE-S suporta uma vasta gama de tecnologias (ATM, ADSL, Frame-Relay, X25, etc...) e é utilizado em aplicações que utilizam comunicação via satélite e ADSL. Maiores informações sobre o WANPIPE-S podem ser obtidas em [Corbic 2006].

- **x25**: suporte à rede X25 (PLP e LAPB). Maiores informações sobre redes X25 podem ser obtidas em [Tranter 2001].

- **xfrm**: biblioteca XFRM, um framework de desenvolvimento utilizado na implementação do IPSEC no kernel GNU/Linux. Maiores informações sobre o framework XFRM podem ser obtidas em [Kent e Atkinson 1998].

2.2 Ferramental Utilizado

O desenvolvimento de novas facilidades no kernel de um sistema operacional requer um ambiente isolado e protegido para testes. Máquinas virtuais são programas que podem ser utilizadas para atingir este isolamento em ambientes computacionais. É necessário também utilizar uma ferramenta para análise dos segmentos TCP-XM que trafegam pela rede. Ferramentas conhecidas como *Analisadores de Pacotes* são utilizadas para este fim. Devido ao tamanho do código fonte do kernel GNU/Linux é necessário também conhecer uma ferramenta para o gerenciamento da árvore de desenvolvimento utilizada. Estas ferramentas são conhecidas como *Gerenciadores de Código-Fonte*.

2.2.1 Máquina Virtual

A configuração de um ambiente isolado e protegido para realização de testes durante o desenvolvimento de novas facilidades no kernel de um sistema operacional pode ser obtida através do uso de máquinas virtuais.

Muita pesquisa tem sido feita para melhoria das técnicas de virtualização empregadas nos últimos anos. Dentre os trabalhos mais recentes, podem ser citados os avanços obtidos com a técnica de para-virtualização através das ferramentas Xen [Dragovic et al. 2003] e Qemu [Bellard 2006; Bellard 2005].

Para o presente trabalho, foram realizados testes com as máquinas virtuais User-Mode-Linux [Dike 2006], VMWare [VMWare 2006] e Qemu [Bellard 2006; Bellard 2005]. O principal âmbito dos testes foi verificar qual máquina virtual permitiria a inicialização automática de uma VLAN de instâncias de máquinas virtuais com suporte a IP multicast através de comandos de shell do sistema operacional Linux. Foi avaliada também a capacidade da máquina virtual de trabalhar diretamente com o binário gerado durante a compilação do kernel GNU/Linux.

A máquina virtual Qemu permite uma inicialização totalmente automatizada de uma VLAN de instâncias através de comandos de shell no sistema operacional Linux, o que facilita o processo de compilação e o processo de testes do kernel GNU/Linux durante o desenvolvimento do protocolo TCP-XM. Por essas razões, a máquina virtual Qemu foi escolhida para o desenvolvimento do protocolo TCP-XM.

2.2.2 Analisador de Pacotes

A visualização do conteúdo dos pacotes que trafegam por uma rede Ethernet pode ser realizada quando a interface de rede utilizada é configurada para um modo de operação conhecido como *modo promíscuo* [Comer 1998]. Ferramentas conhecidas como *Analisadores de Pacotes* ou *sniffers* são utilizadas para este fim.

A atividade de depuração realizada durante o desenvolvimento do protocolo TCP-XM exigiu a implementação do protocolo TCP-XM num sniffer. Foram realizados testes com o Ethereal [Ethereal 2006] e o TCPDump [Lenz 2006]. O principal âmbito dos testes foi verificar qual sniffer possuia a infra-estrutura mais amigável para inclusão de novos protocolos e qual executaria de forma mais eficiente em instâncias de máquinas virtuais.

O sniffer TCPDump possui um código fonte bem estruturado. Não existem plugins para inserção de protocolos como no Ethereal, mas a facilidade de leitura e a estruturação do código do TCPDump facilitaram a inclusão do protocolo TCP-XM. Além disso, por possuir exigências mínimas de recursos do hardware, é uma ferramenta que executa de maneira transparente em máquinas virtuais. Tal justificou a escolha do TCPDump para o desenvolvimento do protocolo TCP-XM.

2.2.3 Gerenciador de Código-Fonte

O código-fonte do kernel GNU/Linux descompactado possui tamanho de 246 MB (na versão 2.6.15.4). O gerenciamento de um código-fonte deste porte pode tornar-se uma péssima experiência para o programador caso a ferramenta certa não for empregada.

A ferramenta utilizada pelos principais desenvolvedores do kernel GNU/Linux chama-se Git [Torvalds e Hamano 2006]. Ela foi desenvolvida por Linus Torvalds e é através dela que as principais árvores de desenvolvimento são mantidas.

A facilidade no gerenciamento e no histórico das alterações realizadas aliadas as facilidades para atualização da árvore de desenvolvimento justificaram a opção pelo Git como ferramenta de gerenciamento do código-fonte.

2.3 Protocolo TCP-XM

Multicast confiável é a categoria de protocolos que consegue transmitir dados de um transmissor para múltiplos receptores ($1 \rightarrow N$) de maneira confiável, ou seja, o transmissor possui garantia de que os segmentos são recebidos pelo receptor (assumindo que o remetente não falhe e que não ocorra particionamento de rede entre o remetente e os receptores).

O protocolo TCP-XM foi desenvolvido por Karl Jeacle [Jeacle 2005]. O TCP-XM é um protocolo de multicast confiável que utiliza IP multicast nativo quando presente na rede e uma emulação de multicast sobre unicast caso contrário.

Além de uma visão geral sobre a área de multicast confiável e sobre o funcionamento do protocolo, nesta Seção estão descritas as premissas básicas para utilização do protocolo e os modos de transmissão suportados. A Subseção 2.3.5 descreve a implementação do protocolo TCP-XM realizada em [Jeacle 2005] e discorre sobre os problemas da metodologia utilizada.

2.3.1 Multicast Confiável

Multicast confiável tem provado ser um problema de difícil solução, com muita pesquisa sobre o assunto na última década. O resultado destas pesquisas foi uma pletora de protocolos criados para resolver pequenos problemas dentro da área. É possível categorizar os protocolos de multicast confiável da seguinte forma [Paul 1998]:

- **protocolos de entrega seqüencial**: dividem o arquivo em segmentos identificados de tamanho fixo. A transmissão de pacotes é relizada obedecendo a seqüência dos identificadores e a confirmação de recepção de cada pacote é essencial para garantir a confiabilidade da transmissão. O protocolo RMTP [Lin e Paul 1996] e o protocolo MFTP [Miller et al. 1997] são exemplos de protocolos que implementam multicast confiável utilizando entrega seqüencial de pacotes.

- **protocolos que emulam multicast sobre unicast**: protocolos que implementam multicast confiável através de técnicas de abstração sobre unicast. O controle de erros e o controle do fluxo são implementados no transmissor [Paul 1998]. O protocolo XTP [1] e o protocolo SCE [Talpade e Ammar 1995] são exemplos de protocolos que realizam esta abstração sobre o protocolo TCP. O protocolo TCP-XM pertence à esta classe de protocolos.

- **protocolos baseados em árvores**: protocolos que implementam multicast confiável através de uma política de *dividir para conquistar* [Paul 1998]. Os nodos receptores

são agrupados numa estrutura de árvore de forma que o controle de erro e as retransmissões são realizadas dos nodos superiores na hierarquia da árvore (nodos pai) para os nodos inferiores (nodos folha). O protocolo M/TCP [Visoottiviseth 2003] e o protocolo LORAX [Levine e Garcia-Luna-Aceves 1996] são exemplos de protocolos que implementam multicast confiável através de uma estrutura de árvores.

- **protocolos de comunicação em grupos**: protocolos que fornecem semânticas de ordenamento e transmissão diferenciados. Um exemplo de técnica utilizada para implementar este tipo de protocolo é dividir a rede em vários transmissores que serão responsáveis por grupos de receptores (em uma estrutura não hierárquica). O protocolo SRM [Floyd et al. 1997] e o protocolo RBP [Chang e Maxemchuk 1984] são exemplos de protocolos que implementam multicast confiável através da comunicação em grupos.

Em [Jeacle 2005], foram realizados experimentos comparativos com protocolos que implementam multicast sobre unicast através de modificações no protocolo TCP. No presente trabalho, não foram realizadas comparações com todos os protocolos testados em [Jeacle 2005] por uma questão de escopo, posto que esta monografia enfoca principalmente a implementação realizada.

Existem trabalhos similares que extendem o protocolo TCP para transmissão multicast. Dentre os principais protocolos que emulam multicast sobre unicast pode-se citar:

- **M/TCP**: extensão do protocolo TCP que introduz suporte à transmissão multicast (através da emulação sobre unicast). O M/TCP é um protocolo *sender-initiated* projetado para pequenos grupos multicast (SIM e XCAST) que insere o endereço de todos os receptores no cabeçalho do pacote para emular multicast sobre unicast. A implementação é realizada utilizando uma árvore com nodos intermediários conhecidos como SAs (Sender's Agent) que controlam a confirmação dos segmentos recebidos e a retransmissão dos segmentos perdidos. Maiores informações sobre o protocolo M/TCP podem ser obtidas em [Visoottiviseth 2003].

- **SCE**: *Single-Connection Emulation* introduz uma camada de abstração entre um protocolo confiável da camada de transporte (ex. TCP) e a camada de rede. A transmissão multicast é realizada utilizando IP multicast. A confirmação da recepção de segmentos é realizada através da interceptação dos múltiplos ACKs recebidos (um por receptor) e repasse à camada de transporte de um único ACK, confirmando o segmento de maneira unicast para a camada de transporte. Um problema visível deste protocolo é a implosão de ACKs [Visoottiviseth 2003]. Maiores informações sobre o protocolo SCE podem ser obtidas em [Talpade e Ammar 1995].

- **TCP-SMO**: protocolo *receiver-initiated* muito similar à arquitetura do TCP-XM que utiliza um mecanismo de SSM (em que cada receptor inscreve-se no canal multicast desejado). Um transmissor possui um canal multicast que agrega *n* conexões unicast (uma para cada receptor). O canal multicast armazena variáveis de controle de congestionamento, RTT e números de seqüência. Estas variáveis são sincronizadas com as conexões unicast mantendo a sincronização entre todas as conexões unicast. Maiores informações sobre o protocolo TCP-SMO podem ser obtidas em [Liang e Cheriton 2002].

- **TCP-M**: modificação do protocolo TCP que inclui suporte a multicast confiável. O TCP-M introduz uma camada de abstração entre os protocolos TCP e IP para realizar a emulação de multicast sobre unicast. A confirmação dos segmentos é realizada por nodos intermediários (normalmente roteadores) para minimizar o problema de implosão de ACKs. Existe uma implementação do protocolo TCP-M no kernel NETBSD [Mysore e Varghese 2006]. Maiores informações sobre o protocolo TCP-M podem ser obtidas em [Mysore e Varghese 2006; Ghosh e Varghese].

Nenhum dos protocolos acima citados combina IP multicast nativo e a emulação de multicast sobre unicast para transmissão. Este é o principal diferencial do protocolo TCP-XM em relação aos demais protocolos de sua categoria. [Jeacle 2005]

2.3.2 Premissas

As premissas a seguir devem ser consideradas para compreender o funcionamento do protocolo TCP-XM [Jeacle e Crowcroft 2005]:

- protocolo foi projetado para *Push-applications*. Push Applications são categorizadas como aplicações multicast *one-to-many latency-unconstrained*, ou seja, transmissão $1 \rightarrow N$ de uma quantidade massiva de dados. [Jeacle e Crowcroft 2005].

- protocolo é *Sender-Initiated* [Jeacle e Crowcroft 2005], ou seja, o transmissor é quem garante a recepção dos pacotes transmitidos [Paul 1998].

- transmissor deve conhecer previamente o endereço unicast de cada receptor para que seja possível emular IP multicast sobre unicast caso o suporte a IP multicast nativo não estiver presente na rede.

2.3.3 Funcionamento

Do ponto de vista do transmissor, uma sessão TCP-XM envolve as tarefas a seguir [Jeacle et al. 2005]:

- **escolha do Grupo Multicast**: escolha que pode ser feita pelo transmissor (através da escolha randômica de um grupo multicast vago) ou por consenso entre os receptores.

- **criação das PCBs**: uma PCB é criada para cada receptor. PCB é uma estrutura de dados interna do sistema operacional que armazena informações em determinado subsistema do kernel. No caso do subsistema de rede as PCBs armazenam informações sobre uma sessão de transmissão de dados.

- **estabelecimento da sessão**: ocorre um *3-way handshake* com cada receptor para sincronização dos números de seqüência. O campo de opções do segmento SYN contém o grupo multicast escolhido pelo transmissor. Os dados que serão transmitidos são replicados em todas filas de transmissão localizadas dentro das PCBs.

- **sincronização das janelas deslizantes**: os números de seqüência unicast e multicast devem ser sincronizados utilizando a fórmula $min(cwnd)\&min(snd\_wnd)$. Existe uma janela de transmissão por receptor e o transmissor somente envia dados que estão na interseção de todas as janelas.

- **transmissão de dados**: durante a primeira transmissão os pacotes são enviados de forma unicast e multicast simultaneamente. Os pacotes multicast possuem tipo de protocolo TCP-XM mas seu destino é o endereço multicast escolhido. A transmissão unicast é cancelada assim que a transmissão multicast é confirmada.

- **controle de erro**: a transmissão simultânea pode ocasionar a duplicação indesejada de segmentos de confirmação (ACK). Mensagens de confirmação duplicadas recebidas via unicast provavelmente tratam-se de retransmissões e portanto devem ser processadas normalmente. Mensagens de confirmação duplicadas confirmando a recepção unicast e multicast são provavelmente mensagens que perderam o ordenamento ou que trafegam em trechos de rede congestionados, o que pode ocasionar um atraso na entrega e a conseqüente duplicação das confirmações [Jeacle 2005]. A política utilizada pelo TCP-XM para solucionar esta situação é [Jeacle e Crowcroft 2005]: Caso uma mensagem de confirmação duplicada for recebida via multicast, ignore-a; Caso uma mensagem de confirmação duplicada for recebida via unicast, primeiro verifique se a mensagem original recebida via multicast já foi confirmada. Caso positivo, descarte-a. Caso contrário processe a mensagem.

- **escolha da tecnologia de transmissão**: IP Multicast é a tecnologia de transmissão utilizada durante toda a sessão caso detectado suporte na rede. Para realizar a detecção do suporte à IP multicast são coletadas informações pelo receptor sobre os últimos *n* (128 por default) pacotes recebidos via multicast. Esta informação é contabilizada e enviada ao transmissor dentro do campo de opções dos segmentos de confirmação de recepção (ACKs) [Jeacle e Crowcroft 2005]. Através desta informação o transmissor reconhece os receptores que suportam IP multicast.

- **retransmissão de pacotes**: no caso de perda de pacotes as retransmissões realizadas são feitas através de transmissão unicast.

- **finalização da sessão**: A conexão é fechada através do envio de um segmento FIN a todos os receptores, como numa conexão TCP comum.

Do ponto de vista do receptor, uma sessão TCP-XM envolve as tarefas a seguir:

- conectar-se ao endereço multicast enviado no segmento SYN (no caso de detecção do suporte à IP multicast) através do protocolo IGMP.

- aceitar dados tanto da conexão unicast quanto da conexão multicast estabelecidas com o transmissor.

- contabilizar o percentual de segmentos recebidos via multicast e enviar esta informação ao transmissor.

2.3.4 Modos de Transmissão

O modo de transmissão define como o protocolo transmite dados aos receptores e como o protocolo comporta-se em relação à detecção da tecnologia de transmissão. Existem cinco modos de transmissão possíveis numa sessão TCP-XM [Jeacle et al. 2005]:

- `TX_UNICAST_ONLY`: o transmissor fez uma requisição explícita para uso de unicast em todas as transmissões.

- `TX_UNICAST_GROUP`: dados são transmitidos via unicast e IP multicast simultaneamente para membros do grupo até que uma confirmação da recepção de pacotes transmitidos via IP multicast seja recebida pelo transmissor. Este é o modo padrão do protocolo.

- `TX_MULTICAST_ONLY`: o transmissor fez uma requisição explícita para uso de IP multicast em todas as transmissões. Esta opção pode ser utilizada em redes aonde existe prévio conhecimento de suporte à IP multicast.

- `TX_UNICAST_GROUP_ONLY`: utilizado quando não existe suporte à IP multicast, ou seja, nenhum pacote enviado via IP multicast é confirmado pelos receptores. Os dados são transmitidos via unicast para todos os receptores.

- `TX_MULTICAST`: utilizado quando o transmissor recebe a confirmação da recepção de pacotes via IP multicast.

No caso de uma escolha explícita do usuário pelo modo de transmissão unicast ou multicast, não serão feitas alterações do modo de transmissão pelo protocolo. Os padrões do protocolo entretanto não serão alterados. Por exemplo, mesmo no modo `TX_MULTICAST_ONLY` as retransmissões são realizadas via unicast.

Quando o suporte à IP multicast nativo for detectado, ele será utilizado para a transmissão de dados entre o transmissor e os receptores que suportarem esta tecnologia. Caso transmissor e receptores suportem IP multicast, exatamente 50% dos pacotes são transmitidos via multicast nativo [Jeacle e Crowcroft 2005].

Caso sejam retransmitidos três pacotes consecutivos o modo do receptor será automaticamente alterado para `TX_UNICAST_GROUP` pelo fato do suporte à IP multicast não estar funcionando apropriadamente (caso contrário retransmissões não seriam necessárias).

2.3.5 Implementação Atual

A implementação atual do protocolo TCP-XM foi realizada através de modificações na pilha TCP/IP lwIP [Dunkels 2001; Jeacle 2005; Dunkels 2004]. A avaliação da implementação foi realizada com uma aplicação chamada *mcp* [Jeacle e Crowcroft 2005], que utiliza exclusivamente o protocolo TCP-XM para transmissão de dados.

Apesar das facilidades encontradas na implementação e utilização de um protocolo no contexto do usuário, é muito difícil realizar uma avaliação precisa sobre o desempenho do protocolo TCP-XM devido aos fatores a seguir:

- sobrecarga ocasionada pelo sistema operacional no gerenciamento dos processos que executam no contexto de usuário.

- sobrecarga ocasionada pelo funcionamento da pilha lwIP, que emula TCP sobre UDP [Jeacle 2005].

- Falta de uma mecanismo de janelas deslizantes na pilha TCP/IP lwIP [Jeacle et al. 2005], restringindo a comunicação a um esquema de Stop-and-Wait [Kurose e Ross 2000].

3 ARQUITETURA DA IMPLEMENTAÇÃO

A implementação do protocolo TCP-XM envolveu duas linhas de desenvolvimento distintas que foram realizadas paralelamente. A primeira corresponde à implementação realizada no subsistema de rede do kernel GNU/Linux. A segunda corresponde ao ferramental criado no contexto do usuário para testes com a funcionalidade acrescida ao kernel. As próximas seções oferecem uma visão geral sobre o funcionamento de uma sessão TCP-XM no kernel GNU/Linux, bem como um relato do trabalho realizado no contexto do usuário.

Na Seção 3.1 são apresentados os diagramas de transição de estado do protocolo TCP-XM, que servem como um guia para o entendimento detalhado do funcionamento do protocolo, descrito Capítulo 4.

3.1 Diagrama de Transição de Estados

Um Diagrama de Transição de Estados (DTE) mostra todos os estados possíveis de um objeto, os eventos que mudam seu estado, as condições que devem ser satisfeitas antes que uma transição (mudança de estado) ocorra e as ações (atividades) durante a vida do objeto [Marmion 2004].

O protocolo TCP-XM possui duas máquinas de estados distintas. A primeira determina o método utilizado para transmissão de segmentos. A segunda determina o comportamento do protocolo em relação aos estados da sessão TCP-XM desde sua criação até seu término.

3.1.1 Modo de Transmissão

O estado do modo de transmissão determina se determinado segmento será transmitido utilizando IP multicast nativo ou uma emulação de multicast sobre unicast.

De uma forma bem resumida, os estados existentes são `TX_UNICAST_ONLY`, `TX_UNICAST_GROUP_ONLY`, `TX_UNICAST_GROUP`, `TX_MULTICAST` e `TX_MULTICAST_ONLY`. Uma descrição mais detalhada dos estados pode ser encontrada na Seção 2.3 ou em [Jeacle 2005]. A variável que armazena o estado da transmissão chama-se `txmode` e é declarada dentro da estrutura `tcpxm_sock`, no arquivo `include/linux/tcpxm.h`.

A Figura 3.1 ilustra as transições entre os possíveis modos de transmissão de uma sessão TCP-XM.

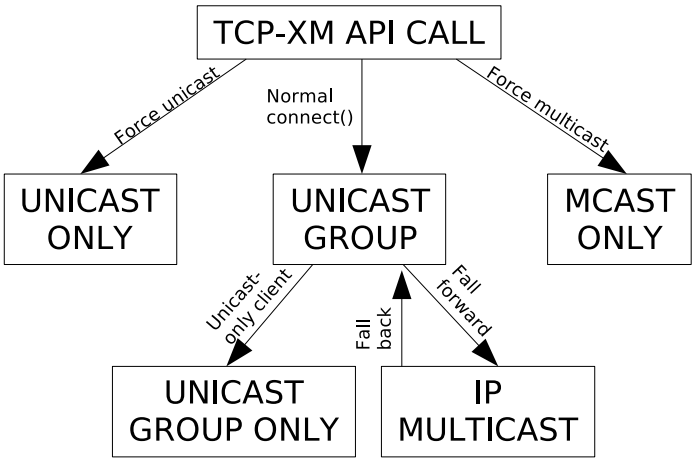

Figura 3.1: Diagrama de Estados do protocolo TCP-XM

3.1.2 Sessão

A máquina de estados utilizada na implementação do protocolo TCP-XM é muito similar a máquina de estados do protocolo TCP implementada no kernel GNU/Linux. Essa similaridade deve-se principalmente à duplicação inicial do código TCP realizada para aproveitar as mesmas estruturas de dados.

O estado atual de um socket é armazenado na variável `sk_state`, localizada dentro da estrutura `tcpxm_sock`. Um socket pode assumir um dos estados a seguir, dependendo do estágio da sessão:

- `TCPXM_SYN_SENT`: segmento SYN foi enviado requisitando uma conexão ao hospedeiro remoto. Socket local aguarda um segmento SYN-ACK.

- `TCPXM_SYN_RECV`: segmento SYN recebido pelo hospedeiro remoto. Segmento SYN-ACK enviado, aguardando ACK final do 3-way handshake.

- `TCPXM_ESTABLISHED`: conexão estabelecida. A transmissão de dados ocorre neste estado.

- `TCPXM_FIN_WAIT1`: socket foi fechado localmente. Esperando a transmissão dos dados dos segmentos restantes na fila de transmissão.

- `TCPXM_FIN_WAIT2`: todos dados ainda não transmitidos foram enviados. Esperando shutdown do host remoto.

- `TCPXM_CLOSING`: ambas extremidades fecharam o socket mas ainda existem dados não enviados.

- `TCPXM_TIME_WAIT`: timeout deve ser executado para captura de retransmissões indevidas antes de entrar no estado `TCPXM_CLOSE`. O socket só entra nesse estado através dos estados `TCPXM_FIN_WAIT2` ou `TCPXM_CLOSING`. É necessário pois o hospedeiro remoto pode não ter recebido o último ACK causando a retransmissão indevida de um segmento.

- `TCPXM_CLOSE_WAIT`: o hospedeiro remoto fechou o socket e está esperando a transmissão dos dados não-enviados e o fechamento local do socket (que move o socket para o estado `TCPXM_LAST_ACK`).

- `TCPXM_LAST_ACK`: o socket foi fechado localmente após o shutdown remoto. Ainda podem existir dados não transmitidos.

- `TCPXM_CLOSE`: socket finalizado.

A Figura 3.2 ilustra as transições entre os estados de um socket TCP-XM explicados acima.

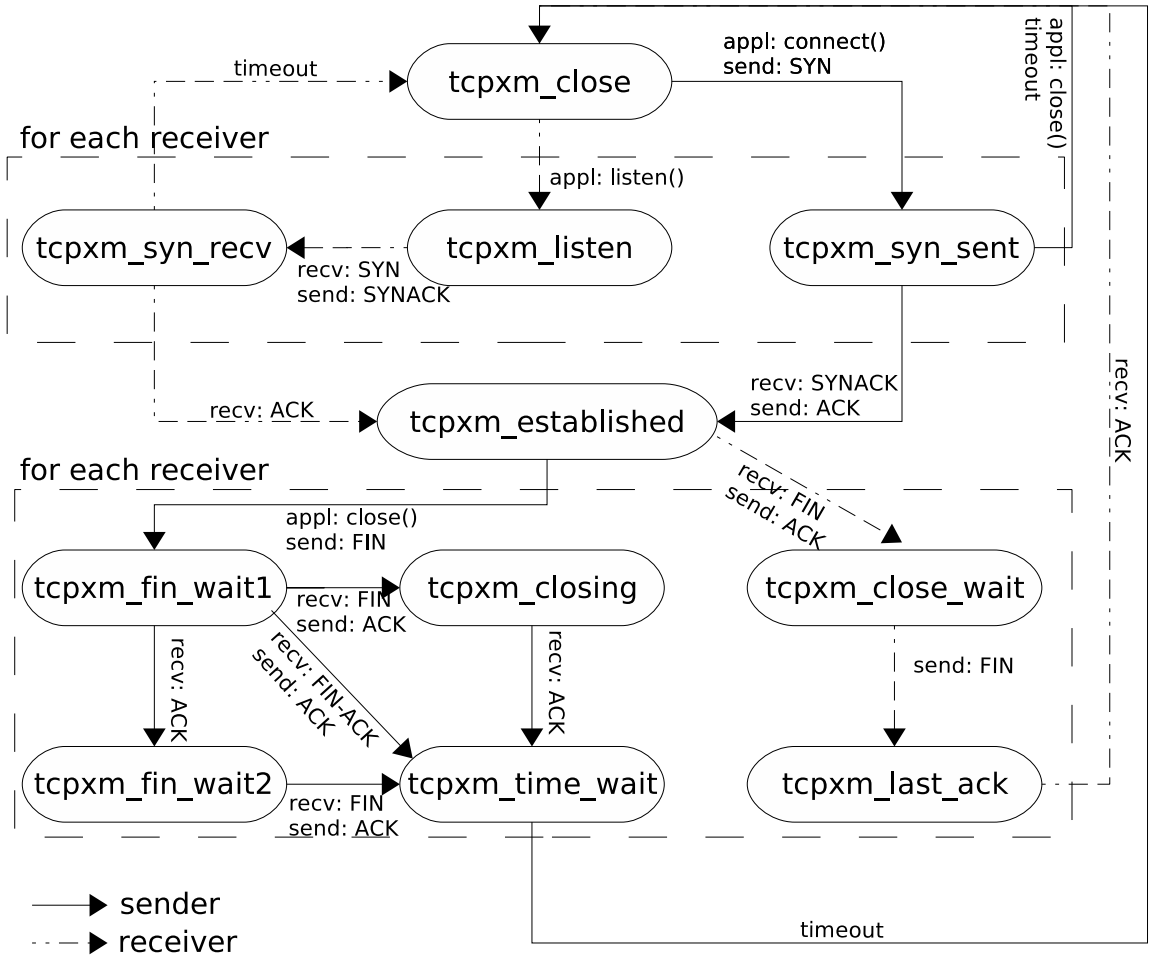

Figura 3.2: Diagrama de Transição de Estados desta implementação do TCP-XM

3.2 Funcionamento de uma Sessão TCP-XM no kernel GNU/Linux

A implementação do protocolo TCP-XM no kernel exigiu a utilização de estruturas de dados tanto para emulação de multicast sobre unicast quanto para a transmissão IP multicast nativo. As estruturas utilizadas e sua co-relação estão descritas na próxima Subseção.

A transmissão multicast confiável de arquivos utilizando o protocolo TCP-XM no kernel GNU/Linux envolve uma fase de configuração da sessão (que pode ser realizada globalmente ou através dos parâmetros padrão), uma fase de estabelecimento da sessão (aonde ocorre a negociação do grupo multicast no 3-way-handshake), uma fase de transmissão dos dados e uma fase de finalização da sessão. Todas estas etapas são realizadas através de comandos da API de sockets BSD.

3.2.1 Estruturas de Dados

O armazenamento dos dados das sessões TCP-XM é feito através da utilização de estruturas implementadas pelo próprio subsistema de rede do kernel GNU/Linux. A única estrutura criada especificamente para o subsistema de rede é uma lista encadeada que armazena informações como endereço unicast e grupo multicast dos receptores que estão comunicando-se através de sessões TCP-XM. Esta lista encadeada chama-se *TCPXM_RECEIVERS* e está descrita em detalhes no Capítulo 4. Durante a transmissão, a lista é percorrida para transmissão unicast dos segmentos de rede, caso necessário. O diagrama na Figura 3.3 ilustra a organização das estruturas de dados utilizadas pelo transmissor em uma sessão TCP-XM.

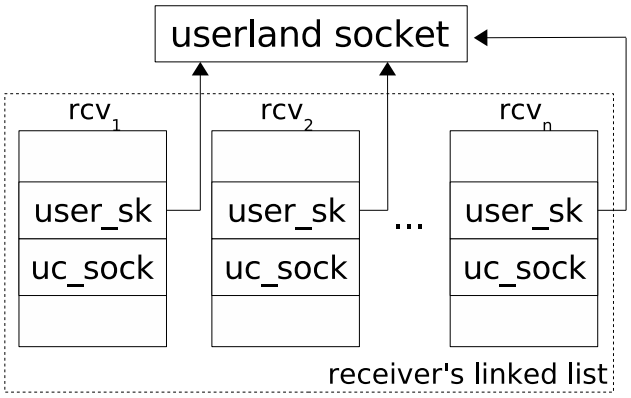

Figura 3.3: Estruturas de dados utilizadas pelo transmissor

O receptor utiliza as mesmas estruturas de dados de uma sessão do protocolo TCP, ou seja, a mesma estrutura ilustrada na Figura 2.1.

3.2.2 Preparação da Sessão

Nesta fase, o transmissor configura os parâmetros de funcionamento do protocolo e informa ao kernel o endereço unicast dos receptores. Para isso o programador deve criar no contexto de usuário um socket `SOCK_STREAM` com número de protocolo `IP_PROTO_TCPXM`. Após a criação do socket, o programador deve utilizar a chamada `setsockopt` no nível `SOL_TCPXM` passando como parâmetro a opção `TCPXM_RCVADD`, para informar o endereço unicast e a porta que deve ser utilizada em cada conexão unicast.

O receptor deve criar um socket `SOCK_STREAM` com número de protocolo `IP_PROTO_TCPXM` para aceitar a conexão unicast do transmissor e posteriormente receber dados por esse socket.

Maiores detalhes sobre a preparação da sessão podem ser obtidos diretamente da listagem do código fonte dos arquivos [Menegotto e Barcellos 2006;

Menegotto e Barcellos 2006].

3.2.3 Estabelecimento da Sessão

Nesta fase, o transmissor conecta-se aos endereços unicast dos receptores através de um 3-way-handshake. No segmento SYN, o transmissor envia o endereço de um grupo multicast aos receptores (224.0.1.5 por padrão). A Figura 3.4 ilustra as principais funções utilizadas para o envio do segmento SYN ao endereço unicast de cada receptor.

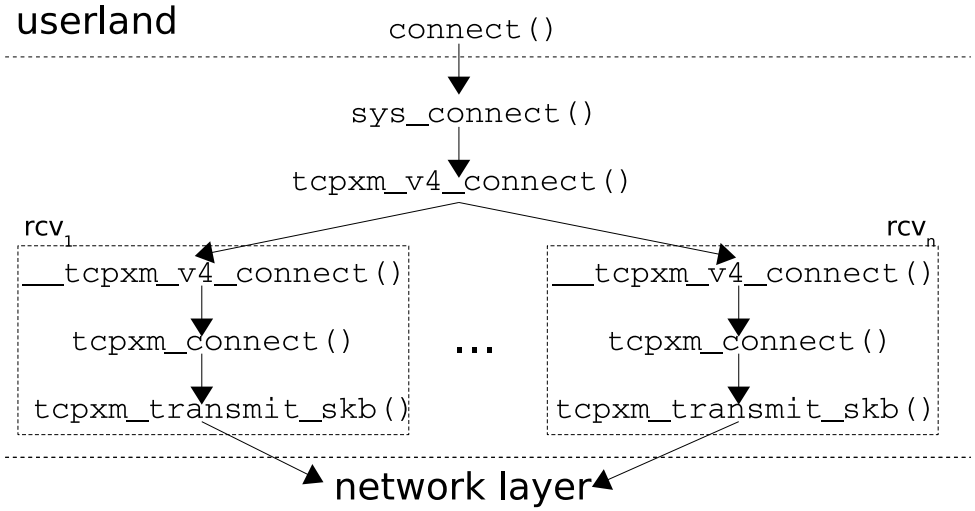

Figura 3.4: Transmissão do segmento SYN ao endereço unicast de cada receptor

Caso os receptores possuam outro endereço de grupo multicast configurado para esta sessão, eles enviam este endereço para o transmissor no segmento SYN-ACK. O caminho ilustrado na Figura 3.5 é utilizado pelo receptor para envio do segmento SYN-ACK.

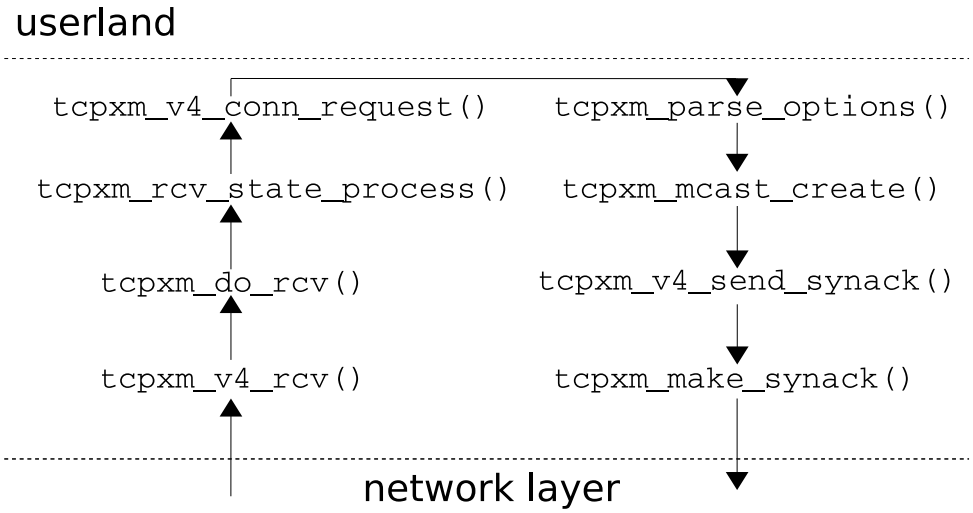

Figura 3.5: Recepção do segmento SYN e envio do segmento SYN-ACK

Ao receber o segmento SYN-ACK, o receptor conecta-se ao grupo multicast escolhido. Após, envia um ACK para o receptor finalizando o 3-way-handshake. O caminho

ilustrado na Figura 3.6 é utilizado para implementação destas tarefas.

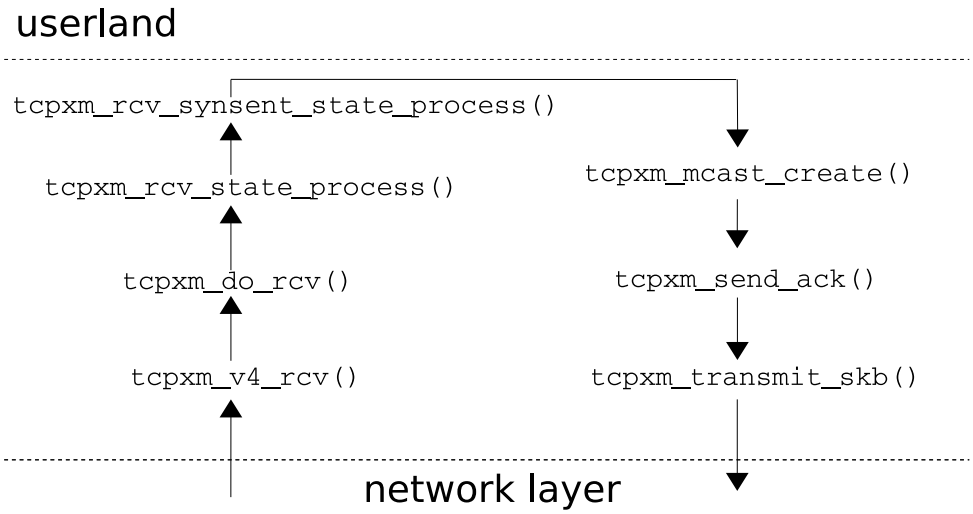

Figura 3.6: Recepção do segmento SYN-ACK e envio do segmento ACK

Maiores detalhes sobre a preparação da sessão podem ser obtidos diretamente da listagem do código fonte dos arquivos [Menegotto e Barcellos 2006; Menegotto e Barcellos 2006; Menegotto e Barcellos 2006; Menegotto e Barcellos 2006].

3.2.4 Transmissão dos Dados

Após o estabelecimento de todas as sessões unicast com os receptores o protocolo começa o envio de dados. Toda função de envio de mensagens da API de sockets BSD utilizada no contexto de usuário acaba utilizando alguma variação da rotina do kernel `tcpxm_sendmsg`. Esta rotina está localizada no arquivo `tcpxm.c` e realiza as funções a seguir:

1. verifica se o modo de transmissão atual é o mais adequado para transmissão dos dados. Esta verificação é realizada através da contabilização dos pacotes multicast recebidos. Caso a quantidade de pacotes multicast recebidos for superior a 50% de toda a transmissão, o modo de transmissão utilizado é `TX_MULTICAST`. Caso contrário, opta-se pela emulação de multicast sobre unicast.

2. percorre a lista encadeada `receivers` executando a função `__tcpxm_sendmsg` para cada nodo da lista encadeada correspondente à estrutura `sock` passada como parâmetro.

3. atualiza os contadores de pacotes enviados.

A função `__tcpxm_sendmsg` transforma os dados do contexto do usuário em `skb's`. A replicação dos dados em todas as PCBs é realizada pelo laço `list_for_each`, que percorre todas PCBs associadas à sessão e enfileira o `skb` criado na fila de transmissão `sk_write_queue` correspondente.

O envio do `skb` para a camada de rede inicia quando a função `tcpxm_push_pending_frames` é executada. A função `tcpxm_push_pending_frames` chama a função

`__tcpxm_push_pending_frames`. Esta função realiza os testes básicos para transmissão do `skb` através da função `tcpxm_write_xmit`.

Somente após a função `tcpxm_write_xmit` chamar a função `tcpxm_transmit_skb`, o segmento é endereçado e transmitido para a camada de rede. Além de realizar a transmissão unicast do `skb`, a função `tcpxm_transmit_skb` verifica a necessidade de transmissão do `skb` utilizando IP multicast nativo. Caso seja necessário, o `skb` é duplicado (ou clonado) e endereçado para o grupo multicast negociado durante o estabelecimento da sessão. Para não enviar duas vezes o mesmo pacote utilizando IP multicast nativo o protocolo controla o último número de seqüência multicast enviado. Esta informação é armazenada na variável `last_mc_seq` da PCB.

A transmissão IP multicast nativa encaminha o `skb` diretamente para a camada de rede através da função `ip_build_and_send_pkt`. A transmissão multicast sobre unicast encaminha o `skb` para a camada de rede através da função `ip_queue_xmit`.

Ao sair da camada de rede, o `skb` é transmitido pela interface de rede e chega ao receptor (numa situação ideal). O receptor deve enviar um segmento ACK confirmando a recepção do pacote. O campo de opções do segmento de confirmação deve conter o resultado da contabilização da recepção dos pacotes multicast, que indica ao transmissor a existência de suporte multicast no receptor que enviou o segmento ACK.

A Figura 3.7 ilustra o caminho do `skb` desde o contexto do usuário até a camada de rede.

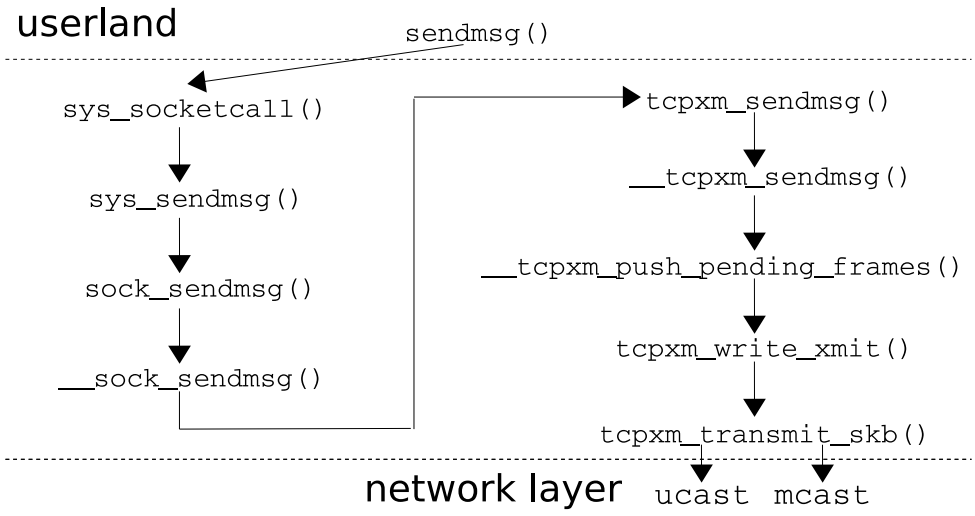

Figura 3.7: Envio de um segmento TCP-XM

3.2.5 Recepção de dados

A primeira função do protocolo TCP-XM executada durante a recepção de um pacote é a função `tcpxm_v4_rcv`. Ela realiza testes básicos para verificar a integridade do segmento recebido e endereça o `skb` para processamento dependendo do estado em que encontra-se a sessão.

Primeiramente, é verificado o tipo do endereço IP destino do `skb` recebido. Caso for multicast, a flag `ismcast` do `skb` é marcada, sinalizando para as demais funções que trata-se de um `skb` recebido via IP multicast nativo. Caso o endereço destino for unicast, a flag permanece inalterada.

Após esta verificação, é necessário localizar a estrutura `sock` destino do `skb`. O algoritmo de localização depende do tipo do `skb` recebido. Caso for multicast, a função utilizada é a `__inet_tcpxm_lookup`. Esta função realiza as comparações a seguir (representadas pelo pseudo-código abaixo) para determinar a estrutura `sock` destino do `skb`:

1. $iphdr \rightarrow dest == pcb \rightarrow mcast\_group\_ip$
2. $iphdr \rightarrow src == pcb \rightarrow remote\_ip$
3. $tcphdr \rightarrow dest == pcb \rightarrow local\_port$

Já se o `skb` for unicast, a função utilizada é a `__inet_lookup`. Esta é a mesma função de localização da estrutura `sock` do protocolo TCP, que realiza as comparações a seguir (representadas pelo pseudo-código abaixo) para determinar a estrutura `sock` destino do `skb`:

1. $iphdr \rightarrow dest == pcb \rightarrow local\_ip$
2. $iphdr \rightarrow src == pcb \rightarrow remote\_ip$
3. $tcphdr \rightarrow dest == pcb \rightarrow local\_port$
4. $tcphdr \rightarrow src == pcb \rightarrow remote\_port$

Caso a estrutura `sock` não for localizada, o `skb` é descartado. Caso contrário, o `skb` é passado para a função `tcpxm_v4_do_rcv`, que encaminha o `skb` conforme o estado da estrutura `sock` localizada. Caso a estrutura `sock` estiver no estado `TCPXM_ESTABLISHED`, o `skb` é encaminhado para processamento pela função `tcpxm_rcv_established`. Para todos os demais estados a função que processa o `skb` chama-se `tcpxm_rcv_state_process`.

A função `tcpxm_rcv_established` é a principal função utilizada durante a transmissão de dados. Ela processa o `skb` recebido atualizando as variáveis de controle das janela de controle e congestionamento. Além disso, a função envia a confirmação do segmento de dados recebido, caso necessário. Uma das últimas atribuições da função é colocar o `skb` recebido na fila de recepção `sk_receive_queue`.

A função `tcpxm_rcv_state_process` realiza o processamento do `skb` nos possíveis estados restantes para uma estrutura `sock`. Ela também processa o `skb` recebido, atualizando as variáveis de controle das janela de controle e congestionamento. Caso necessário, um segmento de confirmação também é transmitido. Normalmente, os pacotes não são colocados na fila de recepção `sk_receive_queue`, pois essa função é utilizada durante o estabelecimento e a finalização da sessão TCP-XM.

O processo no contexto de usuário recebe os dados que estão armazenados na fila `sk_receive_queue` através da função `tcpxm_recvmsg`. Esta função retira os `skbs` das filas de recepção da estrutura `sock` (`sk_receive_queue`, `sk_backlog` e `prequeue`) e repassa a quantidade de bytes requisitada para o processo no contexto do usuário. Esta função também atualiza os contadores de pacotes recebidos (tanto os contadores multicast quanto os unicast). Ela também insere no vetor `msegrcv` o número de seqüência do `skb` recebido e a tecnologia de transmissão utilizada (multicast ou unicast). Esta informação serve para detecção do suporte a IP multicast, durante a montagem do segmento ACK.

Como pode ser percebido o receptor possui pequenas alterações de comportamento em relação ao fluxo de recepção normal do protococolo TCP. O caminho executado para recepção de um segmento através do *slow path* está ilustrado na Figura 3.8. O caminho executado para recepção de um segmento através do *fast path* está ilustrado na Figura 3.9.

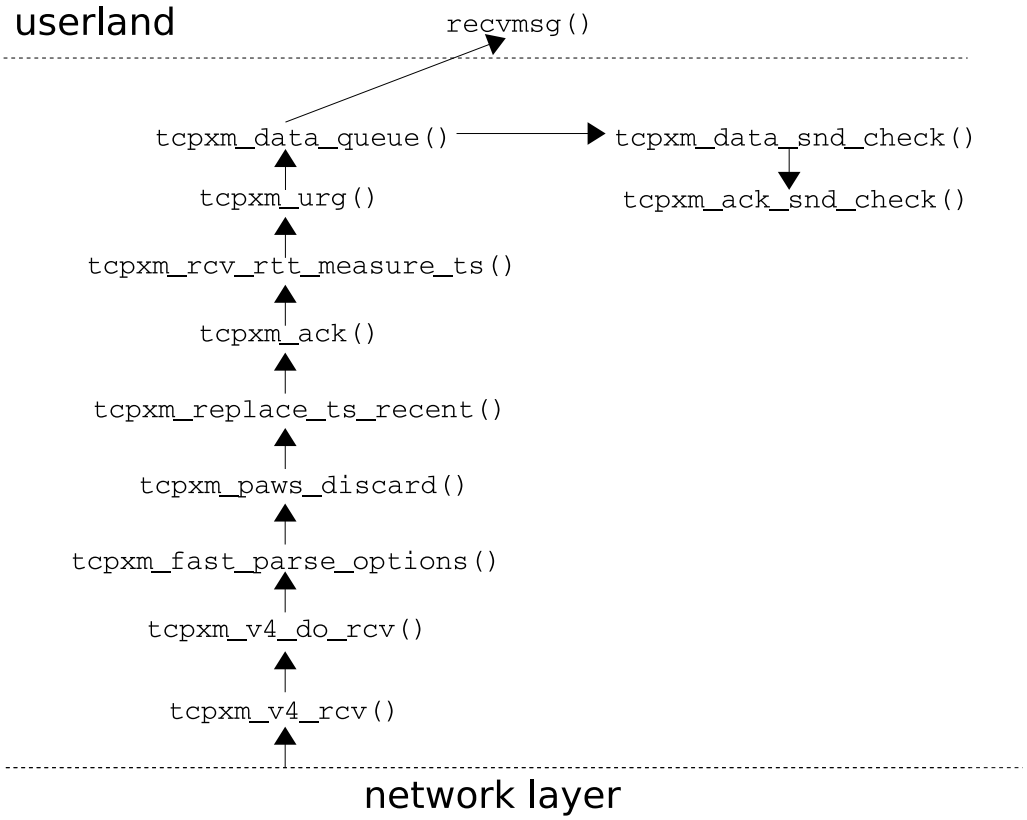

Figura 3.8: Recepção de um segmento de dados: Slow Path

3.2.6 Finalização da Sessão

A finalização de uma sessão TCP-XM depende da maneira como a sessão é encerrada e do estado atual do socket. Numa finalização normal, um segmento FIN é enviado a cada receptor. O caminho executado no transmissor é ilustrado na Figura 3.10.

Ao receber o segmento FIN cada receptor envia um segmento ACK sinalizando a recepção do segmento FIN. O caminho de funções executado no receptor para transmissão do segmento ACK é ilustrado na Figura 3.11.

Após mandar o segmento ACK cada receptor manda um segmento FIN sinalizando o término da sessão ao transmissor. O caminho de funções executado no receptor para transmissão do segmento FIN é ilustrado na Figura 3.12.

O transmissor envia então um segmento ACK para cada receptor finalizando a sessão definitivamente. O caminho de funções executado pelo transmissor para envio do segmento ACK é ilustrado na Figura 3.13.

Caso ocorram problemas com o transmissor um segmento RST é enviado a todos. O caminho de funções executado pelo transmissor para envio de um segmento RST é ilustrado na Figura 3.14.

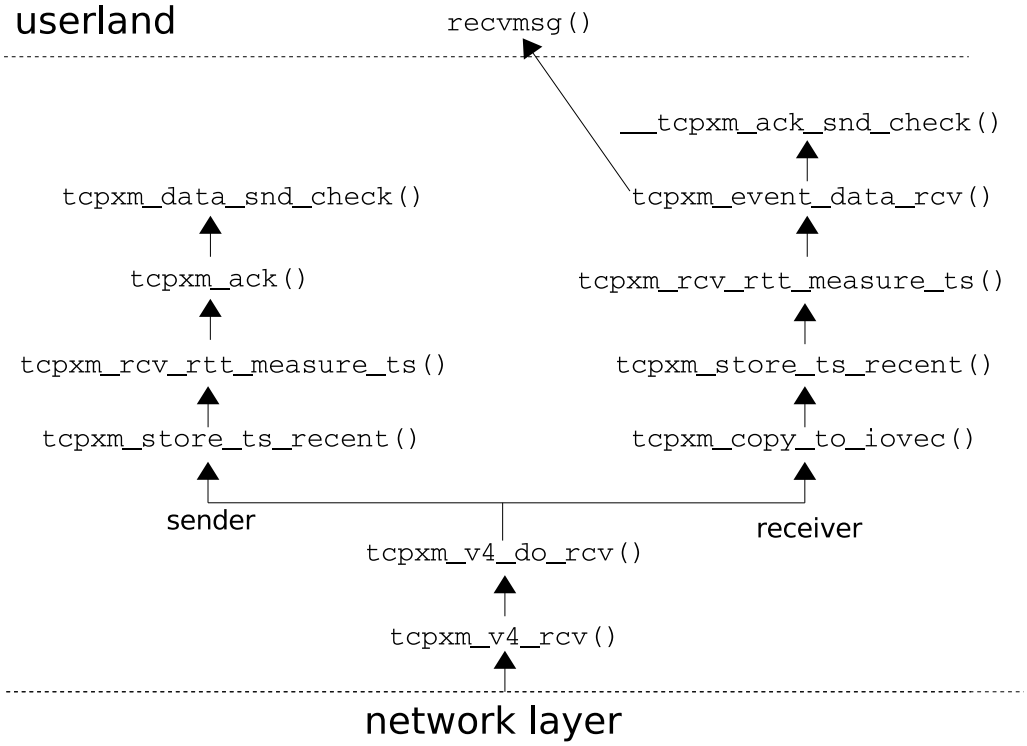

Figura 3.9: Recepção de um segmento de dados: Fast Path

3.3 Implementação No Contexto de Usuário

A implementação do protocolo TCP-XM exigiu a criação de uma VLAN de máquinas virtuais empregando o utilitário Qemu e a implementação de programas no contexto de usuário para realização de testes do funcionamento do protocolo. A inserção do protocolo TCP-XM no sniffer TCPDump também foi realizada para depuração do conteúdo dos pacotes transmitidos e recebidos.

Após a implementação, foi criada uma aplicação para validação da implementação realizada. Esta aplicação chama-se FTPXM (File Transfer Program XM) e transmite arquivos utilizando o protocolo TCP-XM.

Maiores detalhes sobre estas implementações podem ser obtidos nas Subseções 3.3.1, 3.3.3 e 3.3.2 e nos endereços [Menegotto 2006; Menegotto 2006].

3.3.1 Ferramentas utilizadas no desenvolvimento

O utilitário Netconsole foi utilizado nos scripts abaixo para visualização de mensagens de erro. Este utilitário foi criado por Ingo Molnar e é parte integrante do kernel *mainline* desde a versão 2.4.10 [Kleen 2001]. Sua função é enviar mensagens *ooops* através de pacotes UDP para outra estação que esteja escutando na porta especificada durante a inicialização do kernel. Isto é útil em casos que um *ooops* congela determinado nodo da VLAN. Maiores informações sobre este utilitário podem ser encontradas em [Viro 2006].

A instanciação de uma VLAN de máquinas virtuais com o Qemu pode ser obtida através de shell scripts. Foram criados os seguintes scripts para instanciação da VLAN:

- `qemu-vlan.sh`: instancia uma VLAN de máquinas virtuais com no máximo 4 nodos.

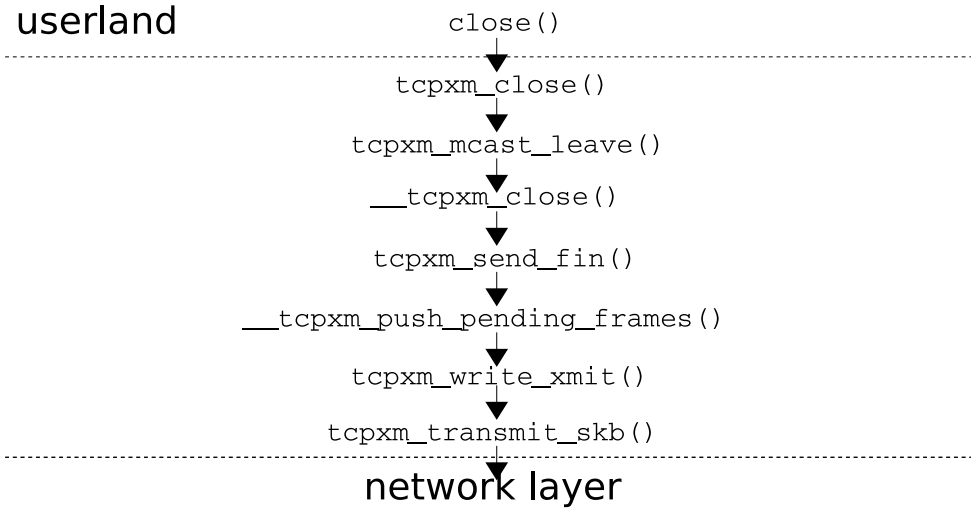

Figura 3.10: Transmissor encerra sessão com segmento FIN

- `qemu-node.sh`: instancia somente um nodo receptor numa VLAN já instanciada pelo comando `qemu-vlan.sh`.
- `qemu-netconsole-sender.sh`: instancia uma VLAN de máquinas virtuais de no máximo quatro nodos com suporte ao netconsole. O kernel deve estar compilado com suporte ao utilitário Netconsole para que este script funcione corretamente. A saída das mensagens de kernel da primeira máquina virtual instanciada será na terceira máquina virtual instanciada, caso ela esteja escutando na porta 6666. A ferramenta *netcat* [Hobbit 2006] pode ser utilizada para esta finalidade.
- `qemu-netconsole-receiver.sh`: instancia uma VLAN de máquinas virtuais de no máximo quatro nodos com suporte ao netconsole. O kernel deve estar compilado com suporte ao utilitário Netconsole para que este script funcione corretamente. A saída das mensagens de kernel da segunda máquina virtual instanciada será na primeira máquina virtual instanciada, caso ela esteja escutando na porta 6666. A ferramenta *netcat* [Hobbit 2006] pode ser utilizada para esta finalidade.
- `qemu-boot.sh`: instancia uma máquina virtual interligada com a máquina host. Este script é utilizado para cópia de dados da máquina host para os nodos da VLAN.

Foram implementados também pequenos programas com comandos da API de sockets BSD para testar as funcionalidades do protocolo durante o seu desenvolvimento. Os programas a seguir foram criados:

- **1toN**: programa que transmite cento e cinqüenta mensagens de 10 bytes aos receptores utilizando o protocolo TCP-XM.
- **commands**: diversos programas que testam comandos isolados da API de sockets BSD utilizando o protocolo TCP-XM (bind, accept, etc.).
- **opts**: programa utilizado para testes no processo de estabelecimento de uma sessão TCP-XM.

A listagem do código fonte destes scripts está disponível em [Menegotto 2006].

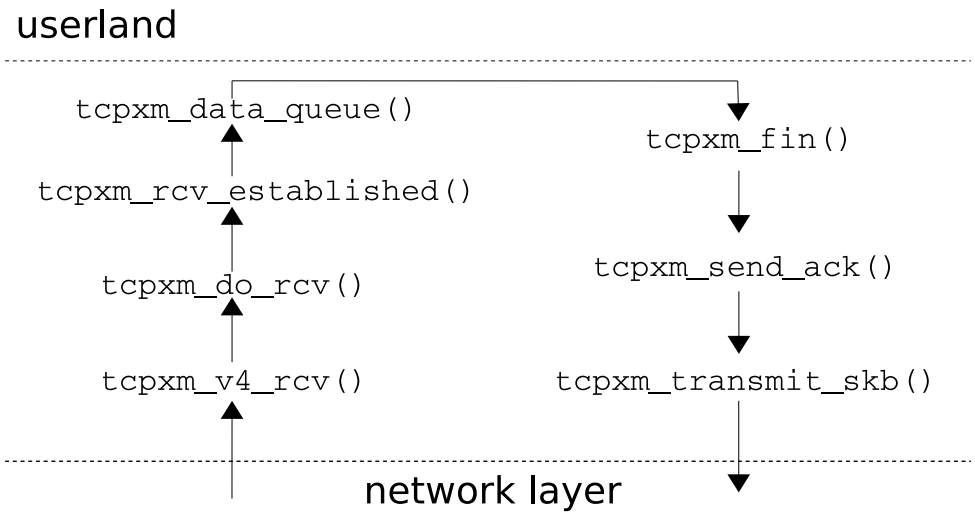

Figura 3.11: Receptor envia segmento ACK ao receber FIN

3.3.2 Alterações no Analisador TCPDump

A visualização do conteúdo dos pacotes transmitidos e recebidos em uma sessão TCP-XM foi realizada utilizando o *sniffer* TCPDump. Para isto foi necessário incluir o protocolo TCP-XM no mesmo, processo que consistiu na:

- duplicação do código-fonte do protocolo TCP com um novo nome (TCPXM).
- alteração de informações *hard-coded* para inserção do protocolo (ex. Número do Protocolo).
- alteração dos scripts de makefile para geração do binário tcpdump com suporte ao protocolo TCP-XM.

Os arquivos do código-fonte original do protocolo TCPDump-3.9.4 alterados para inclusão do protocolo TCP-XM foram:

- `interface.h`: contém as definições das funções de tratamento dos protocolos suportados pelo TCPDump. Foi incluído o protótipo da função `tcpxm_print` e uma chave MD5 para utilização do sniffer.
- `ipproto.c`: contém a lista dos protocolos suportados pelo TCPDump. Foi incluída uma linha contendo o número `IP_PROTO_TCPXM` e o nome do protocolo ("tcpxm").
- `netdissect.h`: contém a definição da chave MD5 utilizada pelo protocolo TCP-XM no TCPDump.
- `print-ip.c`: contém a implementação da rotina que seleciona o tratador de pacotes IP recebidos. Foi adicionada uma entrada no comando switch da função `ip_print_demux()` e uma chamada a função `tcpxm_print()` com os parâmetros corretos.

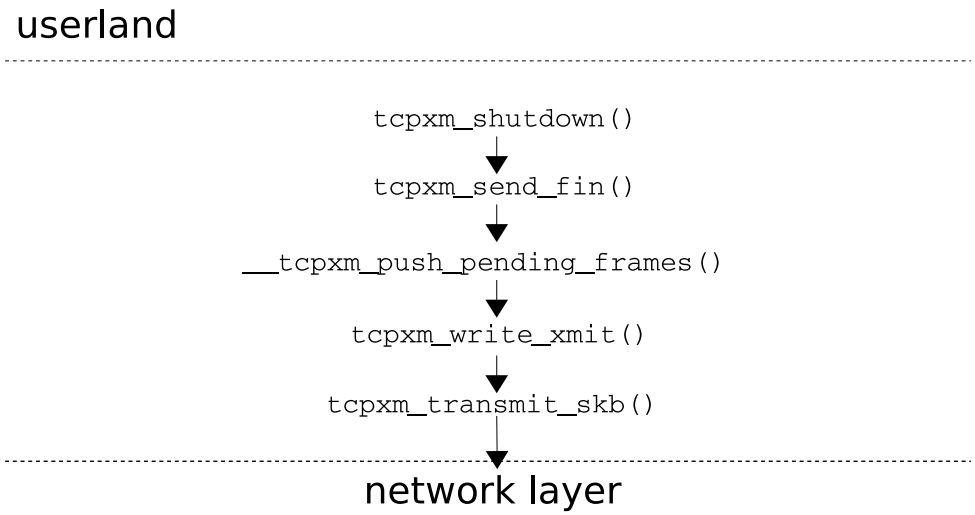

Figura 3.12: Receptor envia segmento FIN

- `print-tcpxm.c`: contém a implementação de todas as rotinas utilizadas na inclusão do protocolo TCP-XM no TCPDump.
- `tcpxm.h`: contém as declarações de constantes utilizadas na inclusão do protocolo TCP-XM no TCPDump.
- `tcpdump.c`: contém a implementação da função `main` do TCPDump. Foi adicionada uma atribuição para a chave MD5 utilizada pelo protocolo TCP-XM.

Maiores informações sobre o código-fonte referente a estas alterações pode ser encontrado em [Menegotto 2006]. Maiores informações sobre o TCPDump podem ser obtidas em [Lenz 2006].

3.3.3 Aplicativo FTPXM

O aplicativo FTPXM realiza a transferência de arquivos de um transmissor para vários receptores utilizando o protocolo TCP-XM. O binário que executa no transmissor chama-se *ftpxm*, e o binário dos receptores chama-se *serverxm*. O funcionamento de cada programa está descrito a seguir.

3.3.3.1 ftpxm

O programa *ftpxm* implementa o transmissor de uma sessão TCP-XM. Através dos parâmetros da linha de comando executada, é possível alterar características do transmissor da sessão TCP-XM. Os parâmetros a seguir podem ser utilizados para configuração do comportamento do transmissor:

- `-u`: especifica o nome do arquivo que contém a lista de endereços unicast e a porta que os receptores estarão ouvindo. A sintaxe do arquivo exige um receptor por linha no formato <endereço IP>/<porta>.
- `-p`: define a porta de transmissão.

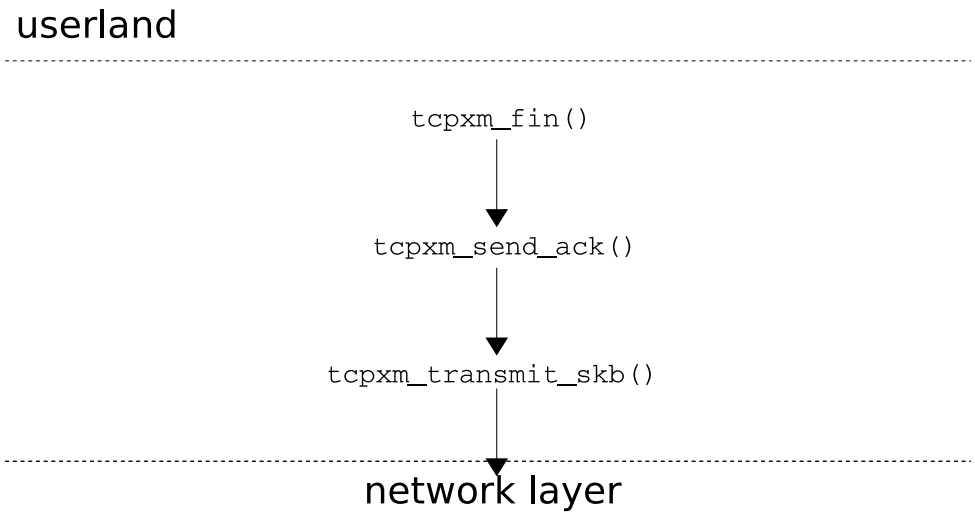

Figura 3.13: Transmissor envia um segmento ACK após recepção de segmento FIN

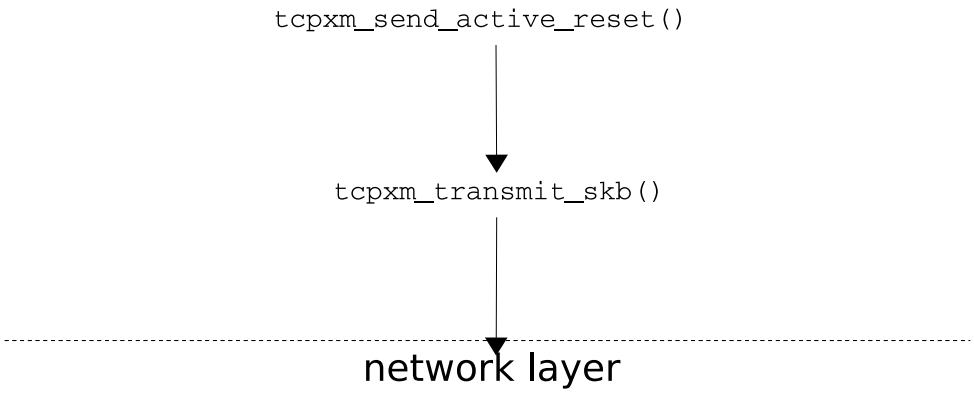

Figura 3.14: Envio de um segmento RST

- `-m`: define o comportamento do protocolo TCP-XM em relação a tecnologia de transmissão. Se o modo for 0 o protocolo utiliza transmissão multicast somente se detectar suporte à IP multicast nativo na rede. Se o modo for 1 o protocolo utiliza somente a emulação de multicast sobre unicast para transmissão de dados. Se o modo for 2 o protocolo utiliza somente IP multicast nativo sem realizar nenhuma tentativa de transmissão através da emulação de multicast sobre unicast.

- `-g`: define o endereço IP do grupo multicast da sessão.

- `-f`: especifica o nome dos arquivos que devem ser transmitidos. Máscaras e expressões do shell podem ser utilizadas neste campo.

Caso nenhuma opção seja especificada, as opções padrão do kernel são utilizadas. É possível alterar os valores padrão do kernel através das variáveis *sysctl*, conforme especifidado na Seção 4.4.

Após o estabelecimento da sessão, o transmissor envia um sinal aos receptores indicando o início da transmissão do arquivo. Após este sinal o transmissor envia o nome do arquivo aos receptores e sinaliza o início da transmissão do conteúdo do arquivo. Após a finalização da transmissão do conteúdo do arquivo, o transmissor sinaliza esse fato aos receptores.

A listagem do código fonte e maiores detalhes sobre a implementação do *ftpxm* podem ser obtidos em [Menegotto 2006].

3.3.3.2 serverxm

O programa *serverxm* implementa o receptor de uma sessão TCP-XM. Assim como no transmissor, é possível alterar características do receptor da sessão TCP-XM via linha de comando. Os parâmetros a seguir podem ser utilizados para configurar o comportamento do receptor:

- `-p`: define a porta de recepção.
- `-m`: define o comportamento do protocolo TCP-XM em relação a tecnologia de transmissão. Se o modo for 0 o protocolo utiliza transmissão multicast somente se detectar suporte à IP multicast nativo na rede. Se o modo for 1 o protocolo utiliza somente a emulação de multicast sobre unicast para transmissão dos dados. Se o modo for 2 o protocolo utiliza somente IP multicast nativo sem realizar nenhuma tentativa de transmissão através da emulação de multicast sobre unicast.
- `-g`: define o endereço IP do grupo multicast da sessão.

Após o estabelecimento da sessão o receptor espera pelo marcador que sinaliza o início do nome do arquivo. Após receber o nome de arquivo o receptor cria um arquivo com o nome especificado no sistema de arquivos. Após receber o marcador que sinaliza o início do conteúdo do arquivo o programa redireciona os dados recebidos para o arquivo criado.

A listagem do código fonte e maiores detalhes sobre a implementação do *serverxm* podem ser obtidos em [Menegotto 2006].

4 IMPLEMENTAÇÃO NO KERNEL

A implementação do protocolo TCP-XM no subsistema de rede do kernel GNU/Linux foi fortemente baseada na implementação do protocolo TCP Reno encontrada neste kernel. Portanto, a arquitetura do código-fonte que implementa o protocolo TCP-XM é praticamente idêntica à estrutura do protocolo TCP Reno.

A implementação começou com a criação da infra-estrutura para inserção do protocolo TCP-XM no subsistema de rede do kernel GNU/Linux. Após a inserção do protocolo, foram implementadas rotinas para o estabelecimento e finalização de sessões $1 \rightarrow N$, tornando possível a transmissão de segmentos multicast sobre unicast.

No passo seguinte, foi realizada a criação das restrições relativas à maquina de estados de modo de transmissão. Após várias abordagens infrutíferas, o código que emulava multicast sobre unicast foi alterado para transmitir segmentos utilizando IP multicast nativo. A recepção de segmentos foi tradada logo depois, juntamente com a sincronização das janelas de transmissão.

O tamanho do código fonte da implementação do kernel do protocolo TCP-XM ficou relativamente grande (cerca de 14.200 linhas), o que tornou impossível a publicação do mesmo nos anexos deste trabalho. O leitor interessado pode encontrar maiores detalhes sobre a implementação em [Menegotto 2006].

As alterações realizadas no subsistema de rede para inserção do protocolo estão descritas na Seção 4.1. Os arquivos criados durante o desenvolvimento do protocolo TCP-XM e os parâmetros de configuração do protocolo estão descritos nas demais seções deste Capítulo.

4.1 Inserção do Protocolo

Na inserção do protocolo TCP-XM no subsistema de rede do kernel GNU/Linux, foram declaradas estruturas de dados que definem o tipo, a família e as *callbacks* (funções que responderão por eventos) do protocolo. São elas:

- `struct proto_ops`: estrutura de ponteiros para funções que responderão a eventos do protocolo TCP-XM. Essas funções fazem a ligação entre a camada socket e a camada AF_INET.
- `struct proto`: estrutura de ponteiros para funções do protocolo TCP-XM que fazem a ligação entre a camada AF_INET e o protocolo de transporte.
- `struct net_protocol`: estrutura de ponteiros para funções de recepção do protocolo de transporte utilizadas na comunicação com a camada IP. É utilizada durante a inserção do protocolo TCP-XM na pilha TCP/IP.

- `struct inet_protosw`: define um protocolo de transporte. Contém ponteiros para as estruturas proto_ops e proto.

- `struct net_proto_family`: determina a família e a função de criação de sockets que o protocolo TCP-XM utiliza.

A correta declaração e inicialização destas estruturas, juntamente com a alocação de memória para as estruturas básicas do protocolo, garantem a inicialização do protocolo TCP-XM na pilha TCP/IP do subsistema de rede do kernel GNU/Linux. As estruturas básicas para o funcionamento de um protocolo que não pertença a família AF_INET são a `struct proto_ops`, a `struct net_proto_familily` e a `struct proto`.

A estratégia utilizada na implementação dos protocolos DCCP e SCTP no kernel GNU/Linux (duplicar a infra-estrutura de inicialização do protocolo TCP) diminui os contratempos iniciais da implementação, visto que o código de inicialização do protocolo TCP já funciona há muitos anos. Esta estratégia também foi adotada na implementação do protocolo TCP-XM descrita nessa monografia.

A função que implementa a inicialização da pilha TCP/IP no subsistema de rede chama-se `inet_init`. Nesta função foi incluído código para inicialização do protocolo TCP-XM. Esta rotina está localizada no arquivo `net/ipv4/af_inet.c`. Foi incluída ainda neste arquivo uma entrada referente ao protocolo TCP-XM no array `inetsw_array`, que contém ponteiros para as estruturas básicas de cada protocolo componente da pilha TCP/IP (TCP, UDP, IGMP, etc...).

A inicialização da MIB SNMP do protocolo TCP-XM é realizada pela rotina `init_ipv4_mibs`, localizada no arquivo `net/ipv4/af_inet.c`. A declaração da MIB é feita dentro dos arquivos `net/ipv4/proc.c` e `include/net/snmp.h` através da estrutura `struct snmp_mib`. Os itens da MIB estão definidos no arquivo `include/linux/snmp.h`.

O mecanismo procfs foi utilizado para configuração de parâmetros de uma sessão TCP-XM. Para isso foi necessário incluir nos arquivos `net/ipv4/sysctl_net_ipv4.c` e `include/linux/sysctl.h` os parâmetros de configuração desejados. Esta lista de parâmetros está descrita na Subseção 4.4.1

A identificação do protocolo TCP-XM no subsistema de rede é realizada através da constante IPPROTO_TCPXM. Esta constante está declarada no arquivo `include/linux/in.h` como $IPPROTO\_TCPXM = 35$ e é armazenada na variável `sk_protocol` da estrutura `sock`.

A emulação de multicast sobre unicast o protocolo TCP-XM necessita do endereço unicast de todos os receptores. Uma lista encadeada chamada `receivers` armazena esta informação. A manipulação desta lista é feita através da API `klist`, que implementa uma lista encadeada circular dentro do kernel GNU/Linux [4].

A estrutura de dados utilizada para criação dos nodos da lista encadeada `receivers` é declarada no arquivo `include/net/sock.h`:

```
struct tcpxm\_receivers {
        struct list\_head list;
        struct sockaddr\_in *sin;
        struct socket *uc\_sock;
        struct sock *user\_sk;
};
```

- `struct list_head list`: contém ponteiros para o nodo anterior e para o próximo nodo da lista encadeada.
- `struct sockaddr_in *sin`: contém o endereço IP unicast, a porta e a família de endereços utilizada para conectar-se ao receptor.
- `struct socket *uc_sock`: utilizada para comunicação unicast com o receptor.
- `struct sock *user_sk`: ponteiro para a estrutura `sock` do socket do contexto de usuário que identifica a sessão que o receptor está associado.

A declaração desta lista encadeada é realizada no arquivo `net/ipv4/tcpxm.c`:

```
struct tcpxm\_receivers receivers;
```

A inicialização de uma `klist` é feita através da macro `INIT_LIST_HEAD` [4]. A lista encadeada `receivers` é inicializada dentro da função `tcpxm_init`. Esta função é executada durante a inicialização do subsistema de rede do kernel GNU/Linux:

```
INIT\_LIST\_HEAD(&receivers.list);
```

A estrutura `sk_buff` também foi modificada. Foi incluído um ponteiro para estrutura que representa o cabeçalho de um segmento TCP-XM (`struct txmh`) e a flag `ismcast`, que identifica o tipo do `sk_buff` (multicast ou unicast). A estrutura `sk_buff` está declarada no arquivo `include/skbuff.h`.

A compilação do protocolo depende da criação de um parâmetro de configuração do kernel referenciando o protocolo TCP-XM. O parâmetro de compilação para o protocolo TCP-XM (`CONFIG_IP_TCPXM`) foi implementado no arquivo `net/ipv4/Kconfig`. O processo de compilação necessita saber quais arquivos fazem parte do código-fonte do protocolo TCP-XM. Esta informação foi incluída no arquivo `net/ipv4/Makefile`.

4.2 Arquivos de Cabeçalho (include/)

4.2.1 include/linux/tcpxm.h

Este é o header aonde estão declaradas as estruturas de dados e funções dependentes de arquitetura. Esta dependência deve-se principalmente a diferença entre método de leitura dos bytes de memória (*low-endian* e *big-endian*) e também à diferença na capacidade de armazenamento dos tipos padrões (como int e char) em arquiteturas diferentes. As estruturas de dados a seguir são declaradas neste arquivo:

- `tcpxmhdr`: representa o cabeçalho de um pacote do protocolo TCP-XM. É manipulado durante a criação do `sk_buff` pela função `__tcpxm_sendmsg` e durante o endereçamento do pacote pela função `tcxpm_transmit_skb`.
- `tcpxminfo`: armazena informações estatísticas de uma sessão TCP-XM. Os campos são atualizados em tempo de execução e podem ser consultados pelo usuário através da opção `TCPXM_GETINFO` da chamada de sistema `setsockopt`.
- `tcpxm_sack_block`: define um bloco SACK no protocolo TCP-XM.

- `tcpxm_options_received`: armazena o campo de opções no cabeçalho de um segmento TCP-XM.
- `tcpxm_request_sock`: representa uma requisição de conexão. É utilizada durante o 3-way handshake, sendo criada quando um segmento SYN válido é processado pelo receptor e destruída quando a conexão passa para o estado `TCPXM_ESTABLISHED`.
- `segrcv`: armazena o meio (multicast ou unicast) de transmissão dos últimos *n* pacotes recebidos (128 por padrão). Este valor pode ser aumentado através da alteração da constante `TCPXM_MAX_SEGMENTS` localizada neste mesmo arquivo.
- `tcpxm_sock`: representa uma PCB do protocolo TCP-XM. Armazena o estado de uma sessão TCP-XM desde seu estabelecimento até o seu encerramento. Em uma sessão TCP-XM existe uma `tcpxm_sock` associada ao socket do contexto de usuário e uma `tcpxm_sock` para cada receptor, armazenando os dados das conexões unicast.
- `tcpxm_timewait_sock`: representa um socket no estado `TCPXM_TIME_WAIT`.

Neste arquivo também são implementadas funções de conversão entre tipos de socket que são dependentes de arquitetura. Dentre elas pode-se citar:

- `tcpxm_sk`: retorna um ponteiro para a estrutura `tcpxm_sock` armazenada na estrutura `sock` passada como parâmetro.
- `tcpxm_twsk`: retorna um ponteiro para a estrutura `tcpxm_timewait_sock` armazenada na estrutura `sock` passada como parâmetro.
- `tcpxm_rsk`: retorna um ponteiro para a estrutura `tcpxm_request_sock` armazenada na estrutura `req` passada como parâmetro.

São declaradas ainda constantes que podem ser encapsuladas no campo de opções do cabeçalho de um segmento TCP-XM e constantes que representam o modo de transmissão utilizado numa sessão TCP-XM.

A listagem do código fonte do arquivo `tcpxm.h` e maiores detalhes sobre a implementação (relatados nos comentários) podem ser encontrados em [Menegotto e Barcellos 2006].

4.2.2 include/net/tcpxm.h

Neste arquivo são declaradas estruturas de dados e funções do protocolo TCP-XM independentes de arquitetura. São implementadas também funções declaradas como `static inline` que são independentes de arquitetura. As estruturas de dados a seguir são declaradas neste arquivo:

- `tcpxm_func`: armazena ponteiros para funções do protocolo TCP-XM como a `queue_xmit`, a `conn_request` e a `setsockopt`.
- `tcpxm_congestion_ops`: representa um mecanismo de controle de congestionamento para o protocolo TCP-XM. Esta estrutura armazena ponteiros para os `callbacks` que definem um método de controle de congestionamento.

- `tcpxm_seq_afinfo`: armazena informações necessárias para declaração de uma nova entrada no sistema de arquivos *procfs*. Contém informações que definem os `callbacks` que devem ser utilizados nas operações de I/O realizadas sobre entradas *procfs*. Maiores informações sobre o sistema *procfs* podem ser encontradas em [4].

- `tcpxm_iter_state`: representa uma entrada no sistema de arquivos *procfs*. Para maiores informações sobre a interface `seq_file` consulte [Eklektix 2003].

Também são declaradas neste arquivo constantes que definem o comportamento do protocolo. Dentre elas pode-se citar:

- `TCPXM_SYN_RETRIES`: determina quantas retransmissões de segmentos SYN podem ser realizadas durante uma tentativa de estabelecimento de conexão.

- `TCPXM_MAX_QUICKACKS`: determina o número máximo de pacotes ACKs enviados no modo *quickack* para acelerar o algoritmo *slow start*.

- `TCPXM_FIN_TIMEOUT`: define quanto tempo o protocolo deve esperar antes de destruir sockets que estão no estado `TCPXM_FIN_WAIT2` e com a flag `TCPXM_TIMEWAIT` ativa.

São declaradas também constantes utilizadas durante o 3-way handshake, constantes que definem o comportamento do protocolo, variáveis sysctl (que definem parâmetros de configuração do protocolo) e constantes que definem a MIB SNMP do protocolo TCP-XM.

Funções curtas que podem ser implementadas como *static inline* também estão codificadas neste arquivo. São exemplos dessas funções:

- `tcpxm_initialize_rcv_mss`: retorna uma estimativa do MSS utilizado pelo receptor baseado nas estatísticas anteriores de conexão.

- `tcpxm_update_main_seq`: retorna e/ou atualiza o número de seqüência no socket do contexto de usuário. Essa informação serve para sincronizar os números de seqüência dos segmentos gerados pela função `__tcpxm_sendmsg`.

- `tcpxm_v4_check`: retorna a soma de verificação TCP-XM baseado nas informações passadas como parâmetro. Este valor é utilizado para verificar a integridade do segmento após sua recepção pela função `tcpxm_v4_rcv`.

- `tcpxm_mcast_create`: realiza a associação (*join*) ao grupo multicast escolhido caso IP multicast esteja disponível na rede.

- `tcpxm_mcast_leave`: encerra a associação com o grupo multicast caso IP multicast tenha sido utilizado durante a sessão. Este procedimento é chamado durante a finalização da sessão TCP-XM.

Os protótipos das demais funções do protocolo TCP-XM também estão declaradas neste arquivo. A listagem do código fonte do arquivo `tcpxm.h` e maiores detalhes sobre a implementação (relatados nos comentários) podem ser encontrados em [Menegotto e Barcellos 2006].

4.2.3 include/net/tcpxm_states.h

Neste arquivo estão declaradas constantes que representam os estados possíveis de um socket TCP-XM (TCPXM_SYN_SENT, TCPXM_ESTABLISHED, TCPXM_LAST_ACK, etc...).

A listagem do código fonte do arquivo `tcpxm_states.h` e maiores detalhes sobre a implementação (relatados nos comentários) podem ser encontrados em [Menegotto e Barcellos 2006].

4.2.4 include/net/tcpxm_ecn.h

Neste arquivo estão declaradas as estruturas de dados e funções que implementam o mecanismo de *Explicit Congestion Notification* (ECN) no subsistema de rede do kernel GNU/Linux [9]. Este mecanismo adiciona um bit no cabeçalho IP indicando congestionamento de determinado nodo de rede.

A listagem do código fonte do arquivo `tcpxm_ecn.h` e maiores detalhes sobre a implementação (relatados nos comentários) podem ser encontrados em [Menegotto e Barcellos 2006].

4.3 Arquivos na Implementação IPv4 (net/)

4.3.1 net/ipv4/tcpxm.c

Neste arquivo estão declarados vetores que controlam o comportamento do protocolo TCP-XM em relação à alocação de memória. Em redes de alto desempenho a correta configuração destes valores traz um aumento significativo no *throughput* da rede [Andreasson 2002]. Dentre eles pode-se citar:

- `sysctl_tcpxm_wmem`: vetor de três posições que armazena na primeira posição o tamanho máximo de memória que pode ser utilizado por um socket durante a alocação. Na segunda posição é armazenado o tamanho máximo de memória utilizado durante as operações de transmissão de pacotes. A terceira posição do vetor é destinada ao tamanho máximo do buffer de envio que pode ser utilizado por um socket TCP-XM.

- `sysctl_tcpxm_rmem`: vetor de três posições que armazena na primeira posição o tamanho máximo de memória que pode ser utilizado por um socket durante a alocação. Na segunda posição é armazenado o tamanho máximo de memória utilizado as operações de recepção de pacotes. A terceira posição do vetor é destinada ao tamanho máximo do buffer de recepção que pode ser utilizado por um socket TCP-XM.

Estão implementadas neste arquivo funções e procedimentos utilizados em todos os demais arquivos do código-fonte do protocolo TCP-XM. São elas:

- `tcpxm_enter_memory_pressure`: liga a flag `tcpxm_memory_pressure` sinalizando ao gerenciador da memória virtual que o protocolo TCP-XM não está conseguindo alocar memória contígua suficiente para seu funcionamento. Maiores informações sobre o mecanismo de `memory_pressure` podem ser obtidas em [Bligh et al. 2005].

- `tcpxm_poll`: retorna um *bitmask* (int) representando eventos relacionados a um socket do protocolo TCP-XM.

- `tcpxm_mark_push`: adiciona a flag `TCPXMCB_FLAG_PSH` ao segmento, sinalizando que o `sk_buff` passado como parâmetro já pode ser transmitido para a camada de rede.

- `tcpxm_forced_push`: verifica a necessidade de envio do primeiro segmento da fila de transmissão da estrutura `tcpxm_sock` passada como parâmetro. Esta verificação é baseada no número de seqüência e no tamanho máximo da janela de transmissão atuais.

- `tcpxm_skb_entail`: coloca o `sk_buff` passado como parâmetro no final da fila de transmissão `sk_write_queue`.

- `tcpxm_mark_urg`: liga a flag URG no `sk_buff` passado como parâmetro indicando que a transmissão deste `sk_buff` para a camada de rede deve ser realizada o mais rápido possível.

- `tcpxm_push`: inicia a transmissão do primeiro `sk_buff` da fila de transmissão `sk_write_queue`.

- `do_tcpxm_sendpages`: transmite dados copiando-os diretamente das páginas de memória para a interface de rede, sem a necessidade de cópia entre as diversas funções de encapsulamento de dados.

- `__tcpxm_sendpage`: verifica se a interface de rede utilizada possui o suporte necessário para a transmissão de pacotes utilizando o mecanismo *TCP Zero Copy*. Caso exista suporte na interface de rede, a função `do_tcpxm_sendpages` é utilizada para o envio de pacotes.

- `tcpxm_sendpage`: inicia a transmissão de um segmento utilizando o mecanismo *TCP Zero Copy* no protocolo TCP-XM . Em resumo, o mecanismo *TCP Zero Copy* minimiza o número de cópias necessárias para a transmissão de dados dentro da pilha TCP/IP. Os dados são enviados diretamente da memória para a placa de rede (através do uso de scatter/gather DMA). Esta função percorre a lista encadeada `receivers` utilizando a função `__tcpxm_sendpage` para enviar a mensagem aos receptores.

- `tcpxm_select_size`: retorna o tamanho de memória necessário em bytes para alocação do cabeçalho de um segmento TCP-XM. Esta função é utilizada durante a criação e inicialização da estrutura `sk_buff`, dentro da função `__tcpxm_sendmsg`.

- `__tcpxm_sendmsg`: encapsula mensagens passadas como parâmetro dentro de `sk_buffs`. Nesta função os `sk_buffs` são alocados e inicializados. Somente na momento do envio o pacote é endereçado (dependendo da tecnologia disponível na rede).

- `tcpxm_set_txmode`: troca o modo de transmissão do `tcpxm_sock` passado como parâmetro dependendo das configurações atuais do kernel.

- `tcpxm_sendmsg`: inicia a transmissão de um segmento através do protocolo TCP-XM. Esta função percorre a lista encadeada `receivers` utilizando a função `__tcpxm_sendmsg` para encapsular os dados. Esta função também verifica a necessidade da troca do modo de transmissão utilizado para determinado receptor.

- `tcpxm_recv_urg`: entrega segmentos TCP-XM com a flag URG ligada diretamente ao socket no contexto de usuário. A confirmação dos dados recebidos é realizada no próximo segmento ACK transmitido nesta sessão.

- `tcpxm_cleanup_rbuf`: verifica a necessidade de confirmação do primeiro segmento da fila de recepção de pacotes. Caso necessário o `sk_buff` é confirmado através do envio de um segmento ACK pela função `tcpxm_send_ack`.

- `tcpxm_prequeue_process`: retira segmentos da fila de recepção `prequeue` e coloca na fila de recepção `sk_backlog`. Segmentos armazenados nesta fila ainda não foram requisitados pelo processo do contexto de usuário mas já foram confirmados.

- `tcpxm_recv_skb`: percorre a fila de recepção `sk_receive_queue` retornando o primeiro `sk_buff` encontrado para processamento.

- `tcpxm_read_sock`: processa um `sk_buff` retornado pela função `tcpxm_recv_skb`, copiando os dados diretamente para o descritor do contexto de usuário (passado como parâmetro). Esta função também atualiza os contadores e realiza a coleta de dados para verificação do suporte a IP multicast nativo.

- `tcpxm_recvmsg`: lê determinada quantidade de bytes das filas de recepção de pacotes e copia estes dados para o socket do contexto de usuário. Esta função também atualiza os contadores e realiza a coleta de dados para verificação do suporte a IP multicast nativo.

- `tcpxm_close_state`: muda o estado da sessão passada como parâmetro através da estrutura `sock` para TCPXM_CLOSE.

- `__tcpxm_close`: finaliza a conexão unicast entre o transmissor e um receptor. Esta é a função que transmite o segmento FIN ao endereço unicast do receptor passado como parâmetro.

- `tcpxm_close`: realiza o término de uma sessão TCP-XM (quando uma chamada close() é realizada sobre o socket criado no contexto de usuário). Esta função utiliza as funções `__tcpxm_close` e `tcpxm_mcast_leave` para atingir seus objetivos.

- `tcpxm_need_reset`: verifica a necessidade de reinicialização da estrutura `sock` passada como parâmetro. Caso necessário um segmento RST é enviado.

- `__tcpxm_disconnect`: envia um segmento RST para o endereço unicast do receptor passado como parâmetro.

- `tcpxm_disconnect`: reinicializa uma conexão através do envio de um segmento RST. A lista encadeada `receivers` é percorrida e a chamada `__tcpxm_disconnect` envia o segmento RST.

- `tcpxm_setsockopt`: implementa a chamada de sistema `setsockopt` da abstração de sockets BSD. Esta chamada serve para configurar opções de uma sessão TCP-XM através da própria aplicação. Maiores informações sobre as opções de configuração disponíveis nesta chamada podem ser encontradas na Subseção 4.4.2.

- `tcpxm_get_info`: retorna informações estatísticas sobre uma sessão TCP-XM através da estrutura `tcpxm_info` passada como parâmetro. Podem ser visualizadas informações sobre os mecanismos SACK, FACK, Slow Start e retransmissão de pacotes. Podem ser visualizados também os valores de MSS, RTT, temporizadores e janelas de transmissão/recepção.

- `__tcpxm_shutdown`: fecha o socket utilizado para transmissão de informações ao endereço unicast do receptor passado como parâmetro e envia um segmento FIN caso necessário.

- `tcpxm_shutdown`: percorre a lista encadeada `receivers` e fecha os sockets das conexões unicast realizadas com os receptores de uma sessão TCP-XM.

- `tcpxm_ioctl`: implementa a configuração de parâmetros do protocolo TCP-XM em tempo real através da chamada de sistemas `ioctl`.

- `tcpxm_getsockopt`: implementa a chamada de sistemas `getsockopt` da abstração de sockets BSD. Esta função retorna o valor de um parâmetro configurado através da chamada de sistemas `setsockopt`. Maiores informações sobre as opções de configuração disponíveis nesta chamada podem ser encontradas na Seção 4.4.

- `set_thash_entries`: realiza o alinhamento das entradas das tabelas hash do protocolo TCP-XM. Esta função é declarada utilizando a macro `__init`, que indica ao compilador para liberar a memória utilizada pela função após a inicialização do kernel GNU/Linux.

- `tcpxm_init`: inicializa as estruturas básicas de dados para inicialização do protocolo TCP-XM na pilha TCP/IP do kernel GNU/Linux.

A listagem do código fonte do arquivo `tcpxm.c` e maiores detalhes sobre a implementação (relatados nos comentários) podem ser encontrados em [Menegotto e Barcellos 2006].

4.3.2 net/ipv4/tcpxm_input.c

Neste arquivo estão declaradas constantes e macros utilizadas na implementação do mecanismo SACK no protocolo TCP-XM. São declaradas também variáveis `sysctl` para configuração do protocolo em tempo de execução. Dentre elas pode-se citar:

- `TCPXM_FLAG_DAT`: indica se o segmento contém dados em sua área de dados.

- `TCPXM_FLAG_WIN_UPDATE`: indica que a confirmação de dados deve avançar a janela de transmissão.

- `TCPXM_FLAG_DATA_ACKED`: indica que os dados contidos neste segmento já foram confirmados.

Neste arquivo encontra-se a implementação de funções e procedimentos que processam pacotes TCP-XM recebidos pelo subsistema de rede do kernel GNU/Linux. É implementado também neste arquivo todo mecanismo de controle de congestionamento e confirmação de pacotes, bem como os mecanismos de SACK, FACK, Fast Recovery/Fast Retransmit, Quickack e Slow Start. Esta implementação está dividida nas funções e procedimentos a seguir:

- `tcpxm_measure_rcv_mss`: verifica se o tamanho em bytes da área de dados do último segmento recebido ultrapassou o MSS anunciado. Caso positivo este tamanho passa a ser o novo MSS. Esta informação é utilizada posteriormente no mecanismo *delayed ack*.

- `tcpxm_incr_quickack`: incrementa o limite de segmentos ACK que podem ser enviados quando a sessão passada como parâmetro entra no modo *quick ack*. O modo *quick ack* é utilizado quando deseja-se confirmar imediatamente cada pacote de dados recebido.

- `tcpxm_enter_quickack_mode`: coloca a conexão representada pela estrutura `sock` passada como parâmetro no modo *quick ack*.

- `tcpxm_in_quickack_mode`: verifica se a conexão representada pela estrutura `sock` passada como parâmetro está no modo *quick ack*.

- `tcpxm_fixup_sndbuf`: atualiza o tamanho do buffer de transmissão após o estabelecimento da conexão unicast. Esta atualização é baseada no MSS negociado durante o 3-way handshake e nos valores configurados pelo usuário no vetor `sysctl_tcpxm_wmem`.

- `__tcpxm_grow_window`: calcula valores para as variáveis `rcv_wnd` (que representa o tamanho do buffer de recepção em bytes) e `window_clamp` (que representa a janela de recepção anunciada). O cálculo é baseado no MSS negociado durante o 3-way handshake e utilizado pela função `tcpxm_grow_window` conforme [Stevens 1997].

- `tcpxm_grow_window`: atualiza variáveis dos mecanismos de controle de congestionamento e Slow Start [Stevens 1997]. O cálculo dos novos valores é realizado pela função `__tcpxm_grow_window`.

- `tcpxm_fixup_rcvbuf`: atualiza o tamanho em bytes do buffer de recepção da estrutura `sock` passada com parâmetro. O buffer de recepção é utilizado para isolar a latência da rede e a latência da aplicação (que pode estar ocupada e não requisitar bytes na mesma velocidade em que eles são recebidos).

- `tcpxm_init_buffer_space`: procedimento executado quando a conexão passada como parâmetro através da estrutura *sock* entrar no estado `TCPXM_ESTABLISHED`. Esta função inicializa os buffers de recepção e as variáveis utilizadas nos mecanismos de *Slow Start* e *Congestion Avoidance* através das funções `tcpxm_grow_window`, `tcpxm_fixup_sndbuf` e `tcpxm_fixup_rcvbuf`.

- `tcpxm_clamp_window`: reduz o tamanho em bytes do buffer de recepção anunciado quando a conexão passada como parâmetro através da estrutura `sock` atingir o limite superior de memória.

- `tcpxm_rcv_rtt_update`: estima o RTT do link de transmissão dos dados. O cálculo é baseado no algoritmo *Dynamic Right-Sizing*, que em resumo permite ao receptor estimar o tamanho da janela de congestionamento do transmissor para melhorar o cálculo do buffer de recepção anunciado. [Fisk e Feng 2001] A descrição original do algoritmo não utiliza timestamps porém a implementação encontrada no kernel GNU/Linux utiliza uma estimativa dos timestamps dos pacotes para realizar o cálculo do RTT devido aos problemas encontrados em [Floyd et al. 2000; Zhang et al. 2002].

- `tcpxm_rcv_rtt_measure`: verifica a possiblidade do cálculo do RTT na conexão passada como parâmetro através da estrutura `sock`. Caso positivo, o cálculo é realizado e as variáveis utilizadas no algoritmo *Dynamic Right-Sizing* (`rcv_rtt_est.seq` e `rcv_rtt_est.time`) são atualizadas para realização do próximo cálculo.

- `tcpxm_rcv_rtt_measure_ts`: realiza o cálculo do RTT baseado somente no timestamp dos segmentos de confirmação.

- `tcpxm_rcv_space_adjust`: atualiza o tamanho em bytes do buffer de recepção. Este procedimento é executado quando os dados recebidos são copiados para o contexto do usuário liberando espaço no buffer para nova recepção de dados.

- `tcpxm_event_data_recv`: inicializa o temporizador utilizado antes do início do envio de segmentos de confirmação (ACK). Uma verificação da necessidade de colocar a conexão no modo *quick ack* também é realizada. Caso a área de dados for maior que 128 bytes a função `tcpxm_grow_window` é utilizada para ajuste do buffer de recepção.

- `tcpxm_rtt_estimator`: calcula o *RTT atenuado* utilizando variações do RTT coletadas durante a transmissão de segmentos. O algoritmo original foi descrito em [Jacobson 1988], porém possui problemas na coleta de timestamps. O algoritmo implementado nessa função é um novo algoritmo proposto em [Karn e Partridge 1991], que melhora o algoritmo original e resolve os problemas com timestamps.

- `tcpxm_set_rto`: calcula o RTO [Paxson e Allman 2000] da sessão passada como parâmetro através da estrutura `sock`, baseado no cálculo do *RTT atenuado*. O *Retransmission TimeOut* é o tempo máximo de espera de um segmento de confirmação (ACK) antes da retransmissão unicast do segmento de dados.

- `tcpxm_bound_rto`: atribui um limite superior ao RTO da sessão passada como parâmetro. Este limite é declarado no parâmetro `TCPXM_RTO_MAX`.

- `tcpxm_update_metrics`: armazena os valores de RTT e Slow Start Threshold (snd_ssthres) das sessões TCP-XM no momento de sua finalização. Estes valores são utilizados pelos mecanismos *Slow Start* e *Congestion Avoidance*.

- `tcpxm_init_cwnd`: inicializa a janela de congestionamento da conexão representada pela estrutura `tcpxm_sock` com os valores definidos em [7]. As variáveis inicializadas são posteriormente utilizadas para controle de congestionamento [Stevens 1997].

- `tcpxm_init_metrics`: atribui à estrutura `sock` passada como parâmetro valores iniciais de parâmetros do protocolo TCP-XM como *RTT* e *Slow Start Threshold*. Esses valores são armazenados pela função `tcpxm_update_metrics` quando uma sessão entra no estado `TCPXM_TIME_WAIT` ou `TCPXM_CLOSE`.

- `tcpxm_update_reordering`: atualiza os contadores de reordenamento de segmentos da sessão passada como parâmetro através da estrutura `sock`, dependendo do mecanismo de controle de congestionamento utilizado.

- `tcpxm_sacktag_write_queue`: processa um segmento de confirmação (ACK) que contém informação SACK. Em resumo, o *Selective Acknowledgment* aumenta o *throughput* da comunicação pois permite a confirmação desordenada e a conseqüente retransmissão dos segmentos perdidos em menos de um RTT. Maiores informações sobre o mecanismo SACK podem ser encontradas em [Mathis et al. 1996].

- `tcpxm_enter_frto`: implementa o algoritmo F-RTO conforme descrito em [Sarolahti e Kojo 2005; 10]. Em resumo, este algoritmo serve para detectar retransmissões desnecessárias através do envio de dois novos segmentos de dados após a expiração do RTO. Caso o receptor confirme a recepção destes segmentos, a retransmissão é desconsiderada.

- `tcpxm_enter_frto_loss`: procedimento utilizado após a execução do procedimento `tcpxm_enter_frto` e a verificação da necessidade de retransmissão. A recepção destes segmentos indica que o mecanismo de recuperação tradicional deve ser utilizado para retransmissão do segmento de dados[Sarolahti e Kojo 2005; 10].

- `tcpxm_clear_retrans`: inicializa os contadores dos mecanismos de retransmissão SACK e FACK.

- `tcpxm_enter_loss`: implementa o processamento executado pelo mecanismo SACK quando é detectada a perda de segmentos de dados em uma sessão TCP-XM, acarretando na retransmissão unicast dos mesmos.

- `tcpxm_check_sack_reneging`: verifica a validade do conteúdo SACK contido no segmento. Esta verificação ocorre baseada nos segmentos de confirmação (ACK) já recebidos.

- `tcpxm_fackets_out`: retorna quantos segmentos estão em trânsito na rede (ainda não foram confirmados pelo receptor). Cada implementação de TCP (Reno/NewReno) possui um algoritmo diferente para este controle: no TCP Reno, o controle é feito pelo próprio mecanismo SACK. Já no caso do TCP NewReno, o algoritmo é o Forward Acknowledgment (FACK), descrito em [Mathis e Mahdavi 1996].

- `tcpxm_skb_timedout`: calcula se o RTO do `sk_buff` passado como parâmetro expirou.

- `tcpxm_head_timedout`: verifica se o RTO do primeiro segmento da fila de transmissão `sk_write_queue` da estrutura `sock` passada como parâmetro expirou.

- `tcpxm_time_to_recover`: verifica a necessidade da execução do algoritmo *FastRetransmit* conforme descrito em [Stevens 1997].

- `tcpxm_check_reno_reordering`: verifica se a conexão passada como parâmetro através da estrutura `sock` não está recebendo excesso de confirmações duplicadas. Isto indica a necessidade de reordenamento de segmentos segundo o algoritmo de *Fast Retransmit* conforme [Stevens 1997].

- `tcpxm_add_reno_sack`: incrementa a variável `sacked_out`. Quando o mecanismo SACK está sendo utilizado, essa variável representa a quantidade de segmentos confirmados pelo mecanismo SACK. Quando o mecanismo não é usado, ela representa a quantidade de segmentos desordenados não confirmados.

- `tcpxm_remove_reno_sacks`: confirma a recepção de segmentos no estado *Recovery*.

- `tcpxm_reset_reno_sack`: reinicializa a variável `sacket_out`.

- `tcpxm_mark_head_lost`: marca o primeiro segmento não confirmado da fila de transmissão como perdido, sinalizando que o mesmo deve ser retransmitido.

- `tcpxm_update_scoreboard`: verifica a existência e a veracidade de segmentos perdidos na estrutura `sock` passada como parâmetro. O algoritmo implementado nesta estrutura é uma variação do algoritmo para detecção de segmentos perdidos do TCP NewReno [Karn e Partridge 1991].

- `tcpxm_moderate_cwnd`: realiza uma redução preventiva na janela de congestionamento do transmissor (`snd_cwnd`) para evitar problemas com segmentos de confirmação que trazem valores inconsistentes relacionados à quantidade de segmentos que o transmissor pode enviar.

- `tcpxm_cwnd_down`: decrementa o tamanho em bytes da janela de congestionamento do transmissor (`snd_cwnd`) baseado na quantidade de segmentos não-confirmados pelo receptor.

- `tcpxm_packet_delayed`: verifica a ordenação dos segmentos retransmitidos baseado nos timestamps dos segmentos.

- `tcpxm_undo_cwr`: aumenta a janela de congestionamento (`snd_cwnd`) de acordo com o valor do *Slow Start Threshold* do transmissor.

- `tcpxm_may_undo`: verifica se existem retransmissões desnecessárias (indicadas pela variável `undo_retrans`) em uma sessão.

- `tcpxm_try_undo_recovery`: verifica a existência de retransmissões desnecessárias utilizando a função `tcpxm_may_undo`. Esta função também verifica a possibilidade de redução no tamanho em bytes da janela de congestionamento utilizando a função `tcpxm_moderate_cwnd`.

- `tcpxm_try_undo_dsack`: verifica a existência de retransmissões desnecessárias devido a recepção de segmentos DSACK que confirmam o segmento de dados que seria retransmitido.

- `tcpxm_try_undo_partial`: verifica a existência de retransmissões desnecessárias durante o algoritmo *Fast Recovery*.

- `tcpxm_try_undo_loss`: verifica a existência de retransmissões desnecessárias quando a conexão está no estado *Loss* da máquina de estados do TCP NewReno.

- `tcpxm_complete_cwr`: reduz o tamanho da janela de congestionamento (`snd_cwnd`) baseado no valor do *Slow Start Threshold* do transmissor.

- `tcpxm_try_to_open`: coloca a conexão representada pela estrutura `sock` passada por parâmetro no estado *Open* da máquina de estados do TCP NewReno.

- `tcpxm_fastretrans_alert`: calcula a quantidade de pacotes em trânsito na rede baseado no número de segmentos que não foram confirmados pelo receptor e no número de segmentos perdidos. Ainda neste procedimento é verificado a necessidade de redução da janela de congestionamento do transmissor.

- `tcpxm_ack_saw_tstamp`: armazena os timestamps recebidos no segmento de confirmação (ACK) e calcula o RTT utilizando a função `tcpxm_rtt_estimator` com os timestamps coletados. Os timestamps são utilizados também no mecanismo PAWS.

- `tcpxm_ack_no_tstamp`: realiza o cálculo do RTT (através da função `tcpxm_rtt_estimator`) sem a utilização de timestamps dos segmentos de confirmação (ACK).

- `tcpxm_ack_update_rtt`: inicia o cálculo do RTT de um link baseado nas informações recebidas nos segmentos de confirmação (ACK) através das funções `tcpxm_ack_saw_tstamp` e `tcpxm_ack_no_tstamp`.

- `tcpxm_cong_avoid`: executa a função principal do mecanismo de controle de congestionamento utilizado na sessão representada pela estrutura `sock` passada como parâmetro.

- `tcpxm_ack_packets_out`: reinicializa os temporizadores de recepção de segmentos de confirmação (ACK) caso existam segmentos de dados transmitidos e ainda não confirmados.

- `tcpxm_tso_acked`: verifica a existência de segmentos de dados transmitidos (e ainda não confirmados) pelo mecanismo de TSO.

- `tcpxm_usrtt`: calcula há quanto tempo o `sk_buff` passado como parâmetro foi recebido pela camada de rede. Este valor é utilizado no cálculo do RTT.

- `tcpxm_clean_rtx_queue`: remove da fila de retransmissão os segmentos de dados que não necessitam ser retransmitidos (pois foram confirmados pelo receptor).
- `tcpxm_ack_is_dubios`: verifica a validade do segmento de confirmação (ACK) recebido na sessão representada pela estrutura `sock` passada como parâmetro.
- `tcpxm_ack_probe`: verifica a necessidade do envio de segmentos de confirmação na sessão representada pela estrutura `sock` passada como parâmetro.
- `tcpxm_may_raise_cwnd`: verifica a possibilidade de crescimento do tamanho em bytes da janela de congestionamento utilizada de uma sessão.
- `tcpxm_may_update_window`: verifica a necessidade da atualização do tamanho em bytes da janela de transmissão utilizada na sessão representada pela estrutura `sock` passada como parâmetro baseado na ordem seqüencial dos identificadores passados como parâmetro.
- `tcpxm_ack_update_window`: atualiza o tamanho em bytes da janela de transmissão representada pela estrutura *sock* passada como parâmetro.
- `tcpxm_process_frto`: implementa o algoritmo de F-RTO conforme descrito em [Sarolahti e Kojo 2005; 10]. Este procedimento chama todas as demais funções envolvidas na execução do algoritmo F-RTO na sessão passada como parâmetro.
- `tcpxm_ack`: realiza o processamento de um segmento de confirmação (ACK) recebido. Esta é a rotina inicial no caminho de execução do processamento de segmentos de confirmação.
- `tcpxm_parse_options`: verifica o campo de opções de um segmento TCP-XM e atualiza a sessão conforme as opções encontradas. Este procedimento é utilizado na recepção de todos segmentos TCP-XM, inclusive durante o estabelecimento da sessão. Caso existam somente informações de timestamp a função `tcpxm_fast_parse_options` é utilizada.
- `tcpxm_fast_parse_options`: procedimento utilizado para coletar o timestamp contido no campo de opções do segmento TCP-XM recebido.
- `tcpxm_store_ts_recent`: armazena na estrutura `tcpxm_sock` o timestamp do último segmento de confirmação (ACK) recebido.
- `tcpxm_replace_ts_recent`: verifica a necessidade da atualização do timestamp de uma sessão TCP-XM.
- `tcpxm_disordered_ack`: verifica se o segmento de confirmação (ACK) recebido está fora da seqüência.
- `tcpxm_paws_discard`: descarta segmentos com o identificador seqüencial inválido segundo o mecanismo de PAWS conforme descrito em [3].
- `tcpxm_sequence`: verifica a ordenação crescente dos números de seqüência passados como parâmetro com a sessão representada pela estrutura `sock`, também passada como parâmetro.

- `tcpxm_reset`: realiza o processamento necessário na recepção de um segmento RST.

- `tcpxm_fin`: realiza o processamento necessário na recepção de um segmento FIN.

- `tcpxm_sack_extend`: verifica se o segmento confirmado pelo mecanismo SACK não está em outro bloco SACK.

- `tcpxm_dsack_set`: atualiza a informação DSACK da sessão passada como parâmetro baseada nos números de seqüência inicial e final passados como parâmetro.

- `tcpxm_dsack_extend`: verifica se o segmento confirmado pelo mecanismo SACK não está em outro bloco SACK caso o mecanismo de *dsack* não está sendo utilizado. Caso contrário atualiza a informação *dsack* da sessão.

- `tcpxm_send_dupack`: envia um segmento de confirmação (ACK) duplicado. Segmentos *dupack* são transmitidos quando ocorre reordenamento nos segmentos recebidos. Esta rotina é utilizada pelo mecanismo PAWS e pelo mecanismo FastRetransmit/FastRecovery.

- `tcpxm_sack_maybe_coalesce`: atualiza o bloco SACK da conexão passada como parâmetro através da estrutura `sock` assim que segmentos são processados pelo subsistema de rede. Este procedimento é executado somente quando o mecanismo SACK é utilizado em uma sessão.

- `tcpxm_sack_swap`: rotaciona dois blocos SACK passados como parâmetro. É utilizado durante o reordenamento de segmentos recebidos com informação SACK.

- `tcpxm_sack_new_ofo_skb`: atualiza a informação SACK recebida no `sk_buff` processado pela função `tcpxm_data_queue`. Caso não exista um bloco SACK para este número de seqüência, um novo bloco SACK é criado para armazenar a informação recebida.

- `tcpxm_sack_remove`: remove um bloco SACK após a confirmação de todos os segmentos contidos no bloco e reinicializa os contadores utilizados para implementação do mecanismo SACK na sessão passada como parâmetro através da estrutura `tcpxm_sock`.

- `tcpxm_ofo_queue`: verifica a possibilidade de transferência de um segmento da fila `out_of_order_queue` para a fila `sk_receive_queue`.

- `tcpxm_data_queue`: insere a estrutura `sk_buff` passada como parâmetro nas filas de recepção conforme a condição de recepção deste segmento. Caso não esteja ordenado pelo identificador, o segmento é inserido na fila `out_of_order_queue`; caso contrário, é inserido na fila `sk_receive_queue`.

- `tcpxm_collapse`: realiza a junção de `sk_buffs` contíguos armazenados entre o primeiro e o último `sk_buff` da fila de recepção.

- `tcpxm_collapse_ofo_queue`: realiza a junção de `sk_buffs` armazenados na fila `out_of_order_queue` utilizando a função `tcpxm_collapse`.

- `tcpxm_prune_queue`: verifica a possibilidade de redução da memória utilizada pela estrutura `sock` passada como parâmetro através da junção de segmentos localizados nas filas `sk_receive_queue` e `out_of_order_queue`.

- `tcpxm_cwnd_application_limited`: ajusta a janela de congestionamento caso ela não seja utilizada completamente durante um RTO, conforme descrito em [6].

- `tcpxm_should_expand_sndbuf`: verifica a necessidade do aumento do tamanho em bytes do buffer de transmissão. Isto ocorre quando um segmento de confirmação (ACK) é processado e libera espaço na janela de congestionamento.

- `tcpxm_new_space`: incrementa em bytes a variável que armazena o tamanho do buffer de transmissão (`sk_sndbuf`) caso o retorno da função `tcpxm_should_expand_sndbuf` seja verdadeiro, indicando a necessidade desse incremento.

- `tcpxm_check_space`: verifica o espaço disponível do buffer de transmissão e a necessidade do aumento do tamanho deste buffer. A função `tcpxm_new_space` é utilizada para esta tarefa.

- `tcpxm_data_snd_check`: verifica a existência de segmentos não transmitidos na fila de transmissão através da função `tcpxm_push_pending_frames`. Essa função é utilizada após a recepção de segmentos de confirmação (ACKs), quando é permitido o avanço da janela de transmissão.

- `__tcpxm_ack_snd_check`: verifica a necessidade do envio de segmentos de confirmação (ACKs) na sessão TCP-XM passada como parâmetro através da estrutura `sock`.

- `tcpxm_ack_snd_check`: procedimento utilizado pela função `tcpxm_rcv_established` para verificar a necessidade do envio de segmentos de confirmação ACK para a sessão representada pela estrutura `sock` passada como parâmetro. Os procedimentos de verificação estão implementados na função `__tcpxm_ack_snd_check`.

- `tcpxm_check_urg`: procedimento utilizado para processar segmentos recebidos com a flag URG habilitada.

- `tcpxm_urg`: realiza validações durante a recepção de pacotes com a flag URG habilitada. O processamento de recepção do segmento URG é realizado pela função `tcpxm_check_urg`.

- `tcpxm_copy_to_iovec`: copia os dados contidos da estrutura `sk_buff` passada como parâmetro para uma estrutura de dados do kernel GNU/Linux conhecida como *iovec*. A estrutura *iovec* é utilizada na implementação do mecanismo de E/S uniforme conhecido como UIO [Cheriton 1987].

- `__tcpxm_checksum_complete_user`: aplica o mecanismo de lock adequado a estrutura `sock` passada como parâmetro para calculo da soma de verificação do pacote passado como parâmetro. O cálculo da soma de verificação é realizado pela função `__tcpxm_checksum_complete`.

- `tcpxm_checksum_complete_user`: verifica a integridade do segmento passado como parâmetro através do cálculo da sua soma de verificação.

- `tcpxm_rcv_established`: função utilizada na recepção de segmentos quando a sessão representada pela estrutura *sock* estiver no estado TCPXM_ESTABLISHED ou TCPXM_TIME_WAIT. O processamento do segmento recebido pode ser realizado através de dois caminhos de execução distintos. No *slow path*, o algoritmo implementado é o descrito por J. Postel em [DARPA 1981]. No *fast path*, é implementado o algoritmo de previsão de cabeçalho (*header prediction*) descrito em [3].

- `tcpxm_rcv_synsent_state_process`: função utilizada para recepção de segmentos SYN-ACK quando a sessão representada pela estrutura `sock` estiver no estado TCPXM_SYN_SENT. Após a recepção de um segmento SYN-ACK, os mecanismos básicos para o estabelecimento da sessão são criados e inicializados.

- `tcpxm_rcv_state_process`: implementa o processamento necessário para recepção de segmentos em todos os estados de uma sessão TCP-XM exceto nos estados TCPXM_ESTABLISHED e TCPXM_TIME_WAIT.

A listagem do código fonte do arquivo `tcpxm_input.c` e maiores detalhes sobre a implementação (relatados nos comentários) podem ser encontrados em [Menegotto e Barcellos 2006].

4.3.3 net/ipv4/tcpxm_output.c

Neste arquivo estão declaradas as variáveis de configuração do protocolo, conforme a seguir:

- `sysctl_tcpxm_retrans_collapse`: habilita a junção de segmentos que estão na fila de retransmissão.

- `sysctl_tcpxm_tso_win_divisor`: controla o percentual da janela de congestionamento que um segmento TSO pode ocupar.

- `sysctl_tcpxm_default_state`: controla o estado inicial das sessões TCP-XM. Os possíveis estados são TX_UNICAST_ONLY (força o modo de transmissão multicast sobre unicast), `TX_MULTICAST_ONLY` (força o modo de transmissão IP Multicast nativo) ou `TX_UNICAST_GROUP` (a melhor tecnologia é escolhida para transmissão).

Estão implementadas também funções e procedimentos que iniciam a transmissão de segmentos TCP-XM para outros nodos da rede. Estão implementadas também funções e procedimentos que manipulam a janela de transmissão e a janela de congestionamento. São elas:

- `tcpxm_update_send_head`: avança o ponteiro que representa o início da fila de transmissão de pacotes `sk_write_queue` da estrutura `sock` passada como parâmetro.

- `tcpxm_acceptable_seq`: retorna o próximo número de seqüência válido na estrutura `sock` passada como parâmetro.

- `tcpxm_advertise_mss`: calcula o MSS inicial enviado no segmento SYN indicando ao receptor qual o tamanho máximo em bytes que cada segmento TCP-XM poderá ter. O tamanho do MSS depende dentre outros fatores do encapsulamento Ethernet utilizado. Para frames Ethernet 802.2 o tamanho do MSS inicial é 1460. Para frames 802.3 o MSS pode ser de até 1452 bytes. Maiore informações sobre o cálculo do MSS podem ser obtidas em [Braden 1989; Stevens 1994; Comer 1998; DARPA 1981].

- `tcpxm_cwnd_restart`: reinicializa a janela de transmissão utilizada na sessão representada pela estrutura `sock` passada como parâmetro após a expiração de um RTO.

- `tcpxm_event_data_sent`: verifica a necessidade de reinicialização da janela de congestionamento.

- `tcpxm_event_ack_sent`: finaliza o temporizador utilizado no envio de um segmento de confirmação (ACK). Este procedimento também desativa o modo *quickack* da sessão caso o mesmo esteja ativo.

- `tcpxm_select_initial_window`: calcula o tamanho inicial em bytes da janela de transmissão, da janela de congestionamento e da janela de recepção utilizadas. A janela de transmissão é negociada entre transmissor e receptor para aumentar o *throughput* do link utilizado. A janela de congestionamento é calculada pelo algoritmo *slow start*. O buffer de retransmissão é baseado no *window_clamp* anunciado durante o estabelecimento da sessão. Maiores informações sobre estas estruturas podem ser encontradas em [Braden 1989; Stevens 1994; Comer 1998; DARPA 1981].

- `tcpxm_select_window`: implementa o redimensionamento da janela de recepção anunciada utilizada numa sessão TCP-XM. Maiores informações sobre o redimensionamento da janela de recepção podem ser obtidos em [3].

- `tcpxm_transmit_skb`: transmite um `sk_buff` para a camada de rede. Esta função encaminha o `sk_buff` criado conforme o seu tipo: caso for multicast o pacote é endereçado e transmitido diretamente para o grupo multicast escolhido. Caso for unicast o pacote é adicionado no final da fila de transmissão unicast. Esta é a última função executada pelo protocolo TCP-XM (camada de transporte) durante a transmissão de um pacote. Todas as funções que transmitem segmentos TCP-XM utilizam esta função.

- `tcpxm_queue_skb`: insere o `sk_buff` passado como parâmetro no final da fila de transmissão `sk_write_queue` da estrutura `sock` passada como parâmetro.

- `tcpxm_set_skb_tso_segs`: calcula o valor da variável `tso_factor`, que indica a quantidade de segmentos que podem ser encapsulados dentro de um único `sk_buff` para envio utilizando o mecanismo TSO (caso implementado pelo driver da interface de rede).
- `tcpxm_fragment`: divide o `sk_buff` passado como parâmetro em dois novos `sk_buff`s. A quantidade de dados em bytes do primeiro `sk_buff` é informada através do parâmetro `len`. O restante dos dados é colocado no segundo `sk_buff`. Os buffers criados são inseridos na fila de transmissão *sk_write_queue*.
- `__pskb_tcpxm_trim_head`: trunca o cabeçalho da estrutura `sk_buff` passada como parâmetro conforme o número de bytes passado no parâmetro `len`.
- `tcpxm_trim_head`: trunca o cabeçalho da estrutura `sk_buff` utilizando a função `__pskb_tcpxm_trim_head` e atualiza as estruturas de dados referentes ao tamanho do pacote representado pela estrutura `sk_buff`.
- `tcpxm_sync_mss`: sincroniza o MSS utilizado na sessão representada pela estrutura `sock` passada como parâmetro com o PMTU do link descoberto pela função `tcpxm_do_pmtu_discovery`.
- `tcpxm_current_mss`: calcula o MSS atual da estrutura `sock` passada como parâmetro tomando como base o MSS negociado durante o 3-way handshake, os parâmetros descobertos pelo mecanismo SACK e o PMTU estimado do link.
- `tcpxm_cwnd_validate`: verifica se o número de pacotes em trânsito (ainda não confirmados) é menor que o tamanho da janela de congestionamento. Caso necessário a janela de congestionamento é reduzida conforme algoritmo descrito em [6].
- `tcpxm_window_allows`: retorna a quantidade de bytes que podem ser enviados segundo a janela de transmissão passada como parâmetro dentro da estrutura `tcpxm_sock`.
- `tcpxm_cwnd_test`: retorna a quantidade de segmentos que podem ser transmitidos segundo a janela de congestionamento.
- `tcpxm_init_tso_segs`: verifica a quantidade de segmentos armazenados na estrutura `sk_buff` passada como parâmetro. Um `sk_buff` pode armazenar diversos segmentos que serão fragmentados e transmitidos caso o mecanismo TSO seja utilizado.
- `tcpxm_minshall_check`: verifica se as variáveis de timestamp da conexão representada pela estrutura `tcpxm_sock` estão ordenadas temporalmente. Este teste é utilizado no algoritmo de Nagle [Nagle 1984], para redução da quantidade de pacotes transmitidos (através do agrupamento de vários segmentos pequenos em um único segmento).
- `tcpxm_nagle_check`: testa se o `sk_buff` passado como parâmetro pode ser transmitido sem violar as regras do algoritmo de Nagle [Nagle 1984]. Este teste é realizado somente quando a opção TCPXM_NODELAY está habilitada. Esta função é utilizada pela função `tcpxm_nagle_test`, onde é realmente implementado o algoritmo de Nagle.

- `tcpxm_nagle_test`: implementa o algoritmo de Nagle para verificar se o `sk_buff` passado como parâmetro pode ser enviado. Esta função realiza validações relacionadas ao tipo de pacote que está sendo transmitido e utiliza o teste da função `tcpxm_nagle_check` para verificar se o segmento não viola as regras do algoritmo de Nagle [Nagle 1984].

- `tcpxm_min_win_test`: verifica se o número de seqüência do segmento que será transmitido é o menor dentre todas as estruturas `sock` de todos os receptores. Este teste serve para garantir a sincronização das janelas de transmissão e de congestionamento unicast de todos os receptores conforme definido em [Jeacle 2005].

- `tcpxm_snd_wnd_test`: verifica se o primeiro segmento do `sk_buff` passado como parâmetro cabe na janela de transmissão contida na estrutura `tcpxm_sock` também passada como parâmetro.

- `tcpxm_snd_test`: verifica se o primeiro `sk_buff` da fila de transmissão da estrutura `sock` passada como parâmetro deve ser transmitido.

- `tcpxm_skb_is_last`: verifica se o `sk_buff` passado como parâmetro é o último da fila de transmissão `sk_write_queue` contida na estrutura `sock` passada como parâmetro.

- `tcpxm_may_send_now`: inicia a validação para o envio do primeiro `sk_buff` da fila de transmissão `sk_write_queue`, utilizando para isto as validações realizadas pelas funções `tcpxm_min_wnd_test`, `tcpxm_snd_test`, `tcpxm_nagle_test` e `tcpxm_cwnd_test`.

- `tcpxm_tso_fragment`: trunca a estrutura `sk_buff` passada como parâmetro em `len` bytes e coloca o restante dos dados em novas estruturas `sk_buff`. Esta fragmentação é realizada inteiramente pela interface de rede caso a mesma possua suporte ao mecanismo de TSO. Os buffers criados são inseridos na fila *sk_write_queue*.

- `tcpxm_tso_should_defer`: verifica a existência de espaço livre na estrutura `sk_buff` passada como parâmetro para o encapsulamento de mais dados. Quanto maior a quantidade de dados encapsulados, menor será a exigência de fragmentação dos segmentos, gerando um possível aumento no *throughput* da rede.

- `tcpxm_write_xmit`: transmite o primeiro `sk_buff` da fila de transmissão `sk_write_queue` para a função `tcpxm_transmit_skb`, que efetivamente transmite o pacote para a camada de rede. Esta função é chamada conforme os pacotes já transmitidos são confirmados pelo receptor.

- `__tcpxm_push_pending_frames`: inicia a transmissão do primeiro `sk_buff` da fila de transmissão `sk_write_queue` através da função `tcpxm_write_xmit`.

- `tcpxm_push_one`: envia somente o primeiro `sk_buff` da fila de transmissão `sk_write_queue`. Pelo fato de enviar somente o primeiro `sk_buff` esta função realiza todas as validações necessárias e chama diretamente a função `tcpxm_transmit_skb` para transmissão deste `sk_buff`.

- `__tcpxm_select_window`: retorna a quantidade de espaço em bytes que a janela de transmissão passada como parâmetro pode ser aumentada.

- `tcpxm_retrans_try_collapse`: une dois `sk_buffs` contíguos durante a retransmissão de segmentos.

- `tcpxm_simple_retransmit`: retransmite um `sk_buff` sem utilizar o mecanismo de `backoff` implementado no arquivo `tcpxm_timer`.

- `tcpxm_retransmit_skb`: retransmite um `sk_buff` utilizando o mecanismo de *backoff* implementado no arquivo `tcpxm_timer`.

- `tcpxm_xmit_retransmit_queue`: função chamada após a confirmação de recepção do primeiro segmento retransmitido. Esta função tenta retransmitir o restante dos `sk_buffs` contidos na fila de retransmissão. A função utilizada para executar a retransmissão dos pacotes é a `tcpxm_retransmit_skb`.

- `tcpxm_send_fin`: aloca um `sk_buff` e cria um segmento FIN, colocando o pacote criado na fila de transmissão através da função `tcpxm_queue_skb`.

- `tcpxm_send_active_reset`: aloca um `sk_buff` e cria um segmento RST, transmitindo-o diretamente através da função `tcpxm_transmit_skb`.

- `tcpxm_send_synack`: transmite um segmento SYN-ACK utilizando a função `tcpxm_make_synack` para criação do pacote e a função `tcpxm_transmit_skb` para transmissão do mesmo.

- `tcpxm_make_synack`: aloca e cria um segmento SYN-ACK como resposta a uma requisição de conexão (segmento SYN). É no segmento SYN-ACK que o receptor informa ao transmissor o grupo multicast que deve ser utilizado durante a sessão, caso IP multicast nativo esteja disponível na rede.

- `tcpxm_connect_init`: inicializa os valores da estrutura `tcpxm_sock` antes que um segmento SYN seja enviado iniciando o 3-way handshake.

- `tcpxm_connect`: aloca um `sk_buff` e cria um segmento SYN transmitindo-o diretamente através da função `tcpxm_transmit_skb`, iniciando o 3-way handshake. É no segmento SYN que o transmissor informa ao receptor o endereço do grupo multicast que pode ser utilizado durante a sessão.

- `tcpxm_send_delayed_ack`: envia *delayed ack's*. Este procedimento inicializa o temporizador e utiliza a função `tcpxm_send_ack` para criar e enviar o segmento ACK.

- `tcpxm_calc_msegrcvper`: calcula a quantidade de segmentos recebidos via IP multicast. Este valor é enviado ao transmissor encapsulado nos segmentos de confirmação (ACK) para que o mesmo possa decidir a tecnologia que deve ser utilizada na transmissão dos segmentos de dados.

- `tcpxm_send_ack`: aloca um `sk_buff` e cria um segmento ACK, transmitindo-o diretamente através da função `tcpxm_transmit_skb`.

- `tcpxm_xmit_probe_skb`: envia um segmento TCP-XM com um identificador fora da seqüência, esperando pela confirmação (ACK) da recepção do pacote.
- `tcpxm_write_wakeup`: verifica a existência de alguma estrutura `sk_buff` pronta para transmissão. Caso exista o primeiro `sk_buff` da fila de transmissão é transmitido.
- `tcpxm_send_probe0`: realiza o processamento necessário após a expiração do temporizador de controle da janela de transmissão. Caso necessário um segmento parcial é enviado indicando a expiração deste temporizador à outra extremidade da conexão.

A listagem do código fonte do arquivo `tcpxm_output.c` e maiores detalhes sobre a implementação (relatados nos comentários) podem ser encontrados em [Menegotto e Barcellos 2006].

4.3.4 net/ipv4/tcpxm_ipv4.c

Neste arquivo estão declaradas estruturas do protocolo TCP-XM referentes ao estabelecimento da sessão em redes IPv4. Dentre elas pode-se citar:

- `tcpxm_hashinfo`: tabela hash utilizada para armazenar informações do protocolo TCP-XM acessadas freqüentemente. A implementação desta tabela hash é realizada utilizando a própria API do kernel.
- `tcpxm_request_sock_ops`: armazena ponteiros para funções utilizadas durante o estabelecimento de uma sessão TCP-XM após a recepção de um segmento SYN.
- `tcpxm_ipv4_specific`: armazena ponteiros para funções relacionadas ao estabelecimento de uma sessão TCP-XM antes da recepção de um segmento SYN.
- `tcpxm4_seq_afinfo`: armazena ponteiros para todas as estruturas necessárias na declaração de uma entrada no sistema de arquivos *procfs*. Esta estrutura é utilizada diretamente na chamada `tcpxm_proc_init`.
- `tcpxm_prot`: estrutura `proto` que define as funções que responderão pelos eventos (*callbacks*) do protocolo TCP-XM. É utilizada durante a inicialização do protocolo TCP-XM.

Neste arquivo é declarada a variável `sysctl_tcpxm_default_mcgroup` que armazena o endereço do grupo multicast padrão do protocolo TCP-XM (224.0.1.5), utilizado caso a aplicação não selecione nenhum endereço IP multicast e exista suporte a IP multicast na rede. Este valor pode ser alterado diretamente do contexto de usuário através do mecanismo *sysfs*.

Estão implementadas também procedimentos e funções utilizadas no estabelecimento de uma sessão TCP-XM e na recepção de segmentos. São elas:

- `tcpxm_v4_get_port`: aloca uma porta local na sessão passada como parâmetro através da chamada `inet_csk_get_port`.
- `tcpxm_v4_hash`: insere a estrutura `sock` passada como parâmetro na tabela hash `tcpxm_hashinfo`.

- `tcpxm_unhash`: remove a estrutura `sock` passada como parâmetro da tabela hash de conexões `tcpxm_hashinfo`.

- `tcpxm_v4_init_sequence`: retorna um número de seqüência inicial para a sessão representada pela estrutura `sock` passada como parâmetro. Este é o número de seqüência utilizado no envio do segmento SYN durante o estabelecimento da conexão unicast entre transmissor e receptor.

- `__tcpxm_v4_check_established`: verifica se a estrutura `sock` passada como parâmetro está no estado TCPXM_ESTABLISHED através da pesquisa na tabela hash `tcpxm_hashinfo`.

- `tcpxm_connect_port_offset`: aloca um número de porta local livre através o uso do algoritmo de MD4. Maiores informações sobre o algoritmo de MD4 podem ser obtidas em [Rivest 1992].

- `tcpxm_v4_hash_connect`: associa uma porta livre à conexão representada pela estrutura `sock` passada como parâmetro. Esta função já insere a porta alocada na tabela hash de portas utilizadas.

- `__tcpxm_v4_connect`: inicia uma conexão unicast para o endereço IP/porta passado como parâmetro através de uma estrutura `sockaddr_in`. Esta função é utilizada pela função `tcpxm_v4_connect` para conectar-se ao endereço unicast do receptor passado como parâmetro.

- `tcpxm_v4_addr2sockaddr`: copia o endereço unicast representado pela estrutura `sockaddr_in` para a estrutura `inet_sock` contida na estrutura `sock` passada como parâmetro.

- `tcpxm_v4_connect`: inicia o 3-way handshake realizado entre transmissor e receptores. Esta função percorre a lista encadeada `receivers` iniciando uma conexão unicast com cada receptor. A estrutura `sock` utilizada em cada conexão é alocada no kernel e armazenada dentro da lista encadeada `receivers`.

- `tcpxm_do_pmtu_discovery`: descobre qual o *Path MTU* do link utilizado em uma sessão TCP-XM. O MTU é o limite superior de bytes que podem ser encapsulados em um pacote Ethernet 802.2 e Ethernet 802.3. O *Path MTU* é o menor MTU encontrado no caminho entre transmissor e receptor. Pacotes que possuem tamanho em bytes maior que o MTU (1492 bytes em redes Ethernet 802.2 e 802.3) são fragmentados pela camada IP durante o processo de transmissão. O algoritmo de cálculo do *Path MTU* de uma conexão está descrito em detalhes em [Mogul e Deering 1990].

- `tcpxm_v4_err`: trata condições de erro obtidas durante operações realizadas pelo módulo ICMP a serviço do protocolo TCP-XM. A descoberta do *Path MTU* é um exemplo de operação deste gênero em que o RTT de um pacote ICMP é utilizado para estivamtiva da latência da rede.

- `tcpxm_v4_send_check`: verifica a integridade dos pacotes recebidos através do cálculo da soma de verificação da camada de transporte contido no pacote.

- `tcpxm_v4_send_reset`: inicia a transmissão de um segmento RST para um transmissor para um receptor devido a problemas na conexão. A transmissão de um segmento RST causará a desconexão de todos os demais receptores e a finalização da sessão TCP-XM para garantir a igualdade dos dados transmitidos.

- `tcpxm_v4_send_ack`: transmite um segmento ACK.

- `tcpxm_v4_timewait_ack`: inicia o envio de um segmento ACK quando um socket TCP-XM está no estado TCPXM_TIME_WAIT. Este procedimento utiliza a função `tcpxm_v4_send_ack` para transmissão do pacote.

- `tcpxm_v4_reqsk_send_ack`: inicia o envio de um segmento ACK para a estrutura `request_sock` passada como parâmetro. A estrutura `request_sock` representa uma futura conexão TCP-XM unicast que ainda não foi estabelecida. A estrutura `request_sock` está contida na estrutura `inet_request_sock` e armazena os dados para controle da sessão. A estrutura `request_sock` é alocada durante a recepção de um segmento SYN pelo receptor.

- `tcpxm_v4_send_synack`: envia um segmento SYN-ACK após a recepção de um segmento ACK durante o 3-way handshake. O segmento SYN-ACK enviado contém o endereço do grupo multicast escolhido pelo receptor no caso de conectividade IP Multicast nativo.

- `tcpxm_v4_reqsk_destructor`: libera a estrutura `request_sock` utilizada durante o 3-way handshake e passada como parâmetro. É utilizada quando a conexão é estabelecida e a estrutura `sock` é criada representando a conexão estabelecida.

- `tcpxm_syn_flood_warning`: emite um alerta no caso de um ataque *SYN Flood* a uma das interfaces de rede utilizando o protocolo TCP-XM. Um ataque *SYN Flood* caracteriza-se pela recepção excessiva de requisições de conexão que não completam-se. Estas requisições gastam recursos do host como memória, cache e tempo de processamento deixando milhares de conexões no estado *Half Open*.

- `tcpxm_v4_save_options`: copia o campo de opções do cabeçalho IP de um segmento recebido para a estrutura `inet_request_sock` que representa uma conexão no estado *Half Open*.

- `tcpxm_v4_conn_request`: recebe um segmento SYN que inicia o 3-way handshake entre transmissor e receptor. O segmento SYN deve conter o endereço do grupo multicast que o transmissor deseja utilizar durante a sessão caso IP Multicast nativo esteja presente na rede. O receptor confirma o grupo multicast no segmento SYN-ACK caso exista suporte a IP Multicast nativo na rede.

- `tcpxm_v4_syn_recv_sock`: transforma a estrutura de dados que representa uma conexão. Até a finalização do 3-way handshake a estrutura chama-se `inet_request_sock`. A recepção do ACK final do 3-way handshake causa a conversão desta estrutura para uma estrutura `sock` que representa a conexão no estado TCPXM_ESTABLISHED.

- `tcpxm_v4_hnd_req`: procura por requisições de conexão realizadas pelo transmissor do `sk_buff` passado como parâmetro. Esta função é chamada pela função `tcpxm_v4_do_rcv` para que não existam requisições de conexão duplicadas.

- `tcpxm_v4_checksum_init`: verifica a necessidade de cálculo da soma de verificação do `sk_buff` passado como parâmetro. Caso necessário a soma de verificação é calculada pela função `csum_tcpudp_nofold` localizada no arquivo `include/asm/checksum.c`. O cálculo da soma da verificação de um pacote é dependente de arquitetura portanto existem diversas implementações desta função nas sub-árvores de arquiteturas do kernel GNU/Linux (`arch/m68k`, `arch/alpha`, `arch/powerpc`, etc...).

- `tcpxm_v4_do_rcv`: implementa a recepção de segmentos TCP-XM chamando as funções de tratamento de pacote conforme o estado da sessão. No caso de uma conexão estabelecida a função `tcpxm_rcv_established` é utilizada. No caso de uma requisição de conexão uma nova estrutura `inet_request_sock` é criada pela função `tcpxm_child_process`. Nos demais casos o tratador chamado é a função `tcpxm_rcv_state_process`.

- `tcpxm_v4_rcv`: inicia o processo de recepção de um segmento TCP-XM na camada de transporte. Ela é a responsável pelas verificações iniciais e através da função `tcpxm_v4_do_rcv` encaminha devidamente o `sk_buff` passado como parâmetro para processamento. É esta função que diferencia o tipo do `sk_buff` durante sua recepção, preenchendo a variável `ismcast` conforme o meio de transmissão utilizado (multicast ou unicast) e escolhendo a estrutura `sock` destino do `sk_buff`.

- `tcpxm_v4_remember_stamp`: armazena o timestamp do último segmento de dados recebido pela estrutura `sock` passada como parâmetro caso a sessão representada não esteja no estado TCPXM_TIME_WAIT. Este timestamp é utilizado na detecção de segmentos duplicados.

- `tcpxm_v4_tw_remember_stamp`: armazena o timestamp do último segmento de dados recebido pela estrutura `sock` passada como parâmetro caso a sessão esteja no estado TCPXM_TIME_WAIT. Este timestamp é utilizado na detecção de segmentos duplicados.

- `tcpxm_v4_init_sock`: implementa a inicialização da estrutura `sock` passada como parâmetro.

- `tcpxm_v4_destroy_sock`: implementa a destruição da estrutura `sock` passada como parâmetro da memória.

- `txmw_head`: retorna o primeiro nodo da lista encadeada de sockets do protocolo TCP-XM no estado `TCPXM_TIME_WAIT`.

- `txmw_next`: retorna o próximo nodo a partir do parâmetro `cur` armazenado na lista encadeada de sockets do protocolo TCP-XM no estado `TCPXM_TIME_WAIT`.

- `tcpxm_listening_get_next`: consulta a tabela hash `listening_hash` e retorna o próximo socket do protocolo TCP-XM no estado `TCPXM_LISTEN` a partir do nodo `cur`.

- `tcpxm_listening_get_idx`: consulta a tabela hash `listening_hash` e retorna a entrada armazenada na posição passada pelo parâmetro `pos`.

- `tcpxm_established_get_first`: retorna o primeiro nodo armazenado na tabela hash `ehash` no estado `TCPXM_ESTABLISHED`.

- `tcpxm_established_get_next`: consulta a tabela hash `ehash` e retorna o próximo socket do protocolo TCP-XM que está no estado `TCPXM_ESTABLISHED` a partir do nodo `cur`.

- `tcpxm_established_get_idx`: consulta a tabela hash `ehash` e retorna a entrada armazenada na posição passada pelo parâmetro `pos`.

- `tcpxm_get_idx`: consulta as tabelas hash `ehash` e `listening_hash` utilizando as funções `tcpxm_listening_get_idx` e `tcpxm_established_get_idx` a partir da posição `pos`.

- `tcpxm_seq_start`: consulta as tabelas hash `ehash` e `listening_hash` utilizando o iterador `seq_file` passado como parâmetro a partir da posição `pos` passada como parâmetro.

- `tcpxm_seq_next`: retorna a próxima entrada das tabelas hash `ehash` e `listening_hash` utilizando um iterador contido na estrutura `seq_file` passada como parâmetro. Este iterador permite percorrer os objetos de uma entrada `procfs` de maneira simplificada.

- `tcpxm_seq_stop`: desabilita os mecanismos de lock utilizados para percorrer as tabelas hash do protocolo TCP-XM.

- `tcpxm_seq_open`: cria e inicializa o iterador `seq_file` utilizado para percorrer as tabelas hash do protocolo TCP-XM.

- `tcpxm_proc_register`: cria uma nova entrada no sistema de arquivos virtual `profcs` associada ao protocolo TCP-XM. A arquitetura de parâmetros da entrada é definida pela estrutura `tcpxm_seq_afinfo`. Esta estrutura armazena ponteiros para funções de tratamento de eventos do sistema de arquivos virtual (leitura, escrita, etc...).

- `tcpxm_proc_unregister`: remove a entrada associada ao protocolo TCP-XM passada como parâmetro do sistema de arquivos virtual `procfs`.

- `tcpxm_v4_get_openreq`: insere na variável local `tmpbuf` informações sobre a estrutura `request_sock` passada como parâmetro. Esta procedimento é utilizado na formatação das informações sobre sockets TCP-XM no estado `TCPXM_SYN_RECV` retornadas pela função `tcpxm_v4_seq_show`.

- `tcpxm_v4_get_sock`: insere na variável local `tmpbuf` informações sobre a estrutura `sock` passada como parâmetro. Este procedimento é utilizado na formatação das informações sobre sockets TCP-XM no estado `TCPXM_ESTABLISHED` retornadas pela função `tcpxm_v4_seq_show`.

- `tcpxm_v4_get_timewait_sock`: insere na variável local `tmpbuf` informações sobre a estrutura `inet_timewait_sock` passada como parâmetro. Este procedimento é utilizado na formatação das informações sobre sockets TCP-XM no estado `TCPXM_TIME_WAIT` retornadas pela função `tcpxm_v4_seq_show`.
- `tcpxm_v4_seq_show`: copia para o contexto de usuário informações sobre os sockets no estado `TCPXM_SYN_RECV`, `TCPXM_ESTABLISHED` e `TCPXM_TIME_WAIT` utilizando as funções `tcpxm_v4_get_openreq`, `tcpxm_v4_get_tcpxm_sock` e `tcpxm_v4_get_timewait_sock`.
- `tcpxm_proc_init`: inicializa as entradas TCP-XM no sistema de arquivos virtual `procfs` utilizando a função `tcpxm_proc_register`.
- `tcpxm_proc_exit`: remove as entradas TCP-XM do sistema de arquivos virtual `procfs` através da função `tcpxm_proc_unregister`.
- `tcpxm_v4_init`: cria o socket de controle do protocolo TCP-XM na pilha TCP/IP do kernel GNU/Linux. Este socket é utilizado durante a inicialização do protocolo TCP-XM.

A listagem do código fonte do arquivo `tcpxm_ipv4.c` e maiores detalhes sobre a implementação deste arquivo podem ser encontrados em [Menegotto e Barcellos 2006].

4.3.5 net/ipv4/tcpxm_minisocks.c

Este arquivo contém a declaração da tabela hash `tcpxm_death_row`, utilizada para agendamento da destruição de sockets no estado `TCPXM_TIME_WAIT`.

Este arquivo contém também a implementação de funções e procedimentos para manipulação de sockets no estado `TCPXM_TIME_WAIT` e para requisições de conexões durante o estabelecimento da sessão. São elas:

- `tcpxm_in_window`: verifica se os números de seqüência passados como parâmetro estão dentro da janela de transmissão passada como parâmetro.
- `tcpxm_timewait_state_process`: implementa o processamento necessário para recepção de segmentos em uma sessão TCP-XM no estado `TCPXM_TIME_WAIT`.
- `tcpxm_time_wait`: altera o estado da conexão representada pela estrutura `sock` passada como parâmetro para `TCPXM_TIME_WAIT`.
- `tcpxm_create_openreq_child`: retorna uma estrutura `sock` filha a partir da estrutura `request_sock` passada como parâmetro. Esta função é utilizada para iniciar a recepção de dados ao final do 3-way handshake.
- `tcpxm_check_req`: implementa a recepção de pacotes em sessões que estão no estado `TCPXM_SYN_RECV` e são representadas por uma estrutura `request_sock`.
- `tcpxm_child_process`: insere um segmento recebido nas estruturas de dados do socket filho criado para recepção dos dados, caso ativo.

A listagem do código fonte do arquivo `tcpxm_minisocks.c` e maiores detalhes sobre a implementação deste arquivo podem ser encontrados em [Menegotto e Barcellos 2006].

4.3.6 net/ipv4/tcpxm_timer.c

Neste arquivo estão declaradas variáveis que controlam o comportamento do protocolo em relação às tentativas de retransmissão de pacotes. Dentre estas variáveis pode-se citar:

- `sysctl_tcpxm_syn_retries`: número máximo de segmentos SYN enviados durante tentativa de estabelecimento de uma sessão.
- `sysctl_tcpxm_keepalive_intvl`: número máximo de segundos de inatividade de uma sessão TCP-XM antes do envio de mensagens *keep-alive*.
- `sysctl_tcpxm_synack_retries`: número máximo de segmentos SYN-ACK enviados durante o 3-way handshake.

Este arquivo contém também procedimentos e funções e utilizadas para criação e manipulação de temporizadores do protocolo TCP-XM. São elas:

- `tcpxm_init_xmit_timers`: inicializa os temporizadores utilizados pelo transmissor na estrutura `sock` passada como parâmetro.
- `tcpxm_write_err`: indica na estrutura `sock` passada como parâmetro a ocorrência de um timeout (expiração do tempo-limite).
- `tcpxm_out_of_resources`: desabilita o consumo de recursos por sockets no estado `TCPXM_TIME_WAIT` caso o limite máximo de sockets órfãos seja alcançado ou caso o protocolo estiver no estado de `memory pressure`.
- `tcpxm_orphan_retries`: calcula o número de tentativas de comunicação com sockets no estado `TCPXM_TIME_WAIT` antes de sua destruição.
- `tcpxm_write_timeout`: realiza o processamento após a expiração do temporizador de transmissão de segmentos de dados.
- `tcpxm_delack_timer`: implementa o temporizador utilizado no mecanismo *delayed ACK*.
- `tcpxm_probe_timer`: implementa o temporizador utilizado durante as tentativas de retransmissão de um pacote conforme definido pela variável `sysctl_tcpxm_retries2`.
- `tcpxm_retransmit_timer`: implementa o temporizador utilizado no mecanismo de retransmissão de segmentos de dados.
- `tcpxm_write_timer`: implementa o temporizador utilizado na transmissão de segmentos.
- `tcpxm_synack_timer`: implementa o temporizador utilizado na espera de um segmento SYN-ACK no 3-way handshake.
- `tcpxm_set_keepalive`: configura o tempo máximo em segundos de espera a respostas de mensagens `keepalive`.

- `tcpxm_keepalive_timer`: implementa o temporizador utilizado na espera de respostas à mensagens `keepalive`.

A listagem do código fonte do arquivo `tcpxm_timer.c` e maiores detalhes sobre a implementação deste arquivo podem ser encontrados em [Menegotto e Barcellos 2006].

4.4 Parâmetros de uma Sessão TCP-XM

Uma sessão TCP-XM executada no kernel GNU/Linux pode ser configurada através de dois mecanismos. Ambos permitem a configuração do comportamento do protocolo através do contexto de usuário em tempo de execução. O primeiro mecanismo é a chamada de sistema `sysctl` e o segundo mecanismo é a chamada de sistema `setsockopt`. Ambos estão descritos nas Subseções 4.4.1 e 4.4.2.

4.4.1 Sysctl

A chamada de sistemas `sysctl` permite a configuração do comportamento do protocolo TCP-XM de maneira global, ou seja, a mudança afeta todas as sessões TCP-XM ativas. Esta alteração pode ser executada pelo própria aplicação (através da chamada de sistema) ou através de arquivos localizados em `/proc/net/ipv4/tcpxm`. Cada arquivo do diretório `/proc/net/ipv4/tcpxm` representa um parâmetro. A configuração do parâmetro é feita através da escrita do valor desejado no arquivo correspondente. São eles:

- `sysctl_tcpxm_timestamps`: habilita o envio do timestamp no cabeçalho de pacotes do protocolo TCP-XM conforme descrito em [3]. O timestamp é utilizado para melhora da estimativa do RTT de um link (RTTM) e para implementação de um algoritmo conhecido como PAWS, que descarta pacotes duplicados. Ambos algoritmos estão descritos em [3].

- `sysctl_tcpxm_window_scaling`: habilita a negociação do tamanho da janela de transmissão durante o 3-way handshake conforme descrito em [3]. Quando habilitada permite a utilização de janelas de transmissão maiores que 64 Kb, permitindo à aplicação o aumento do tamanho do buffer de transmissão.

- `sysctl_tcpxm_sack`: habilita o mecanismo de SACK conforme descrito em [Mathis et al. 1996].

- `sysctl_tcpxm_fin_timeout`: determina o tempo máximo em segundos que um socket TCP-XM aguarda pelo último segmento FIN antes de seu encerramento.

- `sysctl_tcpxm_keepalive_time`: determina o tempo em segundos que uma conexão pode ficar inativa antes que mensagens *keepalive* sejam enviadas.

- `sysctl_tcpxm_keepalive_probes`: determina o número máximo de mensagens *keepalive* enviadas antes do encerramento do socket inativo.

- `sysctl_tcpxm_keepalive_intvl`: determina o intervalo de tempo em segundos entre o envio de mensagens *keepalive* em sockets inativos.

- `sysctl_tcpxm_syn_retries`: determina o número máximo de segmentos SYN retransmitidos em uma tentativa de estabelecimento de conexão.

- `sysctl_tcpxm_synack_retries`: determina o número máximo de segmentos SYN-ACK retransmitidos em uma tentativa de estabelecimento de conexão.

- `sysctl_tcpxm_retries1`: determina o número de tentativas de retransmissão de segmentos de dados em sessões que estão no estado TCPXM_ESTABLISHED. Uma tentativa de retransmissão é composta de diversas retransmissões de pacotes.

- `sysctl_tcpxm_retries2`: determina o número de retransmissões de um segmento de dados em uma tentativa de retransmissão.

- `sysctl_tcpxm_orphan_retries`: determina o número máximo de tentativas de sinalização aos receptores quando uma sessão for encerrada pelo transmissor.

- `sysctl_tcpxm_syncookies`: habilita o envio de *syncookies* quando esgotar o espaço livre da fila de recepção `sk_backlog`. Esta opção serve para evitar ataques de negação de serviço utilizando a técnica de *synflood*.

- `sysctl_tcpxm_retrans_collapse`: habilita a junção de segmentos de dados que estão na fila de retransmissão.

- `sysctl_tcpxm_stdurg`: habilita a interpretação de segmentos URG conforme descrito em [DARPA 1981]. Segundo [DARPA 1981] deve ser utilizado um ponteiro para o final da área de dados do segmentos URG, ao contrário do padrão do kernel, que utiliza um ponteiro para o início da área de dados.

- `sysctl_tcpxm_rfc1337`: desabilita o encerramento automático de sessões TCP-XM no estado `TCPXM_TIME_WAIT` quando um segmento RST é recebido.

- `sysctl_tcpxm_abort_on_overflow`: habilita o encerramento automático de conexões em que a aplicação está ocupada e não consegue processar os segmentos recebidos. Esta opção é desabilitada por padrão.

- `sysctl_tcpxm_max_orphans`: determina o número máximo de sockets órfãos do protocolo TCP-XM (sockets que não estão associados a nenhum descritor no contexto de usuário) .

- `sysctl_tcpxm_fack`: habilita o uso do mecanismo FACK em sessões TCP-XM. O mecanismo FACK melhora a detecção de segmentos perdidos através de variáveis do mecanismo SACK. Esta opção é desabilitada automaticamente em situações de excesso de reordenamento de segmentos.

- `sysctl_tcpxm_reordering`: determina o número máximo de reordenamentos aplicados a um único segmento antes que a retransmissão seja solicitada.

- `sysctl_tcpxm_ecn`: habilita o mecanismo de sinalização explícita de congestionamento (ECN) no protocolo TCP-XM. Este mecanismo insere uma flag indicando congestionamento do link de transmissão dentro do cabeçalho do protocolo IP conforme descrito em [9].

- `sysctl_tcpxm_dsack`: habilita o mecanismo de *Duplicate SACK* conforme descrito em [Floyd et al. 2000; Zhang et al. 2002]. O mecanismo de *Duplicate*

SACK extende o mecanismo de SACK descrito em [Mathis et al. 1996] para retransmissão de segmentos de dados de forma que um transmissor possa inferir a ordem de recepção dos mesmos, evitando o envio de segmentos duplicados. Maiores informações sobre este algoritmo podem ser encontradas em [Floyd et al. 2000; Zhang et al. 2002].

- `sysctl_tcpxm_mem[3]`: vetor de três inteiros que define o comportamento do protocolo TCP-XM em relação à alocação de memória. A primeira posição do vetor determina o limite mínimo de memória alocada, em que o protocolo não empresa nenhum mecanismo de economia de memória. A segunda posição do vetor determina o limite de memória utilizada pelo protocolo antes que o mesmo entre num estado de *memory pressure*, em que a alocação de memória começa a ser controlada. A terceira posição do vetor determina o limite máximo de memória que pode ser alocado pelo protocolo TCP-XM.

- `sysctl_tcpxm_wmem[3]`: vetor de três inteiros que define o comportamento dos buffers de transmissão do protocolo TCP-XM em relação à alocação de memória. A primeira posição do vetor define o tamanho mínimo do buffer de transmissão alocado para um socket TCP-XM. A segunda posição do vetor define o tamanho padrão alocado para o buffer de transmissão de um socket TCP-XM. A terceira posição do vetor define o tamanho máximo alocado para o buffer de transmissão de um socket TCP-XM.

- `sysctl_tcpxm_rmem[3]`: vetor de três inteiros que define o comportamento dos buffers de recepção do protocolo TCP-XM em relação à alocação de memória. A primeira posição do vetor define o tamanho mínimo do buffer de recepção alocado para um socket TCP-XM. A segunda posição do vetor define o tamanho padrão alocado para o buffer de recepção de um socket TCP-XM. A terceira posição do vetor define o tamanho máximo alocado para o buffer de recepção de um socket TCP-XM.

- `sysctl_tcpxm_app_win`: define quantos bytes da janela de transmissão devem ser reservados para eventuais sobrecargas ocorridas em uma sessão TCP-XM.

- `sysctl_tcpxm_adv_win_scale`: determina a distribuição de memória entre o buffer de recepção da aplicação e a janela de transmissão.

- `sysctl_tcpxm_tw_reuse`: habilita a reutilização de sockets no estado `TCPXM_TIME_WAIT` para novas conexões quando esta operação é segura do ponto de vista do protocolo.

- `sysctl_tcpxm_frto`: habilita o mecanismo F-RTO conforme descrito em [Sarolahti e Kojo 2005; 10]. O mecanismo de F-RTO melhora o processo de retransmissão de pacotes do protocolo TCP-XM em links aonde ocorrem interrupções abruptas de conectividade através do gerenciamento dos temporizadores utilizados para retransmissões. Maiores detalhes sobre o algoritmo F-RTO podem ser obtidos em [Sarolahti e Kojo 2005; 10].

- `sysctl_tcpxm_low_latency`: variável que sinaliza o protocolo TCP-XM para favorecer a baixa latência da rede ao invés do alto *throughput* em suas decisões.

- `sysctl_tcpxm_nometrics_save`: desabilita a gravação de métricas como RTT e Slow Start Threshold durante o fechamento de sessões TCP-XM.

- `sysctl_tcpxm_moderate_rcvbuf`: sinaliza ao kernel GNU/Linux que o buffer de recepção de pacotes da estrutura *sock* passada como parâmetro deve ser truncado conforme a segunda posição do vetor `tcpxm_rmem`. Esta situação é desejável quando o protocolo TCP-XM está no modo de *memory pressure*.

- `sysctl_tcpxm_tso_win_divisor`: variável que armazena o percentual da janela de congestionamento que pode ser utilizado para transmissão de um único segmento através do mecanismo TSO.

- `sysctl_tcpxm_abc`: habilita o mecanismo de *Appropriate Byte Count* conforme definido em [Allman 2003]. Este mecanismo modifica o comportamento do protocolo em relação ao aumento do tamanho da janela de transmissão. Caso habilitada a janela de transmissão será expandida conforme o número de confirmações existentes dentro de um segmento ACK. Maiores informações sobre o mecanismo de *Appropriate Byte Count* podem ser obtidas em [Allman 2003].

- `sysctl_tcpxm_default_state`: força uma tecnologia de transmissão para todas as sessões TCP-XM. Utilizando esta opção é possível forçar o uso de IP Multicast nativo (1), forçar a emulação de multicast sobre unicast (0) ou deixar o comportamento padrão do protocolo (2), onde a melhor tecnologia disponível é utilizada. A opção 2 é a padrão.

- `sysctl_tcpxm_default_mcgroup`: define o endereço do grupo multicast padrão para todas as sessões TCP-XM. O endereço padrão é 224.0.1.5. Este endereço é utilizado caso nenhum endereço for especificado pela aplicação e alguma forma de transmissão multicast seja realizada. O endereço deve ser informado no formato *network byte order*.

- `sysctl_tcpxm_delay_after_ff`: define o tempo máximo em segundos que o protocolo espera após realizar um *fall forward*.

4.4.2 Setsockopt / Getsockopt

A chamada *setsockopt* é uma chamada padrão em implementações da abstração de sockets BSD. Ela permite a configuração de parâmetros de uma sessão de comunicação em tempo de execução. Os parãmetros da chamada `setsockopt` definem a estrutura `sock`, o parâmetro e o valor que deve ser atribuído ao parâmetro.

A verificação do valor atribuído a determinado parâmetro pode ser feita com a chamada `getsockopt`. Os parâmetros da chamada `getsockopt` definem a estrutura `sock` e o parãmetro que devem ser lidos.

Os parâmetros a seguir podem ser manipulados através de chamadas `setsockopt`/`getsockopt`:

- `TCPXM_RCVADD`: adiciona o endereço unicast de um receptor na lista encadeada `receivers`. O parâmetro desta função é uma estrutura `sockaddr_in` que contém o endereço, a porta e a família de endereços utilizada. Esta chamada é obrigatória em sessões TCP-XM que emulam multicast sobre unicast, pois é mandatório que o transmissor conheça o endereço unicast dos receptores.

- `TCPXM_CONGESTION`: define o mecanismo de controle de congestionamento utilizado em uma sessão TCP-XM.
- `TCPXM_MAXSEG`: define o tamanho máximo de segmento (MSS) utilizado em uma sessão TCP-XM.
- `TCPXM_NODELAY`: habilita o uso do algoritmo de *Nagle* em uma sessão TCP-XM. O algoritmo de Nagle [Stevens 1994; Comer 1998; Nagle 1984] reduz a quantidade de segmentos transmitidos através da junção do conteúdo de segmentos de tamanho pequeno em um único de tamanho apropriado.
- `TCPXM_CORK`: define se segmentos parciais podem ser transmitidos em uma sessão TCP-XM. Esta opção é mutuamente exclusiva em relação a opção `TCPXM_NODELAY`.
- `TCPXM_KEEPIDLE`: define o tempo máximo em segundos que uma conexão pode permanecer inoperante antes que mensagens de *keep-alive* sejam enviadas.
- `TCPXM_KEEPINTVL`: define o intervalo em segundos entre o envio de mensagens *keep-alive*.
- `TCPXM_KEEPCNT`: define o número máximo de mensagens *keep-alive* que podem ser transmitidas antes do encerramento da conexão.
- `TCPXM_SYNCNT`: define o número máximo de retransmissões de segmentos SYN numa tentativa de conexão.
- `TCPXM_LINGER2`: define o tempo máximo em segundos que um socket permanece no estado `TCPXM_FIN_WAIT2` antes do seu encerramento.
- `TCPXM_DEFER_ACCEPT`: define se o kernel vai esperar o primeiro segmento de dados chegar antes de acordar o processo receptor.
- `TCPXM_WINDOW_CLAMP`: define o tamanho máximo da janela de transmissão utilizada em uma sessão TCP-XM.
- `TCPXM_QUICKACK`: habilita o uso do modo *quick ack* em sessões TCP-XM. No modo *quick ack* cada segmento de dados recebido é imediatamente confirmado através do envio de um pacotes ACK. No modo de confirmação padrão é realizada confirmação via SACK ou *delayed ACK*.
- `TCPXM_RCVDEL`: remove um endereço unicast da lista encadeada *receivers*. O encerramento de uma sessão TCP-XM implica na remoção automática de todos os endereços unicast da lista *receivers* associados à sessão.
- `TCPXM_OPTMCGROUP`: especifica o endereço do grupo multicast utilizado numa sessão TCP-XM. O endereço passado como parâmetro deve estar no formato *network byte order* (representa um endereço IP através de um inteiro de 32 bits).

5 AVALIAÇÃO

A avaliação da implementação de um protocolo multicast confiável no subsistema de rede do kernel de um sistema operacional é uma tarefa dispendiosa. Para que uma implementação seja devidamente testada, devem ser utilizadas diferentes topologias de rede (LAN, MAN, WAN, etc...), diferentes tecnologias de transmissão (Ethernet, Fast Ethernet, Gigabit Ethernet, 802.11, etc.) e diferentes cenários de utilização (diferentes taxas de transmissão, por exemplo).

Não foi possível realizar uma avaliação adequada nesta monografia devido ao esforço necessário e o tempo disponível. Por outro lado, realizou-se um esforço no sentido de **planejar** a avaliação. Nas seções a seguir, são descritas premissas ambientais e operacionais exigidas em uma futura avaliação.

5.1 Premissas Ambientais

As premissas ambientais dizem respeito à infra-estrutura sobre a qual o protocolo TCP-XM vai operar. A velocidade de comunicação entre transmissor e receptores é afetada diretamente por fatores que envolvem o hardware dos nodos da rede e a qualidade dos canais de comunicação utilizados. Os fatores a seguir devem ser considerados durante uma avaliação da implementação realizada nesta monografia.

A **taxa de perda de pacotes** é a taxa percentual de perdas aleatórias de pacotes ocorridas durante uma sessão TCP-XM.

Em uma rede local cabeada, a taxa de perdas é usualmente negligível, pois há muito mais capacidade do que demanda, e os meios de transmissão raramente levam a erros de integridade. Já em uma rede de longo alcance, podem ocorrer perdas em níveis elevados, causadas usualmente por congestionamento na rede. Adicionalmente, em redes com enlaces sem fio, sejam locais ou de longo alcance, perdas são usuais e relacionadas à atenuação do sinal, ruído e interferência. Por fim, podem ocorrer defeitos em interfaces de rede, conectores e cabeamento, que podem levar à perda total de pacotes. O protocolo deveria ser testado com diversas taxas de perda de pacotes, geradas de forma aleatória, como por exemplo 0% (sem perdas), 1%, 5%, 10%, 50% e 90%.

O resultado esperado, embora não desejado, é que a implementação do TCP-XM diminua a taxa de envio (em função do mecanismo de controle de congestionamento). Mais importante, o mecanismo de controle de erro deve funcionar adequadamente, detectando e corrgindo os erros encontrados. Assim sendo, a transmissão de dados deve continuar fazendo progresso e mais cedo ou mais tarde entregar a cada um dos receptores uma cópia integral dos dados transmitidos, tal e qual enviados pelo transmissor.

Experimentos devem ser executados considerando uma **carga na rede**, particularmente com múltiplos fluos passando através de um “enlace gargalo”. O protocolo TCP-

XM deve adaptar-se a diversas cargas, pois trata-se de um protocolo elástico. Devem ser consideradas diferentes taxas de tráfego de fundo, geradas por uma outra máquina através de um fluxo de taxa constante (CBR), com taxas tais como 0%, 10%, 50% e 90% da capacidade do enlace gargalo.

O resultado esperado deste experimento é que a implementação do TCP-XM se adapte às condições da rede (no gargalo) e tome conta do restante da capacidade disponível no enlace gargalo. Seria um erro caso o TCP-XM enviasse muito mais do que a parcela restante, gerando perdas desnecessárias, assim como seria ineficiente enviar menos. Mais especificamente, a taxa de transmissão do TCP-XM será guiada pelo receptor mais lento; no experimento, é possível colocar apenas um receptor, ou todos, atrás do enlace gargalo. Por fim, este experimento poderia ser realizado variando-se dinamicamente o tráfego de fundo, avaliando-se a capacidade do TCP-XM em se adaptar e a velocidade com que isso ocorre.

Em um cenário realista, receptores podem estar situados em uma rede local ao transmissor, agrupados em uma rede local distante, ou espalhados geograficamente. É necessário avaliar o comportamento do protocolo nesses três cenários. No primeiro caso, o RTT é bastante baixo e homogênio entre todos os receptores. No segundo caso, os RTTs também são homogêneos, porém maiores. No terceiro e último cenário, o conjunto de RTTs entre transmissor e receptores é heterogêneo. Para avaliar adequadamente, deveriam ser considerados, para o primeiro experimento, um RTT de 1ms, para o segundo, RTTs de 100ms e 500ms, e para o terceiro, RTTs de 0,5ms, 1ms, 10ms, 100 ms e 500ms uniformemente distribuídos no grupo de receptores.

Quanto aos resultados, espera-se que o TCP-XM opere “normalmente” (de maneira razoavelmente eficientemente) nos três cenários. No terceiro cenário, em particular, é normal que o desempenho seja ditado pelo receptor mais lento, que será em tese aquele(s) com maior RTT.

A **largura de banda** dos enlaces é decisiva quanto ao throughput do protocolo. O presente teste deve avaliar o comportamento do protocolo TCP-XM em enlaces com diferentes capacidades, medindo-se o throughput máximo, em condições ideais. Num ambiente real de produção, por exemplo, um receptor pode estar conectado ao transmissor através de cabo par-trançado a 100 Mbps e outro pode estar conectado através de uma interface wireless 802.11b. Minimamente, devem ser considerados tamanhos de enlace de 56Kbps, 1Mbps, 10Mbps e 100Mbps; se possível, enlaces de 1Gbps deveriam ser também testados.

Como citado anteriormente, tecnologias de transmissão empregadas nos enlaces hoje em dia variam bastante. Uma das propriedades em que enlaces diferem é o **tamanho do MTU**. É recomendável certificar-se de que o protocolo TCP-XM é capaz de trabalhar eficientemente com diferentes tamanhos de MTU. O experimento deve avaliar o funcionamento da implementação considerando diferentes valores de MTU, sugerindo-se adotar como valores de teste 500, 1500 e 2000 bytes. O teste avalia o comportamento no caminho fim-a-fim. Para executar o experimento, seria necessário dispor de uma rede física com tais MTUs, mas uma outra alternativa mais prática seria configurar no próprio sistema do remetente, via setsockopt.

Com a evolução das tecnologias de transmissão em rede, a largura de banda é cada vez maior nos enlaces, porém a capacidade de processamento dos computadores não acompanha na mesma velocidade. Em caso de protocolos complexos, com muito processamento por pacote, ou uso de criptografia, o processamento pode ser o fator limitante no throughput obtido. Portanto, além da taxa de erro e a largura de banda dos enlaces, a con-

figuração (capacidade de processamento) dos nodos da rede, particularmente da máquina que está enviando dados, será fundamental para o bom funcionamento do protocolo. Este experimento visa avaliar a capacidade máxima de transmissão da implementação TCP-XM no kernel, em situações de borda em que o transmissor possue grande capacidade e pequena capcidade, para um número susbtancial de receptores. A aplicação no transmissor tenta transferir na maior velocidade possível um arquivo grande para um conjunto de receptores em uma rede local cabeada, sem tráfego de fundo. Os receptores podem ser executados em diferentes configurações de hardware e o protocolo TCP-XM deve tratar esta situação.

Os resultados desse teste devem mostrar que a implementação do protocolo explora eficientemente o hardware disponível no lado do transmissor, e não sobrecarrega em demasia a CPU no lado receptor. O throughput atingido deve ser comparado a um experimento adicional conduzido com TCP e unicast, para as mesmas máquinas. Para um número baixo de receptores, o resultado deve ser próximo à capacidade obtida pelo TCP. Na prática, considerando hardware de prateleira de alta capacidade, pode-se chegar a quase 500Mbps em uma rede Gigabit.

5.2 Premissas Operacionais

As premissas operacionais para avaliação da implementação realizada nesta monografia dizem respeito à maneira como a aplicação do transmissor e dos receptores utiliza o protocolo TCP-XM. Os fatores a seguir devem ser considerados em uma avaliação.

Possivelmente o fator mais importante, o **número de receptores** define a quantidade de receptores em uma sessão TCP-XM. Devem ser considerados sessões TCP-XM com 1, 2, 4, 8, 16 e 32 receptores. Note-se que por definição o TCP-XM não comporta um número muito maior de receptores.

O resultado deve mostrar que o desempenho cai a medida que aumenta o número de receptores, pois quanto maior o grupo, maior será o tempo gasto no estabelecimento e finalização da sessão TCP-XM, bem como na cópia dos dados transmitidos em todas as PCBs. Além disso, maior será a probabilidade (cumulativa) que pelo menos um receptor perca um pacote, demande uma retransmissão e afete o algoritmo de controle de congestionamento. Espera-se que o experimento mostre que o throughput não converge para zero em função de perdas comuns observadas por múltiplos receptores.

Outro aspecto operacional importante é a **quantidade de bytes transmitida** pela aplicação em uma sessão TCP-XM. Existe, provavelmente, uma sobrecarga que é fixa para sessões com o mesmo número de receptores; ou seja, independe da quantidade de dados que a aplicação envia. O teste deve avaliar o impacto da quantidade de bytes transmitida na eficiência do protocolo, comparando situações extremas em que se envia 1 byte e quando se envia 1 Gigabyte. Os resultados são afetados pelo modo de transmissão, naturalmente; por exemplo, caso IP Multicast nativo esteja disponível na sessão, o tempo gasto para transmissão dos segmentos pode ser diminuído significativamente. Devem ser consideradas transmissões de 1B, 1kB, 100kB, 1MB e 100MB tanto na transmissão IP Multicast nativo quanto na transmissão multicast sobre unicast.

Os resultados devem ilustrar a existência desse custo fixo, decorrente do tempo de estabelecimento e fechamento da sessão. A sobrecarga de comunicação também deve ser bastante diferente entre as situações de 1 byte e 1 MB, porque no primeiro caso existe uma sobrecarga proporcionalmente significativa de segmentos necessários para estabelecimento da conexão, bem como nos cabeçalhos do único segmento.

O **tamanho dos segmentos** representa a forma com que os bytes a serem transmitidos são passados pela aplicação à camada de transporte. Por exemplo, a aplicação pode enviar 10000 segmentos de 1 byte, ou enviar 10 segmentos de 1000 bytes. O experimento deve explicitamente testar casos em que o tamanho dos segmentos exceda o MTU, provocando fragmentação. Devem ser considerados segmentos de 1B, 100B, 500B, 1000B, 1500B e 32000B, para um MTU de 1500 bytes.

Os resultados devem mostrar que o protocolo de transporte consegue lidar com todos os tamanhos de segmentos passados pela aplicação, dentro do que se admite para um socket (até 64 kB), consolidando múltiplos bytes em segmentos maiores bem como efetuando fragmentação de segmentos maiores quando estes excederem o MTU.

Experimentos devem ser realizados de forma a mostrar que a implementação lida consistentemente com múltiplas **sessões TCP-XM concorrentes**, iniciadas pela mesma máquina transmissora (empregando números de porta diferentes, naturalmente). O teste avalia a gerência realizada pelo kernel de múltiplas sessões TCP-XM, que compartilham a lista encadeada de receptores. É interessante também avaliar o comportamento do protocolo TCP-XM implementado em relação à utilização de máquinas multiprocessadas para transmissão dos dados, para efeito de suporte à SMP e travas no kernel. Devem ser considerados duas sessões, quatro sessões e oito sessões concorrentes.

Por fim, deve-se considerar experimentos relativos à **gerência de sessão**, em que receptores abandonam a sessão em meio ao estabelecimento da conexão, durante a fase de transmissão de dados, e no início da fase de encerramento iniciada pela máquina transmissora. Além de explorar casos em que receptores abandonam a sessão, é necessário testar situações em que o transmissor (a aplicação transmissora) decide abortar a sessão em uma das três fases de uma sessão, já citadas.

6 CONSIDERAÇÕES FINAIS

O objetivo final desta monografia era realizar uma implementação em nível de kernel do protocolo TCP-XM.

Para este fim, foi necessário primeiramente escolher o ferramental de desenvolvimento do protocolo. Após foi necessário estudar o subsistema de rede do kernel GNU/Linux e investigar a arquitetura de implementação de um protocolo de rede. Nesta primeira fase, foram também estudados protocolos de multicast confiável, particularmente aqueles que buscam oferecer funcionalidade similar ao TCP, porém para múltiplos receptores.

A implementação iniciou com a inserção do protocolo TCP-XM no kernel GNU/Linux através da duplicação e adaptação do protocolo TCP para operar sobre múltiplas sessões. Após inserido o suporte a transmissão multicast, foi implementada a máquina de estados de sessão do protocolo TCP-XM. Na fase final foi necessário implementar o mecanismo de sincronização das múltiplas janelas de transmissão, trabalho que representou um grande desafio.

Durante a implementação em nível de kernel, foram criados diversos mini-aplicativos e scripts para teste e depuração. O aplicativo para transmissão de dados criado foi denominado *ftpxm*. Foi inserido também no analisador de pacotes *tcpdump* o suporte ao protocolo TCP-XM. Esta alteração é vital para depuração de uma sessão TCP-XM.

Devido à complexidade encontrada durante a fase de implementação, não foi realizada uma avaliação apropriada da implementação descrita nesta monografia. No entanto, o trabalho intelectual de definição dos experimentos para avaliação, e os resultados esperados, foram citados.

A descrição do protocolo TCP-XM realizada em [Jeacle 2005] é bastante rica e completa, e foi seguida à risca com uma notável exceção: uma pequena otimização no mecanismo de detecção do suporte a multicast para facilitar a implementação no transmissor.

De um ponto de vista empírico, a transmissão de multicast confiável utilizando o TCP-XM é superior a utilização de múltiplas conexões unicast em contexto de usuário. Entretanto, não é possível realizar esta afirmação sem uma bateria intensiva de testes.

O desenvolvimento de um protocolo multicast confiável em nível de kernel é uma atividade extensa e que envolve uma série de estudos e experimentos para melhoria da implementação, além de um possível aumento na robustez das aplicações. Existem trabalhos de experimentação e testes com a implementação que devem ser realizados, seguido de uma melhoria e possível otimização do código fonte da implementação no kernel. Dentre eles pode-se citar:

- **experimentos com multicast nativo**: uma bateria de testes mais intensa utilizando IP multicast nativo deve ser realizada para validação da implementação realizada.

Os testes mais intensos desta monografia foram realizados em multicast sobre unicast.

- **comparação desta implementação com protocolos de multicast confiável**: uma bateria de comparações entre a implementação do TCP-XM em nível de kernel descrita nesta monografia com outros protocolos de multicast confiável que propiciam TCP sobre multicast deve ser realizada. Outra comparação que deve ser realizada é com a implementação no contexto de usuário descrita em [Jeacle 2005].

- **melhoria e otimização da implementação**: existem trechos de código que podem ser otimizados e agrupados de melhor forma dentro do subsistema de rede do kernel GNU/Linux. Existem dezenas de funções que são muito similares à implementação do protocolo TCP e portanto podem ser agrupadas.

- **testes da implementação no kernel em diferentes arquiteturas de hardware**: a implementação realizada foi testada somente em arquiteturas `i386` e `x86_64`. Uma bateria intensiva de testes deve ser realizada em outras arquiteturas como `arm` e `m68k` para utilização do protocolo TCP-XM em dispositivos embarcados.

REFERÊNCIAS

[Allman 2003]ALLMAN, M. *TCP Congestion Control with Appropriate Byte Counting (ABC): (RFC) 3465*. [S.l.], 2003.

[Andreasson 2002]ANDREASSON, O. *Ipsysctl tutorial 1.0.4*. 2002. Disponível em: <http://ipsysctl-tutorial.frozentux.net/chunkyhtml/index.html>. Acesso em: jun 2006.

[Beck 2002]BECK, M. *Linux Kernel Programming*. 3rd. ed. Great Britain: [s.n.], 2002.

[Bellard 2005]BELLARD, F. Qemu, a fast and portable dynamic translator. In: *Proceedings of the Usenix 2005 Annual Conference, FREENIX track*. [S.l.: s.n.], 2005. p. 41–46.

[Bellard 2006]BELLARD, F. *Qemu*. 2006. Disponível em: <http://fabrice.bellard.free.fr/qemu/>. Acesso em: nov 2006.

[Bennett e Zhang 1997]BENNETT, J. C. R.; ZHANG, H. Hierarchical packet fair queueing algorithms. In: IEEE/ACM (Ed.). [S.l.: s.n.], 1997. (IEEE/ACM Transactions on Networking„ v. 5), p. 675–689.

[Bieringer 2006]BIERINGER, P. *Linux: IPV6*. 2006. Disponível em: <http://www.bieringer.de/linux/IPv6/>. Acesso em: nov 2006.

[Bligh et al. 2005]BLIGH, M. J. et al. Can you handle the pressure? making linux bulletproff under load. In: *Proc. of the Linux Symposium 2005*. Otawwa:Canada: [s.n.], 2005. v. 1, p. 29–40.

[Bossek 2006]BOSSEK, R. *Bluez: Official Linux Bluetooth Protocol Stack*. 2006. Disponível em: <http://www.bluez.org/>. Acesso em: nov 2006.

[Braden 1989]BRADEN, R. *Requirements for Internet Hosts - Communication Layers: RFC 1122*. [S.l.], 1989.

[Chang e Maxemchuk 1984]CHANG, J.-M.; MAXEMCHUK, N. F. Reliable broadcast protocols. In: ACM (Ed.). *Proc. of ACM SIGCOMM*. [S.l.: s.n.], 1984. (Transactions on Computer Systems, v. 2), p. 173–251.

[Cheriton 1987]CHERITON, D. R. Uio: a uniform i/o system interface for distributed systems. *ACM Trans. Comput. Syst.*, ACM Press, New York, NY, USA, v. 5, n. 1, p. 12–46, 1987.

[Cisco 2002]CISCO. *Decnet*. 2002. Disponível em: <www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/decnet.htm>. Acesso em: nov 2006.

[Cisco 2005]CISCO. *Ethernet*. 2005. Disponível em: <http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/ethernet.htm>. Acesso em: nov 2006.

[Comer 1998]COMER, D. E. *Interligação de Redes com TCP/IP*. 3rd. ed. Rio de Janeiro, Brasil: [s.n.], 1998.

[Corbic 2006]CORBIC, N. *WANPIPE*. 2006. Disponível em: <http://freshmeat.net/projects/wanpipe/>. Acesso em: nov 2006.

[Cox 1996]COX, A. *Network Buffers and Memory Management*. 1996. Disponível em: <http://www.linuxjournal.com/article/1312>. Acesso em: nov 2006.

[DARPA 1981]DARPA. *Transmission Control Protocol: (RFC) 793*. [S.l.], 1981.

[Deering 1989]DEERING, S. *Host Extensions for IP Multicasting: RFC 1112*. [S.l.], 1989.

[Dhandapani 1999]DHANDAPANI, G. *Netlink Sockets: an Overview*. 1999. Disponível em: <http://qos.ittc.ku.edu/netlink/html/node1.html>. Acesso em: nov 2006.

[Dike 2006]DIKE, J. *User-Mode-Linux*. 2006. Disponível em: <http://user-mode-linux.sourceforge.net/>. Acesso em: nov 2006.

[Dragovic et al. 2003]DRAGOVIC, B. et al. Xen and the art of virtualization. In: *Proceedings of the ACM Symposium on Operating Systems Principles*. [S.l.: s.n.], 2003.

[Druschel e Banga 1996]DRUSCHEL, P.; BANGA, G. Lazy receiver processing (lrp): A network subsystem architecture for server systems. In: *Symp. on Operating Systems Design and Implementation*. [S.l.: s.n.], 1996. v. 2, p. 29–40.

[Dunkels 2001]DUNKELS, A. *Minimal TCP/IP Implementation with Proxy Support*. Kista, Sweden, 2001.

[Dunkels 2004]DUNKELS, A. *lwIP: A Lightweight TCP/IP Stack*. 2004. Disponível em: <http://www.sics.se/ adam/lwip/>. Acesso em: nov 2006.

[Eklektix 2003]EKLEKTIX. *Driver porting: The seq_file interface*. 2003. Disponível em: <http://lwn.net/Articles/22355/>. Acesso em: jun 2006.

[Ethereal 2006]ETHEREAL. *Ethereal*. 2006. Disponível em: <http://www.ethereal.com/>. Acesso em: nov 2006.

[Fairhust 2006]FAIRHUST, G. *IEEE 802.3 Logical Link Control*. 2006. Disponível em: <http://www.erg.abdn.ac.uk/users/gorry/course/lan-pages/llc.html>. Acesso em: nov 2006.

[1]FENTON, J.; DEMPSEY, B.; WEAVER, A. The multidriver: A reliable multicast service using the xpress transfer protocol. In: IEEE/ACM (Ed.). Minnesota, USA: [s.n.], 1990. (IEEE/ACM Transactions on Networking„ v. 5), p. 351–358.

[Fisk e Feng 2001]FISK, M.; FENG, W. Dynamic right-sizing in tcp. In: *Los Alamos Computer Science Institue Symposium*. [S.l.: s.n.], 2001.

[Floyd et al. 1997]FLOYD, S. et al. A reliable multicast framework for light-weight sessions and application level framing. In: . [S.l.: s.n.], 1997. v. 5, n. 6, p. 784–803.

[Floyd et al. 2000]FLOYD, S. et al. *An Extension to the Selective Acknowledgement (SACK) Option for TCP: (RFC) 2883*. [S.l.], 2000.

[Ghosh e Varghese]GHOSH, R.; VARGHESE, G. *Congestion Control in Multicast Transport Protocols*. Disponível em: <citeseer.ist.psu.edu/169032.html>. Acesso em: mai 2006.

[Greear 2005]GREEAR, B. *802.1Q VLAN Implementation for Linux*. 2005. Disponível em: <http://www.candelatech.com/ greear/vlan.html>. Acesso em: nov 2006.

[2]HANDLEY, M.; KOHLER, E.; FLOYD, S. *Datagram Congestion Control Protocol (DCCP): (RFC) 4340*. [S.l.], 2006.

[Hemminger 2006]HEMMINGER, S. *Linux-Net Wiki Pages*. 2006. Disponível em: <http://linux-net.osdl.org/index.php/Main_Page>. Acesso em: nov 2006.

[Heuser 2003]HEUSER, W. *Linux Infra-Red HOWTO*. 2003. Disponível em: <http://tuxmobil.org/howto_linux_infrared.html>. Acesso em: nov 2006.

[Hobbit 2006]HOBBIT. *The GNU Netcat: Official Home Page*. 2006. Disponível em <http://netcat.sourceforge.net/>. Acesso em nov 2006.

[Hutzelman 2006]HUTZELMAN, J. *OpenAFS*. 2006. Disponível em: <http://www.openafs.org>. Acesso em: nov 2006.

[IBM 2004]IBM. *SCTP for the Linux Kernel*. 2004. Disponível em: <http://lksctp.sourceforge.net/>. Acesso em: nov 2006.

[Jacobson 1988]JACOBSON, V. Congestion avoidance and control. In: *ACM SIGCOMM '88*. Stanford, CA: [s.n.], 1988. p. 314–329.

[3]JACOBSON, V.; BORMAN, D.; BRADEN, R. *TCP Extensions for High Performance: (RFC) 1323*. [S.l.], 1992.

[Jeacle 2005]JEACLE, K. *TCP-XM*. Tese (Doutorado) — Wolfson College/Cambridge, Barton Road, UK, 2005.

[Jeacle e Crowcroft 2004]JEACLE, K.; CROWCROFT, J. Extending globus to support multicast transmission. In: CAMBRIDGE, U. of (Ed.). *Proc. of the UK E-Science AHM2004*. Nottingham, Inglaterra: Computer Laboratory, 2004.

[Jeacle e Crowcroft 2005]JEACLE, K.; CROWCROFT, J. mcp: A reliable multicast file transfer tool. In: CAMBRIDGE (Ed.). [S.l.: s.n.], 2005. (4th UK E-SCIENCE ALL HANDS MEETING).

[Jeacle e Crowcroft 2005]JEACLE, K.; CROWCROFT, J. Tcp-xm: Unicast-enabled reliable multicast. In: IEEE (Ed.). *Proc. of the 14th International Conference on Computer Communications and Networks*. San Diego, USA: [s.n.], 2005.

[Jeacle et al. 2005]JEACLE, K. et al. Hybrid reliable multicast with tcp-xm. In: ACM (Ed.). *Proc. of the 1st International Conference on Emerging Networking Experiments and Technology*. Toulouse, França: ACM, 2005.

[Karn e Partridge 1991]KARN, P.; PARTRIDGE, C. Improving round-trip time estimates in reliable transport protocols. *ACM Transactions on Computer Systems*, v. 9, n. 4, p. 364–373, 1991.

[Kent e Atkinson 1998]KENT, S.; ATKINSON, R. *Security Architecture for the Internet Protocol: (RFC) 2401*. [S.l.], 1998.

[Ketrenos 2006]KETRENOS, J. *ieee80211 Subsystem for Linux*. 2006. Disponível em: <http://ieee80211.sourceforge.net/>. Acesso em: nov 2006.

[Kleen 2001]KLEEN, A. *Patch Netconsole: Log Kernel Messages Over the Network*. 2001. Disponível em <http://lwn.net/2001/0927/a/netconsole.php3>. Acesso em nov 2006.

[4]KROAH-HARTMAN, G.; CORBET, J.; RUBINI, A. *Linux Device Drivers*. 3rd. ed. New York: [s.n.], 2005.

[Kurose e Ross 2000]KUROSE, J. F.; ROSS, K. W. *Computer Networking: A Top-Down Approach Featuring the Internet*. 1st. ed. Reading, USA: [s.n.], 2000.

[Law 2006]LAW, D. *IEEE 802.3 CSMA/CD (Ethernet)*. 2006. Disponível em: <http://www.ieee802.org/3/>. Acesso em: nov 2006.

[Leffler et al. 1993]LEFFLER, S. J. et al. *An Advanced 4.4BSD Inter-Process Communication Tutorial*. 1993. Disponível em: <http://www-users.cs.umn.edu/ bentlema/unix/advipc/ipc.html>. Acesso em: nov 2006.

[Lenz 2006]LENZ, P. *TCPDump*. 2006. Disponível em: <http://www.tcpdump.org>. Acesso em: nov 2006.

[Levine e Garcia-Luna-Aceves 1996]LEVINE, B. N.; GARCIA-LUNA-ACEVES, J. J. A comparison of known classes of reliable multicast transport protocol. In: IEEE/ACM (Ed.). [S.l.: s.n.], 1996. (International Conference on Network Protocols).

[Liang e Cheriton 2002]LIANG, S.; CHERITON, D. R. Tcp-smo: Extending tcp to support medium-scale multicast applications. In: *INFOCOM*. [S.l.: s.n.], 2002.

[Lin e Paul 1996]LIN, J. C.; PAUL, S. Rmtp: A reliable multicast transport protocol. In: *INFOCOM*. San Francisco, CA: [s.n.], 1996. p. 1414–1424.

[Mankin 1998]MANKIN, A. *IETF Criteria for Evaluating Reliable Multicast Transport and Application Protocol: RFC 2357*. [S.l.], 1998.

[Marmion 2004]MARMION, J. *Representação e Validação do projeto transacional mediante Diagramas de Transição de Estados*. 2004. Disponível em: <http://www.ibrau.com.br/diagramasdetransicaodeestados.htm>. Acesso em: mai 2006.

[Mathis e Mahdavi 1996]MATHIS, M.; MAHDAVI, J. Forward acknowledgement: Refining TCP congestion control. In: *SIGCOMM*. [S.l.: s.n.], 1996. p. 281–291.

[Mathis et al. 1996]MATHIS, M. et al. *TCP Selective Acknowledgment Options: (RFC) 2018*. [S.l.], 1996.

[McKenney 1990]MCKENNEY, P. Stochastic fairness queuing. In: IEEE (Ed.). [S.l.: s.n.], 1990. (IEEE INFOCOMM'90).

[Menegotto 2006]MENEGOTTO, A. *FTPXM's Page*. 2006. Disponível em <http://planeta.terra.com.br/informatica/menegotto/userland/ftpxm/>. Acesso em nov 2006.

[Menegotto 2006]MENEGOTTO, A. *TCPXM's Page*. 2006. Disponível em <http://planeta.terra.com.br/informatica/menegotto/tcpxm/>. Acesso em nov 2006.

[Menegotto 2006]MENEGOTTO, A. *TCPXM's Userland Page*. 2006. Disponível em <http://planeta.terra.com.br/informatica/menegotto/userland/>. Acesso em nov 2006.

[Menegotto e Barcellos 2006]MENEGOTTO, A.; BARCELLOS, M. P. *include/linux/tcpxm.h*. 2006. Disponível em <http://planeta.terra.com.br/informatica/menegotto/linux-2.6/include/linux/tcpxm.h>. Acesso em nov 2006.

[Menegotto e Barcellos 2006]MENEGOTTO, A.; BARCELLOS, M. P. *include/net/tcpxm_ecn.h*. 2006. Disponível em <http://planeta.terra.com.br/informatica/menegotto/linux-2.6/include/net/tcpxm_ecn.h>. Acesso em nov 2006.

[Menegotto e Barcellos 2006]MENEGOTTO, A.; BARCELLOS, M. P. *include/net/tcpxm.h*. 2006. Disponível em <http://planeta.terra.com.br/informatica/menegotto/linux-2.6/include/net/tcpxm.h>. Acesso em nov 2006.

[Menegotto e Barcellos 2006]MENEGOTTO, A.; BARCELLOS, M. P. *include/net/tcpxm_states.h*. 2006. Disponível em <http://planeta.terra.com.br/informatica/menegotto/linux-2.6/include/net/tcpxm_states.h>. Acesso em nov 2006.

[Menegotto e Barcellos 2006]MENEGOTTO, A.; BARCELLOS, M. P. *net/ipv4/tcpxm.c*. 2006. Disponível em <http://planeta.terra.com.br/informatica/menegotto/linux-2.6/net/ipv4/tcpxm.c>. Acesso em nov 2006.

[Menegotto e Barcellos 2006]MENEGOTTO, A.; BARCELLOS, M. P. *net/ipv4/tcpxm_input.c*. 2006. Disponível em <http://planeta.terra.com.br/informatica/menegotto/linux-2.6/net/ipv4/tcpxm_input.c>. Acesso em nov 2006.

[Menegotto e Barcellos 2006]MENEGOTTO, A.; BARCELLOS, M. P. *net/ipv4/tcpxm_ipv4.c*. 2006. Disponível em <http://planeta.terra.com.br/informatica/menegotto/linux-2.6/net/ipv4/tcpxm_ipv4.c>. Acesso em nov 2006.

[Menegotto e Barcellos 2006]MENEGOTTO, A.; BARCELLOS, M. P. *net/ipv4/tcpxm_minisocks.c*. 2006. Disponível

em <http://planeta.terra.com.br/informatica/menegotto/linux-2.6/net/ipv4/tcpxm_minisocks.c>. Acesso em nov 2006.

[Menegotto e Barcellos 2006]MENEGOTTO, A.; BARCELLOS, M. P. *net/ipv4/tcpxm_output.c*. 2006. Disponível em <http://planeta.terra.com.br/informatica/menegotto/linux-2.6/net/ipv4/tcpxm_output.c>. Acesso em nov 2006.

[Menegotto e Barcellos 2006]MENEGOTTO, A.; BARCELLOS, M. P. *net/ipv4/tcpxm_timer.c*. 2006. Disponível em <http://planeta.terra.com.br/informatica/menegotto/linux-2.6/net/ipv4/tcpxm_timer.c>. Acesso em nov 2006.

[5]METZ, C.; MCDONALD, D.; PHAN, B. *PF_KEY Key Management API Version 2: (RFC) 2367*. [S.l.], 1998.

[Miller et al. 1997]MILLER, K. et al. *Starburst Multicast File Transfer Protocol (MFTP) Specification: Internet Draft*. [S.l.], 1997.

[Mogul e Deering 1990]MOGUL, J.; DEERING, S. *Path MTU Discovery: (RFC) 1191*. [S.l.], 1990.

[Murai 2006]MURAI, K. J. *Linux IPv6 Development Project*. 2006. Disponível em: <http://www.linux-ipv6.org/>. Acesso em: nov 2006.

[Murray 1999]MURRAY, R. *Econet Enthusiasts*. 1999. Disponível em: <http://www.heyrick.co.uk/econet/>. Acesso em: nov 2006.

[Mysore e Varghese 2006]MYSORE, M.; VARGHESE, G. *FTP-M: An FTP-like Multicast File Transfer Application*. 2006. Disponível em <http://www-cse.ucsd.edu/Dienst/Repository/2.0/Body/ncstrl.ucsd_cse/CS2001-0684/postscript>. Acesso em nov 2006.

[Nagle 1984]NAGLE, J. *Congestion Control in IP/TCP Internetworks: (RFC) 896*. [S.l.], 1984.

[Ng 1999]NG, T. S. E. *Hierarchical Packet Schedulers*. 1999. Disponível em: <http://www.cs.cmu.edu/ hzhang/HFSC/main.html>. Acesso em: nov 2006.

[Novell 1989]NOVELL. *NetWare System Technical Interface Overview*. 1st. ed. Reading, USA: [s.n.], 1989.

[NSA 2006]NSA. *Security-Enhanced Linux*. 2006. Disponível em <http://www.nsa.gov/selinux/>. Acesso em nov 2006.

[OpenGroup 2006]OPENGROUP. *The Unix System*. 2006. Disponível em: <http://www.unix.org>. Acesso em: nov 2006.

[6]PADHYE, J.; HANDLEY, M.; FLOYD, S. *TCP Congestion Window Validation: (RFC) 2861*. [S.l.], 2000.

[7]PARTDRIGE, C.; ALLMAN, M.; FLOYD, S. *Increasing TCP's Initial Window: (RFC) 2414*. [S.l.], 1998.

[Paul 1998]PAUL, S. *Multicasting on the Internet and Its Applications*. 1st. ed. Massachussets, USA: [s.n.], 1998.

[Paxson e Allman 2000]PAXSON, V.; ALLMAN, M. *Computing TCP's Retransmission Timer: (RFC) 2988*. [S.l.], 2000.

[Pick 2006]PICK, J. *The Linux Kernel Archives*. 2006. Disponível em: <http://www.kernel.org>. Acesso em: nov 2006.

[8]PINGALI, S.; TOWSLEY, D.; KUROSE, J. F. A comparison of sender-initiated and receiver-initiated reliable multicast protocols. In: . New York, NY, USA: ACM Press, 1994. v. 22, n. 1, p. 221–230.

[Postal 1982]POSTAL, J. B. *Simple Mail Transport Protocol: RFC 821*. [S.l.], 1982.

[9]RAMAKRISHNAN, K.; FLOYD, S.; BLACK, D. *The Addition of Explicit Congestion Notification (ECN) to IP: (RFC) 3168*. [S.l.], 2001.

[Reynolds e Postel 1985]REYNOLDS, J.; POSTEL, J. B. *File Transfer Protocol: RFC 959*. [S.l.], 1985.

[Rivest 1992]RIVEST, R. *The MD4 Message-Digest Algorithm: (RFC) 1320*. [S.l.], 1992.

[Russel e Welte 2002]RUSSEL, R.; WELTE, H. *Linux netfilter Hacking HOWTO*. 2002. Disponível em: <http://www.netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO.html>. Acesso em: nov 2006.

[Sarolahti e Kojo 2005]SAROLAHTI, P.; KOJO, M. *Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP and the Stream Control Transmission Protocol (SCTP): (RFC) 4138*. [S.l.], 2005.

[10]SAROLAHTI, P.; KOJO, M.; RAATIKAINEN, K. *F-RTO: A New Recovery Algorithm for TCP Retransmission Timeouts*. Helsinki, Finland, 2002.

[Sarolahti e Kuznetsov 2002]SAROLAHTI, P.; KUZNETSOV, A. Congestion control in linux tcp. In: *Proc. of USENIX 2002*. [S.l.: s.n.], 2002.

[11]SIDHU, G. S.; ANDREWS, R.; OPPENHEIMER, A. *Inside Appletalk*. 2nd. ed. Reading, USA: [s.n.], 1990.

[Srinivasam 1995]SRINIVASAM, R. *RPC: Remote Procedure Call Protocol Specification Version 2: (RFC) 1831*. [S.l.], 1995.

[Stevens 1997]STEVENS, W. *TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms: (RFC) 2001*. [S.l.], 1997.

[Stevens 1994]STEVENS, W. R. *TCP/IP Illustrated*. 14th. ed. Reading, USA: Addison-Wesley, 1994.

[Stewart et al. 2000]STEWART, R. et al. *Stream Control Transmission Protocol: (RFC) 2960*. [S.l.], 2000.

[Talpade e Ammar 1995]TALPADE, R.; AMMAR, M. H. Single connection emulation (sce): an architecture for providing a reliable multicast transport service. In: IEEE/ACM (Ed.). [S.l.: s.n.], 1995. (International Conference on Distributed Computer Systems, v. 15).

[Tanembaum 1994]TANEMBAUM, A. *Distributed Systems: Principles and Paradigms*. 1st. ed. New York: [s.n.], 1994.

[Torvalds e Hamano 2006]TORVALDS, L.; HAMANO, J. C. *Git*. 2006. Disponível em: <http://git.or.cz>. Acesso em: nov 2006.

[Tranter 2001]TRANTER, J. *Linux Amateur Radio AX.25 HOW-TO*. 2001. Disponível em: <http://www.tldp.org/HOWTO/AX25-HOWTO/>. Acesso em: nov 2006.

[Viro 2006]VIRO, A. *Linux Kernel Documentation: networking.txt*. 2006. Disponível em <http://www.mjmwired.net/kernel/Documentation/networking/netconsole.txt>. Acesso em nov 2006.

[Visoottiviseth 2003]VISOOTTIVISETH, V. *Sender-Initiated Multicast for Small-Group Communication*. Tese (Doutorado) — Gratuated School of Information Science, Takayama District, Japão, 2003.

[Visoottiviseth et al. 2001]VISOOTTIVISETH, V. et al. M/tcp: The multicast-extension to transmission control protocol. In: *Proc. of the 3rd International Conference on Advanced Communication Technology*. Muju:Korea: [s.n.], 2001. (Lecture Notes in Computer Science).

[VMWare 2006]VMWARE. *VMWare*. 2006. Disponível em: <http://www.vmware.com/>. Acesso em: nov 2006.

[Welte 1996]WELTE, H. *skb - Linux Network Buffers*. 1996. Disponível em: <http://ftp.gnumonks.org/pub/doc/skb-doc.html>. Acesso em: nov 2006.

[12]WHETTEN, B.; MONTGOMERY, T.; KAPLAN, S. M. A high performance totally ordered multicast protocol. In: IEEE (Ed.). *Proc. of the 14th IEEE INFOCOM*. Boston, USA: [s.n.], 1995. (Lecture Notes in Computer Science).

[Zeldovich 2002]ZELDOVICH, N. *Rx protocol specification draft*. 2002. Disponível em: <http://web.mit.edu/kolya/afs/rx/rx-spec>. Acesso em: nov 2006.

[Zhang et al. 2002]ZHANG, M. et al. *Improving TCP's Performance under Reordering with DSACK*. [S.l.], 2002.