

UNIVERSIDADE FEDERAL DE SANTA CATARINA

**IMPLEMENTAÇÃO DE UM PROTOCOLO DE CONTROLE DE
SESSÃO PARA APLICAÇÕES TCP/IP**

Rafael Henchen

**UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE CIÊNCIAS DA COMPUTAÇÃO**

**Implementação de um protocolo de controle de sessão para
aplicações TCP/IP.**

Autor:
Rafael Henchen

Orientadora:
Dayna Maria Bortoluzzi, Msc.

Banca Examinadora:
Professor Antônio A. Medeiros Fröhlich, Dr.
Professora Elizabeth Sueli Specialski, Dra.

Palavras-chave:
Controle de Sessão, Socket API, Implementação do TCP/IP, BSD Net/3,
Middleware

Florianópolis, 16 de fevereiro de 2004.

**UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE CIÊNCIAS DA COMPUTAÇÃO**

**Implementação de um protocolo de controle de sessão para
aplicações TCP/IP.**

Este trabalho de conclusão de curso foi julgado adequado à obtenção de grau de Bacharelado em Ciências da Computação e aprovado em sua forma final pelo Curso de Ciências da Computação da Universidade Federal de Santa Catarina.

Florianópolis, 13 de fevereiro de 2004.

Dayna Maria Bortoluzzi, Msc.
Universidade Federal de Santa Catarina

Professor Antônio A. Medeiros Fröhlich, Dr.
Universidade Federal de Santa Catarina

Professora Elizabeth Sueli Specialski, Dra.
Universidade Federal de Santa Catarina

DEDICATÓRIA

Aos meus pais:
Tânia de Fátima Hennen,
Ademar Hennen.
Como é bom ser filho de vocês.
Obrigado por me ajudarem a realizar este
sonho.

AGRADECIMENTOS

Obrigado aos meus pais e meu irmão por terem me apoiado durante toda minha vida acadêmica.

Obrigado aos meus tutores Dayna, professora Beth e professor Guto por toda dedicação.

E em especial ao meu anjo da guarda, Betina, por seu amor, por sua compreensão e por ter estado do meu lado, me dando forças durante este momento crucial da minha vida, eu não teria conseguido sem você.

Sumário

SUMÁRIO.....	6
LISTA DE FIGURAS	8
LISTA DE SIGLAS	9
RESUMO	10
1 INTRODUÇÃO	11
2 CONTROLE DE SESSÃO EM REDES DE COMPUTADORES	14
2.1 A CAMADA DE SESSÃO DO RM-OSI	14
2.1.1 <i>Conceitos</i>	15
2.1.2 <i>Fases e Serviços da Camada de Sessão</i>	16
2.1.2.1 Estabelecimento da Conexão	17
2.1.2.2 Transferência de Dados	17
2.1.2.3 Encerramento da Conexão	19
2.1.2.4 Unidades Funcionais da Camada de Sessão	19
2.2 O MODELO TCP-IP	20
2.2.1 <i>Camada de Transporte</i>	22
2.2.2 <i>Camada de Aplicação</i>	23
3 INET SESP – PROTOCOLO DE SESSÃO PARA TCP/IP	24
3.1 TIPOS DE SERVIÇO.....	24
3.2 PRIMITIVAS DE SERVIÇO.....	25
3.3 DEFINIÇÃO DO CABEÇALHO	27
3.4 SERVIÇOS OFERECIDOS.....	28
3.4.1 <i>Estabelecimento de conexão</i>	28
3.4.2 <i>Transferência de dados</i>	29
3.4.3 <i>Gerência de diálogo</i>	29
3.4.4 <i>Pontos de sincronismo</i>	30
3.4.5 <i>Gerência de atividade</i>	30
3.4.6 <i>Encerramento de uma sessão</i>	30
3.5 ADAPTAÇÃO DAS APLICAÇÕES EXISTENTES	31
4 IMPLEMENTAÇÃO DO TCP/IP	32
4.1 MEMORY BUFFERS	34
4.2 CAMADA DE SOCKET	37
4.2.1 <i>Socket System Calls</i>	38
4.3 CAMADA DE PROTOCOLOS.....	42
4.3.1 <i>Interface do Protocolo TCP</i>	45
5 IMPLEMENTAÇÃO DO SESP	48
5.1 MODELOS DE IMPLEMENTAÇÃO DO SESP	48
5.1.1 <i>Rotinas de sessão</i>	48
5.1.2 <i>Modelo de implementação integrado ao Net/3</i>	52
5.1.3 <i>Modelo de implementação como biblioteca de aplicação</i>	54
5.2 PROTOTIPAÇÃO DO SESP	56
5.2.1 <i>PDU do SESP</i>	56
5.2.2 <i>Alocação de dados no protocolo SESP</i>	57
5.2.2.1 SESP Memmory Buffer.....	58
5.2.2.2 SESP Socket Descriptor.....	59
5.2.2.3 SESP Control Block.....	60
5.2.3 <i>Simplified SESP Socket API</i>	62
5.2.3.1 Common API	62
5.2.3.2 SESP Additional API.....	67

5.2.4	<i>Estabelecimento de Conexão e Negociação de Serviços</i>	68
5.2.5	<i>Transmissão de Dados</i>	69
5.2.5.1	<i>Armazenamento de Dados para Sincronização</i>	69
5.2.6	<i>Recepção de Dados</i>	72
5.2.7	<i>Tratamento de Erros</i>	73
6	CONCLUSÃO	75
6.1	DIFICULDADES ENCONTRADAS.....	76
6.2	TRABALHOS FUTUROS.....	76
	REFERÊNCIAS BIBLIOGRÁFICAS	78
	APÊNDICE A – SESP SOCKET API	81
	APÊNDICE B – INET SESP	99
	APÊNDICE C – ESTRUTURAS AUXILIARES (MBUF.H E ERRNO.H)	127
	APÊNDICE D – MAKEFILE	129
	APÊNDICE E – ARTIGO	130

Lista de Figuras

FIGURA 2.1 - ATIVIDADE, DIÁLOGO E PONTOS DE SINCRONIZAÇÃO	16
FIGURA 2.2 - UNIDADES FUNCIONAIS / SERVIÇOS DE SESSÃO / <i>TOKENS</i>	20
FIGURA 2.3 - ARQUITETURA TCP/IP	21
FIGURA 3.1 - PRIMITIVAS DE SERVIÇO DO PROTOCOLO DE SESSÃO	26
FIGURA 3.2 - CABEÇALHO DA PDU DO PROTOCOLO DE SESSÃO	27
FIGURA 4.1 – COMUNICAÇÃO ENTRE AS CAMADAS NA ARQUITETURA DO NET/3 (WRIGHT, 1995).	33
FIGURA 4.2 – ESTRUTURA DA MEMORY BUFFER (WRIGHT, 1995).	34
FIGURA 4.3 – FILA DE MEMORY BUFFERS (WRIGHT, 1995).	35
FIGURA 4.4 – A ESTRUTURA DE ENDEREÇAMENTO GENÉRICA <i>SOCKADDR</i> (CÓDIGO FONTE DO FREEBSD 4.8)	37
FIGURA 4.5 – A ESTRUTURA DE ENDEREÇAMENTO <i>SOCKET</i> PARA INTERNET IPV4.	38
FIGURA 4.6 – CHAMADAS DE SISTEMA PARA COMUNICAÇÃO EM REDES NO NET/3 (WRIGHT, 1995, p. 445)	39
FIGURA 4.7 – USO DE <i>SOCKETS</i> PARA UMA APLICAÇÃO CLIENTE-SERVIDOR SIMPLES (STEVENS, 1998, p. 86)	41
FIGURA 4.8 – PONTOS DE ACESSO A PROTOCOLOS (WRIGHT, 1995, p. 190)	42
FIGURA 4.9 – AS ESTRUTURAS <i>PROTOSWE PR_USRREQS</i> DO BSD. (CÓDIGO FONTE DO FREEBSD 4.8)	43
FIGURA 4.10 – VARÁVEL DE SISTEMA <i>INETSW</i> – EXEMPLO DE DEFINIÇÃO DA ESTRUTURA <i>PROTOSW</i>	44
FIGURA 4.11 – DECLARAÇÃO DE <i>TCP_USRREQS</i> (CÓDIGO FONTE DO FREEBSD 4.8)	45
FIGURA 4.12 – USO DAS PRINCIPAIS ROTINAS TCP DENTRO DO NÚCLEO (WRIGHT, 1995, p. 796)	46
FIGURA 5.1 - MODELO DE FLUXO DAS REQUISIÇÕES DE USUÁRIO AO SESP NO KERNEL DO BSD	52
FIGURA 5.2 - SESP <i>CONTROL BUFFERS</i> NO KERNEL	53
FIGURA 5.3 - MODELO DE FUNCIONAMENTO DA BIBLIOTECA SESP	55
FIGURA 5.4 - CABEÇALHO DA PDU DO SESP	56
FIGURA 5.5 – VISÃO GERAL DAS ESTRUTURAS DO SESP	57
FIGURA 5.6 - SESP <i>MEMMORY BUFFER</i>	58
FIGURA 5.7 – ESTRUTURA DO SESP <i>SOCKET { }</i>	59
FIGURA 5.8 - ESTRUTURA DO <i>SESPCB { }</i>	60
FIGURA 5.9 - SESP SUPORTANDO MAIS DE UM PONTO DE SINCRONIZAÇÃO	70

Lista de Siglas

API.....	Application Program Interface
BSD.....	Berkeley Software Distribution
DARPA.....	Defense Advanced Research Projects Agency
E/S.....	Entrada e Saída
ELF.....	Executable and Linking Format
FTP.....	File Transfer Protocol
GCC.....	Gnu C Compiler
HTTP.....	Hypertext Transfer Protocol
IAB.....	Internet Activity Board
IP.....	Internet Protocol
ITU-T.....	International Telecommunication Union – Telecom Standardization
NIC.....	Network Interface Card
OSI.....	International Organization for Standardization
OSI.....	Open Systems Interconnection
PCB.....	Protocol Control Block
PDU.....	Protocol Data Unit
Posix.....	Portable Operating System Interface
RFC.....	Request for Comments
RM.....	Reference Model
SESP.....	Session Protocol
SO.....	Sistema Operacional
TCP.....	Transmission Control Protocol
TLI.....	Transport Layer Interface
UDP.....	User Datagram Protocol
WWW.....	World Wide Web
XNS.....	Xerox Netowk System

Resumo

Este trabalho é vinculado ao projeto de desenvolvimento de um protocolo de controle de sessão, para aplicações TCP/IP, chamado *Session Protocol* (SESP), que vem sendo desenvolvido em uma tese de doutorado no Programa de Pós-Graduação em Engenharia de Produção da UFSC. Viabilizar a implementação de um protótipo deste protocolo para sua validação experimental, é o objetivo deste trabalho. Como o objetivo final do projeto consiste em integrá-lo ao núcleo de um sistema operacional, a arquitetura de implementação do TCP/IP é pesquisada a fundo, buscando definir um modelo que permita integrar o SESP ao *kernel* do sistema operacional BSD. Uma abordagem alternativa para a fase de prototipação define o protocolo como uma biblioteca de aplicação. O protótipo é implementado como uma biblioteca ELF para sistemas operacionais Linux.

Palavras-chave: Controle de Sessão, Socket API, Implementação do TCP/IP, BSD Net/3, Middleware.

1 Introdução

Com o surgimento das redes de computadores, principalmente com a interligação de redes que utilizam diferentes tecnologias de transmissão de dados, observou-se a necessidade da adoção de modelos padronizados que especificassem as funções dos protocolos e seus inter-relacionamentos. A Defense Advanced Research Projects Agency (DARPA)¹ então veio a consolidar um dos primeiros sistemas heterogêneos de redes de computadores chamado de ARPANET, conhecido hoje como modelo TCP/IP². Este modelo em seus primórdios era apenas tido como uma solução acadêmica, voltada unicamente para os problemas da ARPANET. Assim a *International Organization for Standardization* (ISO), apoiada pela indústria, propôs um modelo mais claro e mais completo que o TCP/IP, este ficou conhecido como *Reference Model - Open Systems Interconnection* (RM-OSI), mas que se mostrou pouco prático e usual. Na década de 90 consolida-se o paradigma comercial da Internet, e surge a *World Wide Web* (WWW), baseando-se na rede de pesquisa internacional criada a partir da ARPANET, e agora apoiada pela iniciativa privada, a Internet acaba por consolidar o modelo TCP/IP como o modelo definitivo para interligação de redes, estimulado pelo seu baixo custo de implantação e desenvolvimento.

O primeiro Sistema Operacional (SO) a implementar a arquitetura TCP/IP foi a versão 4.2 do *Berkley Software Distributio* (BSD), fomentado pelo projeto DARPA, pois as distribuições livres e de código aberto do BSD eram as mais utilizadas nas instituições governamentais e universidades dos EUA, isto com o objetivo de disseminar o modelo em desenvolvimento e para que o maior número possível de pesquisadores e especialistas pudessem participar, livre e espontaneamente, do projeto.

Novas facilidades e serviços foram implementadas nas aplicações da Internet. Uma destas facilidades é o controle de sessões utilizado atualmente em grande parte das aplicações multimídia em tempo real, aplicações distribuídas e aplicações *peer-to-peer*.

¹Agência ligada ao Departamento de Defesa norte americano que incentivou a pesquisa na área de redes de computadores com o apoio de diversas universidades, órgãos governamentais e algumas empresas privadas dos EUA.

² TCP/IP é um acrônimo para *Transmission Control Protocol* e *Internet Protocol*, pois são estes dois protocolos que constituem a base principal de funcionamento do modelo.

Atualmente se encontra em andamento no Programa de Pós-Graduação em Engenharia de Produção da Universidade Federal de Santa Catarina, o projeto de modelagem e desenvolvimento de um protocolo que satisfaça as necessidades das aplicações TCP/IP dos tempos atuais, principalmente aplicações multimídia e de transmissão de grandes massas de dados, e que se baseia nas definições e conceitos da camada de sessão do modelo RM-OSI. Este protocolo atualmente é chamado de *Session Protocol* (SESP).

Desenvolver um protocolo de comunicação de dados pode ser realizado muito mais facilmente com o auxílio de técnicas e ferramenta de descrição formal, sendo que apenas descrevê-lo formalmente garantirá apenas sua validação, mas não garantirá que o mesmo poderá eventualmente ser implementado de forma eficiente. Não se deve esquecer que as características físicas e de modelagem dos diferentes sistemas operacionais tem influência sobre os sistemas que venham a ser utilizados sobre estes. Como por exemplo, na definição dos tipos de dados e tamanhos dos campos dos cabeçalhos do protocolo, número máximo de bytes enviados nos datagramas IP por cada sistema operacional (SO), limitações para alocação e re-alocação de dados, suporte a sinalização por semáforos ou ferramentas similares, estrutura de funcionamento de SOs com suporte a multiprocessos e *threads*, entre outros.

Com a finalidade de contribuir no projeto de pesquisa do protocolo SESP, este trabalho tem como objetivos os seguintes itens:

- Um estudo de como um protótipo do SESP poderá ser implementado dentro da arquitetura TCP/IP dos SOs da família BSD.
- Levantar questões relativas à implementação que possam influenciar na modelagem do protocolo.
- Elaboração de um modelo de software que viabilize a implementação do protocolo.
- Implementar um primeiro protótipo do protocolo SESP capaz de efetuar o estabelecimento de sessão orientada a conexão (via TCP) e que ofereça as funcionalidades básicas já modeladas.

Para isto, adotou-se a seguinte metodologia: o capítulo 2 faz um estudo sobre camadas de sessão em geral, observando principalmente como ela é definida no

modelo OSI, e de como é sua abordagem nas aplicações da pilha TCP/IP; o capítulo 3 apresenta o estado atual do modelo do *Session Protocol* para TCP/IP; o capítulo 4 aborda resumidamente como é implementado o TCP/IP e a API conhecida como BSD *Sockets*, no sistema operacional 4.4 BSD-Lite e seus precursores; e por fim, o capítulo 5 aborda o processo de modelagem e implementação do protótipo o protocolo SESP, tendo como guia, a arquitetura do *Net#3* (Wright, 1995).

2 Controle de Sessão em Redes de Computadores

Neste capítulo será feito um levantamento sobre as funcionalidades da camada de sessão nas duas principais arquiteturas para redes heterogêneas: o RM-OSI e o modelo TCP/IP.

Como no modelo TCP/IP não existe uma camada específica de sessão, serão abordadas as camadas de transporte e aplicação.

2.1 A Camada de Sessão do RM-OSI

O modelo de referência OSI, diferente do TCP/IP, prevê uma camada exclusiva para prestação dos serviços de estabelecimento, gerenciamento e término de sessões. Este tipo de abordagem facilita o desenvolvimento de aplicações, uma vez que elas não precisam implementar estes serviços. Também permite um melhor controle sobre os recursos da rede já que o modelo estabelece políticas para este fim.

Segundo a recomendação X.215 do ITU-T “o serviço de sessão provê os meios necessários para a troca organizada e sincronizada de dados entre usuários co-operantes da camada de sessão” (1995, pg. 16).

Existem dois tipos de serviços de sessão:

- Serviços no modo orientado a conexão, onde a conexão de sessão é mapeada sobre apenas uma conexão de transporte num dado instante. É possível que uma conexão de sessão possa se estender temporalmente sobre mais de uma conexão de transporte (não necessariamente simultâneas), ou mesmo é permitido que duas ou mais conexões de sessão operem sobre a mesma conexão de transporte. Neste modo o modelo prevê estabelecimento, o gerenciamento (controle do fluxo de dados, sincronização dos diálogos, gerenciamento de permissões das partes comunicantes) e encerramento da sessão.
- Serviços no modo não-orientado a conexão, onde a função da camada de sessão é praticamente inexistente. Neste caso, a camada de sessão apenas mapeia endereços de transporte em endereços de sessão.

2.1.1 Conceitos

Antes de se descrever os serviços da camada de sessão, deve-se fazer algumas considerações gerais definindo alguns conceitos utilizados por esta.

- a) *Token*: é um atributo dinâmico de uma conexão de sessão que, quando atribuído a um dos usuários comunicantes, permite a este que faça uso de certos serviços com exclusividade. Os *tokens* definidos são: dados, encerramento de conexão, sincronização secundária e sincronização principal/atividade, sendo que estes são negociados no estabelecimento da conexão.
- b) *Pontos de Sincronização e Diálogo*: Existem dois tipos de ponto de sincronização: principal e secundário. Unidades de diálogo são intervalos de comunicação, delimitados pelos pontos de sincronização principal, que independem das unidades de diálogo anteriores e posteriores, sendo que esta pode ser mapeada em uma função específica de uma aplicação. Cada ponto de sincronização principal define o fim de uma unidade de diálogo e o início de outra, e estes devem ser confirmados explicitamente pelas demais entidades comunicantes. Os pontos de sincronização secundários são utilizados, opcionalmente, durante uma unidade de diálogo, para garantir que os dados enviados anteriormente a este ponto não necessitem ser retransmitidos no caso de uma re-sincronização, aumentando a flexibilidade envolvida na recuperação de falhas. Estes pontos podem, ou não, ser confirmados.
- c) *Atividade*: são unidades lógicas de um trabalho, constituídas de um ou mais diálogos. Num dado instante somente uma atividade é permitida em uma conexão de sessão, porém, diversas atividades consecutivas podem ser transmitidas em uma conexão. O modelo também prevê a possibilidade de uma atividade ser interrompida (para a transmissão de uma atividade mais importante, ou suspensão de uma atividade muito longa) podendo ser retomada posteriormente na mesma conexão ou em conexões subseqüentes a partir de um ponto de sincronização, principal ou secundário, cabendo ao usuário do serviço o armazenamento das informações necessárias ao reinício da atividade.

- d) *Re-sincronização*: consiste em colocar a conexão de sessão em um estado a partir do qual possa ser restabelecido o envio de dados se houver qualquer erro, isto inclui a renegociação dos *tokens* e a definição de um novo ponto de sincronização. A re-sincronização pode ser solicitada por qualquer usuário de sessão. Se a sincronização for simétrica pode se requisitar que um ou ambos os fluxos de dados sejam re-sincronizados, se for assimétrica, ambos os fluxos são sincronizados simultaneamente, pois o ponto de sincronização é único para a conexão de sessão.
- e) *Negociação*: estabelecer as regras entre ambos usuários durante a fase de estabelecimento de conexão. Quando um usuário propuser o uso de uma determinada unidade funcional que necessite de *token(s)* ele também deve propor a distribuição inicial do(s) *token(s)*, ou ele pode propor que um determinado usuário faça a(s) escolha(s).

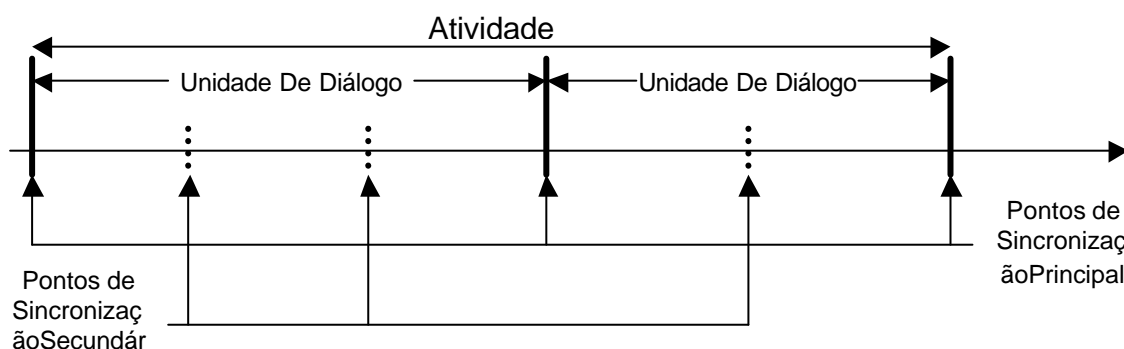


Figura 2.1 - Atividade, Diálogo e Pontos de Sincronização

2.1.2 Fases e Serviços da Camada de Sessão

No modo orientado a conexão o serviço de sessão compreende três fases distintas: estabelecimento da conexão, transferência de dados e encerramento da conexão.

No modo não-orientado a conexão o serviço de sessão simplesmente envia a unidade de dados do usuário de sessão transmissor para o usuário de sessão receptor.

2.1.2.1 Estabelecimento da Conexão

Esta fase prevê o estabelecimento de uma conexão de sessão entre dois usuários da camada de sessão e a negociação cooperativa de *tokens* e parâmetros que definem os serviços que serão usados durante a conexão, são estes parâmetros: o identificador da conexão, a qualidade do serviço, unidades funcionais utilizadas e a distribuição inicial dos *tokens*.

2.1.2.2 Transferência de Dados

Nesta fase tem-se a troca de dados entre os dois usuários de sessão comunicantes. São definidos 4 serviços relacionados a transferência dos dados:

- Transferência de dados normais, que permite a transferência de unidades de dados normais durante a conexão de sessão. O seu uso pode ser controlado por um *token* de dados se na conexão estiver sendo utilizada a unidade funcional de operação no modo *half-duplex*.
- Transferência de dados expressos: que possibilita o envio de dados de tamanhos limitados em uma conexão livre das limitações de gerenciamento de *token* e controle de fluxo. Normalmente é utilizado para o envio de informações de controle quando o transporte de dados está restrito.
- Transferência de dados tipados, que permite o envio de dados tipados independente de disposição e designação de *token* de dados.
- Transferência de dados *capability*, que são usado para fazer a troca confirmada de dados entre usuários enquanto não existir uma atividade em andamento na sessão.

São definidos 3 serviços relacionados ao gerenciamento de *tokens*:

- *give-token*, que é usada quando um usuário explicitamente deseja passar um ou mais *tokens* ao outro usuário comunicante.
- *please-token*, que é usada quando um usuário requisita ao outro usuário comunicante um ou mais *tokens* específicos.
- *give-control*, que é usado para um usuário passar todos os seus *tokens* ao outro usuário.

São definidos 5 serviços associados a sincronização e resincronização da sessão:

- Ponto de sincronização secundário, que é usado para separar um fluxo de dados (normais ou tipados) de outro. Seu uso é controlado pela posse do “*Token de Sincronização Secundária*”.
- Ponto de sincronização simétrico, usado para definir pontos de sincronização secundários sem a posse do *token*.
- Separação de dados, é a funcionalidade que possibilita que os dados enviados antes de um ponto de sincronização secundário sejam descartados no caso de uma resincronização subsequente, evitando a retransmissão de dados já recebidos.
- Ponto de sincronização principal, possibilita a transmissão de um fluxo de dados contínuo (normais, expressos ou tipados) definidos por unidades de diálogo. Seu uso é controlado pelo “*Token de Sincronização Principal/Atividade*”.
- Resincronização, é usado para que a conexão seja restabelecida em um ponto de sincronização anterior, ou um novo ponto de sincronização, permitindo a troca dos *tokens* entre os usuários. Os dados enviados após o último ponto de sincronização podem ser perdidos.

São definidos 5 tipos de serviços relacionados a atividades (sendo estes controlados pelo “*Token de Sincronização Principal/Atividade*”):

- Início de atividade, que é usado para que o usuário explicitamente indique que uma nova atividade é iniciada.
- Retomada de atividade, que indica que uma atividade previamente interrompida é retomada. Controlado pelo “*Token de Sincronização Principal/Atividade*”.
- Interrupção de atividade, que ocasiona a parada anormal de uma atividade com o armazenamento de dados já confirmados (via pontos de sincronização) para que estes não sejam retransmitidos quando a conexão for restabelecida, os dados enviados após o ponto de sincronização são perdidos.
- Descarte de atividade, que também ocasiona a parada anormal de uma atividade, só que todos os dados enviados desde o início da mesma são descartados, e esta não pode ser retomada.

- Fim de atividade, que indica o fim da atividade iniciada, juntamente com o envio de um ponto de sincronização principal.

São definidos 2 serviços de notificação de erros e exceções:

- Notificação de exceção pela entidade provedora do serviço, notifica ao usuário do serviço a ocorrência de uma exceção ou erro de protocolo na entidade fornecedora do serviço.

- Notificação de exceção pelo usuário do serviço, o usuário do serviço pode através deste serviço notificar uma exceção sem possuir o *token* que permite a transmissão de dados.

2.1.2.3 Encerramento da Conexão

O modelo permite que a conexão seja encerrada normalmente por uma das partes comunicantes, podendo este serviço estar restrito ao usuário que detêm o *token* de encerramento de conexão (se este for utilizado). Neste tipo de encerramento existe a garantia de entrega dos dados que ainda estão sendo transmitidos; ou também, a conexão pode ser abortada por qualquer uma das partes comunicantes, tanto usuários como entidades provedoras de serviço, o que implica na perda de dados.

2.1.2.4 Unidades Funcionais da Camada de Sessão

Prevendo que as aplicações não utilizariam todas as funcionalidades de sua camada de sessão a ISO introduziu o conceito de unidades funcionais, sendo estes agrupamentos lógicos de serviços da camada. O uso destas unidades é negociado entre os usuários no estabelecimento da conexão de sessão. Para o uso de um determinado serviço é necessário que o usuário possua um determinado *token*, veja figura 2:

Unidade Funcional	Serviço(s)	Token
<i>Kernel</i> (não negociável)	Estabelecimento de conexão Transferência de dados normal Encerramento ordenado Aborto de conexão iniciada pelo usuário (U-Abort) Aborto de conexão iniciada pelo provedor (P-Abort)	

Encerramento Negociado	Encerramento Negociado <i>Give-token</i> <i>Please-token</i>	encerramento encerramento
Transferência <i>Half-duplex</i>	<i>Give-token</i> <i>Please-token</i>	dados dados
Transferência <i>Full-Duplex</i>	Transferência de dados normal	
Dados expressos	Transferência de dados expressos	
Dados tipados	Transferência de dados tipados	
Dados <i>Capability</i>	Transferência de dados <i>capability</i>	
Sincronização secundária	Ponto de Sincronização secundário <i>Give-token</i> <i>Please-token</i>	Sinc. secundário Sinc. secundário
Sincronização simétrica	Sincronização simétrica	
Separação de dados	<i>Data separation</i> (deve ser associada ao serviço de sincronização secundária ou ao serviço de sincronização simétrica)	
Sincronização principal	Ponto de sincronização principal <i>Give-token</i> <i>Please-token</i>	Sinc. principal Sinc. principal
Ressincronização	Ressincronização	
Exceção	Notificação de exceção pelo provedor Notificação de exceção pelo usuário	
Gerenciamento de atividade	Iniciar atividade Resumir atividade Interromper atividade Descartar atividade Finalizar atividade <i>Give-token</i> <i>Please-token</i> <i>Give-control</i>	atividade atividade atividade atividade atividade atividade atividade

Figura 2.2 - Unidades Funcionais / Serviços de Sessão / Tokens

2.2 O Modelo TCP-IP

Como mencionado anteriormente este foi o primeiro modelo de rede a realmente possibilitar a interconexão de inúmeras redes que utilizavam diferentes tecnologias de enlace de dados.

O TCP/IP, assim como o RM-OSI, também estrutura e especifica seus protocolos em camadas, veja figura 3. Esta arquitetura também é conhecida como pilha de protocolos TCP/IP, que consiste nos seguintes níveis:

- Aplicação: “Nesta camada, estão os protocolos que dão suporte às aplicações dos usuários” (DANTAS, 2002 pg. 112). Estes protocolos especificam os serviços utilizados por estas aplicações como transferência de arquivos, envio de correio eletrônico, acesso remoto, gerência de redes, navegação em hipertexto, entre outros.
- Transporte: Oferece os serviços de transporte confiável dos dados para a camada de aplicação. Podendo ser orientada a conexão, via *Transport Control Protocol (TCP)*, ou não, via *User Datagram Protocol (UDP)*.
- Internet, ou inter-rede: Baseia-se principalmente no *Internet Protocol (IP)* que é responsável pelo envio dos dados da sua origem até seu destino, independente da localização destes. Também oferece protocolos de controle e resolução de endereços.
- Interface de rede: A arquitetura TCP/IP não especifica padrões para as camadas de Enlace e Física, ela somente prevê que a camada de Enlace de Rede deve oferecer uma interface que compatibilize a tecnologia específica da rede com os protocolos da camada internet.

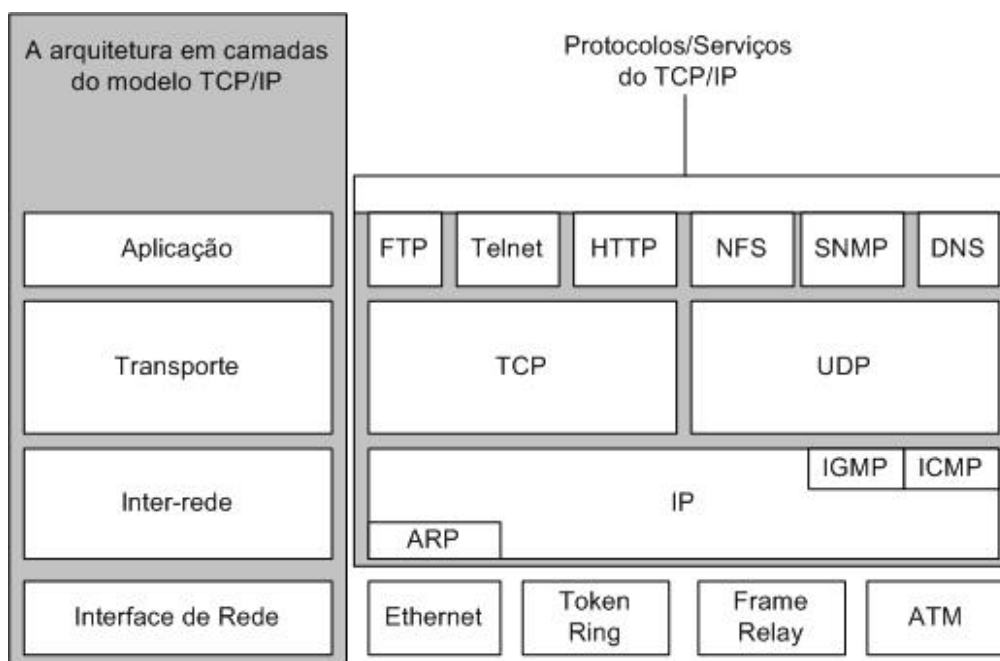


Figura 2.3 - Arquitetura TCP/IP

Os protocolos da arquitetura TCP/IP podem ser modelados e implementados por qualquer pessoa, contanto que eles sejam especificados em documentos denominados *Request for Comments* (RFCs). Uma vez que o protocolo se torne estável o comitê *Internet Activity Board* (IAB) pode torná-lo um padrão Internet.

Aborda-se, resumidamente, somente as camadas de aplicação e de transporte, apenas para ilustrar e justificar outros pontos de maior relevância no presente trabalho.

2.2.1 Camada de Transporte

Como já mencionado, os protocolos desta camada tem o objetivo de fornecer o serviço de transmissão fim-a-fim das mensagens entre os processos de usuário, que são as entidades da camada de aplicação. Desta forma pode-se fazer uso do transporte orientado à conexão, com garantia de entrega das mensagens, ou do transporte não orientado à conexão, onde não existe garantia de entrega das mensagens.

Para efetuar a transmissão orientada à conexão o projeto da, até então ARPANET, desenvolveu o *Transmission Control Protocol* (TCP). Este complexo protocolo toma como princípio que, potencialmente, qualquer pacote enviado pode não chegar a seu destino, ou mesmo eles podem chegar duplicados ou fora da ordem de envio, assim a entidade TCP emissora exige a confirmação de chegada de cada pacote enviado para que pacotes subseqüentes possam ser transmitidos.

“Para permitir que vários processos em um único *host* possam simultaneamente transmitir cadeias de dados, ou seja, possam simultaneamente usar seus serviços, o TCP utiliza o conceito de porta” (SOARES, 1995, pg. 350). Que consiste em associar um identificador numérico a cada processo de aplicação. Desta forma o endereço IP concatenado com o identificador da porta definem um endereço de *socket* TCP/IP.

O envio de datagramas não confirmados, feito pela camada de Inter-rede (protocolo IP), é oferecido à camada de aplicação por meio do UDP (*User Datagram Protocol*) da camada de transporte. A transmissão de mensagens não orientada a conexão tem a visão otimista de que todas as mensagens serão recebidas ordenadamente, não necessitando confirmação nem identificação da ordem de envio.

Quando um protocolo de aplicação estiver fazendo uso do protocolo UDP, a pilha de protocolos utilizada é UDP/IP e não TCP/IP. [...] Devido ao uso extensivo do protocolo TCP, é muito comum o usuário se referir à pilha TCP/IP. Todavia, existem casos em que o TCP não está sendo utilizado (DANTAS, 2002, pg. 122).

A *Application Program Interface* (API) entre a camada de transporte e a camada de aplicação consiste em conjunto de chamadas de sistema que dependem tanto do sistema operacional quanto da linguagem de programação, algumas APIs mais conhecidas são os *BSD Sockets* e a *Transport Layer Interface*³ (Wright, 1995).

2.2.2 Camada de Aplicação

Cada serviço da camada de aplicação é oferecido por um protocolo específico, como por exemplo, a transferência de arquivos utiliza o protocolo *File Transfer Protocol* (FTP), ou os servidores e clientes da WWW utilizam o protocolo HTTP.

Na arquitetura TCP/IP as aplicações são especificadas em RFCs, porém elas são implementadas de forma isolada, ou seja, não existem padrões que definam como deve ser sua estruturação, como no RM-OSI.

As aplicações se comunicam utilizando diretamente a camada de transporte por meio de sua API, sendo a forma mais usual em sistemas UNIX o uso de *BSD Sockets* como apresentado no capítulo 4, mas podendo variar de acordo com o sistema operacional.

Desta forma as implementações das aplicações não possuem um padrão comum de estabelecimento e controle de sessão, pois estas funcionalidades são implementadas por cada aplicação da forma que é mais conveniente a cada uma.

³ *System V TLI*, que é a interface para uso dos protocolos de transporte TCP e UDP no UNIX *System V* para a linguagem C.

3 INET SESP – Protocolo de Sessão para TCP/IP

A seguir será apresentada a estrutura do protocolo SESP. Sendo que todas as definições foram retiradas do projeto de pesquisa, vinculado a tese de doutorado de Dayna Maria Bortoluzzi, em andamento no Programa de Pós-Graduação em Engenharia de Produção da Universidade Federal de Santa Catarina.

É importante parafrasear que essa estrutura, embora tenha sido implementada nesse trabalho de conclusão de curso, pode sofrer alterações em virtude da evolução das pesquisas envolvidas na defesa da tese. Com intuito de estabelecer um controle sobre as versões dos diferentes modelos do projeto foi estabelecido que este primeiro modelo, ou modelo experimental, é definido como SESP versão 0.

3.1 Tipos de serviço

Na definição do protocolo SESP propõe-se a criação de três categorias de serviço de sessão a serem oferecidas:

- serviço confirmado e com conexão: usa os serviços do TCP na camada de transporte;
- *pass thought* TCP: não implementa serviço de sessão, apenas oferece uma interface de passagem direta da aplicação para o protocolo TCP da camada de transporte;
- *pass thought* UDP: não implementa serviço de sessão, apenas oferece uma interface de passagem direta da aplicação para o protocolo UDP da camada de transporte, e também pode ser interpretado como uma analogia ao serviço de sessão não orientado a conexão do RM-OSI.

No modo orientado a conexão o serviço de sessão compreende três fases distintas: estabelecimento da conexão, transferência de dados e encerramento da conexão. A definição de um serviço de sessão confirmado e com conexão visa atender

a demanda de aplicações que precisam de controle de sessão e utilizam o protocolo TCP como mecanismo de transporte.

Os serviços de *pass through* são definidos para atender às aplicações que já possuem implementado o seu próprio controle de sessão. Desta forma, o protocolo de sessão não impõe seus serviços às aplicações que não o desejarem, facilitando a aceitação deste modelo.

3.2 Primitivas de serviço

Do modelo OSI foram retirados seis grupos de primitivas de serviços de sessão: estabelecimento de conexão, liberação de conexão, transferência de dados, gerenciamento de *tokens*, sincronização e gerenciamento de atividades. Dos quais apenas os três primeiros são obrigatórios e não podem ser negociados, e são chamados de núcleo do modelo de sessão.

O quadro mostrado na figura 3.4 apresenta a lista das primitivas de serviço oferecidas pelo protocolo proposto.

PRIMITIVA	FUNÇÃO
Serviços de Sessão	
S_CONNECT_R S_CONNECT_A	inicia (<i>request</i>) conexão de sessão aceita (<i>accept</i>) requisição de conexão
S_RELEASE S_U_ABORT S_P_ABORT	liberação negociada de conexão liberação abrupta (usuário) liberação abrupta (fornecedor)
S_DATA S_EXPEDITED_DATA S_TYPED_DATA	transferência de dados normais transferência de dados urgentes transferência de dados tipados
S_TOKEN_GIVE S_TOKEN_PLEASE S_CONTROL_GIVE	passagem de ficha de dados pedido de ficha passagem de todas as fichas
S_SYNC S_RESYNCHRONIZE	inserção de ponto de sincronização pedido de resincronização
S_ACTIVITY_START S_ACTIVITY_END S_ACTIVITY_DISCARD S_ACTIVITY_INTERRUPT S_ACTIVITY_RESUME	início de uma atividade fim de uma atividade abandono de uma atividade interrupção de uma atividade retomada de uma atividade
Pass-through TCP	
T_OPEN	estabelecimento de conexão TCP

T_CLOSE	liberação de conexão TCP
T_SEND_DATA	envia dados
T_RECEIVE_DATA	recebe dados
T_STATUS	informa sobre o estado da conexão
Pass-through UDP	
T_SEND_DATA	envia dados
T_RECEIVE_DATA	recebe dados

Figura 3.1 - Primitivas de serviço do protocolo de sessão

Os dois primeiros grupos são relacionados, respectivamente, com a inicialização e término das sessões. Os serviços de liberação de sessão podem ser de três tipos: o primeiro, caracterizado pela primitivas S_RELEASE, especificando um serviço confirmado de término negociado de sessão (sem perda de dados); os dois outros, para e liberação abrupta de sessão (com eventual perda de dados), caracterizados pelas primitivas S_U_ABORT e S_P_ABORT, indicando, respectivamente, terminação de iniciativa do usuário (U - *user*) e do fornecedor do serviço (P - *provider*).

O terceiro grupo é caracterizado pelas quatro classes de primitivas para a transferência de dados (S_DATA, S_EXPEDITED_DATA e S_TYPED_DATA) cada uma para um dos três tipos de dados (normais, urgentes e tipados).

O quarto grupo é referente ao gerenciamento de diálogo, contendo as primitivas relativas a passagem da fixa: STOKEN_GIVE, S_TOKEN_PLEASE e S_CONTROL_GIVE.

O quinto grupo contém os serviços relativos à sincronização, para inserção do ponto de sincronização usa-se a primitiva S_SYNC), seja para a resincronização a partir de um ponto dado (S_RESYNCHRONIZE).

O sexto grupo, através dos serviços, S_ACTIVITY_START, S_ACTIVITY_END, S_ACTIVITY_DISCARD, S_ACTIVITY_INTERRUPT e S_ACTIVITY_RESUME é dedicado ao gerenciamento das atividades (início, fim, abandono, interrupção e retomada).

Além das primitivas importadas da camada de sessão do OSI, foram inseridas mais sete primitivas no modelo, utilizadas nos mecanismos de *pass-trought* sobre o TCP e *pass-trought* sobre UDP, que são interpretadas como a chamada direta das rotinas de transporte.

3.3 Definição do cabeçalho

A figura 3.2 mostra o layout atual do cabeçalho do protocolo SESP:

Id_Session	Id_Service	Service Classes	Category	Protocol	Length
------------	------------	-----------------	----------	----------	--------

Figura 3.2 - Cabeçalho da PDU do protocolo de sessão

- **Id_Session (32 bits):** Identificador da sessão. É um valor único definido pela entidade de sessão do servidor, durante o processo de negociação de classes de serviço no estabelecimento de conexão de sessão. Se a sessão estiver inativa, ou seja, ainda não foi negociada, o valor do identificador é 0. O método adequado para definir o valor do identificador ainda não foi estabelecido no protocolo, e no momento é feito através de um contador seqüencial.
- **Id_Service(8 bits):** Identificador da primitiva de serviço. Este campo é preenchido seguindo a relação de primitivas de serviços de sessão, mencionadas na figura 3.1.
- **Service Classes(13 bits):** Um conjunto de *flags* que definem as classes de serviços em negociação. Devem seguir a seguinte orientação de valores:
 - transferência de dados *half-duplex* (sinalizada pelo primeiro bit 0001);
 - transferência de dados com sincronização (sinalizada pelo segundo bit 0010);
 - gerência de atividade (sinalizada pelo terceiro bit 0100);
 - encerramento de conexão negociada (sinalizada pelo quarto bit 1000).
 - os demais bits não são usados no modelo atual e ficam livres para uso em especificações futuras.
- **Id_Category (3 bits):** Identificador da categoria de transporte usada pela sessão. Podendo ser:
 - serviço confirmado com conexão (valor 001);
 - *pass through* TCP (valor 110);
 - *pass through*UDP (111).
- **Id_Protocol (16 bits):** Identifica qual protocolo da camada de aplicação está utilizando o protocolo de sessão. Esse campo permite mapear o tipo de tráfego de acordo com as portas padrões utilizadas pelas aplicações da Internet. Os valores para este campo ainda não foram definidos.
- **Length (32 bits):** Indica o tamanho da mensagem (em bytes) enviada pela aplicação.

3.4 Serviços oferecidos

No modelo do protocolo SESP, foram especificados os serviços de estabelecimento e liberação de conexões, transferência de dados normais, transferência de dados tipados e transferência de dados expressos (urgentes), com a mesma funcionalidade definida na camada de sessão do modelo OSI. Complementarmente foram considerados os serviços adicionais oferecidos pela camada de sessão do modelo OSI: gerenciamento de diálogo, gerência de atividades e sincronização de dados. Além desses serviços, identificou-se a necessidade de incluir a possibilidade de uma aplicação não utilizar nenhuma das facilidades oferecidas pela camada de sessão, acessando diretamente o serviço de transporte TCP ou UDP. Para isso, incluiu-se o serviço de *pass-throught*.

Algumas questões relativas aos serviços ainda não possuem definição concreta, portanto serão assumidas por este trabalho da forma mais simples e conveniente.

3.4.1 Estabelecimento de conexão

Esta fase prevê o estabelecimento de uma conexão de sessão entre dois usuários da camada de sessão e a negociação das classes de serviços que serão na fase de transferência de dados.

Quando o cliente solicita o início de uma conexão de sessão, o protocolo aguarda o estabelecimento da conexão de transporte, para então enviar uma requisição de conexão de sessão, montando uma PDU com a primitiva S_CONNECT_R e informando quais as classes de serviço que deseja utilizar.

O servidor, após aceitar a conexão de transporte, aguarda pela requisição de conexão de sessão. A PDU de requisição é analisada, onde as classes de serviço requisitadas pelo cliente são comparadas com os serviços que o servidor oferece. Um identificador de sessão é gerado e a PDU de resposta (S_CONNECT_R) é enviada ao cliente, contendo as classes de serviços disponíveis e o identificador da sessão.

A utilização de qualquer serviço adicional oferecido pela camada de sessão deve ser negociada no estabelecimento de uma conexão de sessão, com exceção do serviço de *pass-through*.

3.4.2 Transferência de dados

Nessa fase é realizada a troca dos dados entre os dois usuários de sessão comunicantes. São definidos 3 serviços relacionados com a transferência dos dados:

- Transferência de dados normais, que permite a transferência de unidades de dados normais durante a conexão de sessão.
- Transferência de dados expressos, que possibilita o envio de dados, de tamanhos limitados, em uma conexão, livre das limitações de controle de fluxo. Normalmente são utilizados para o envio de informações de controle, quando o transporte de dados está restrito.
- Transferência de dados tipados, que tem a liberdade de transmissão se a categoria de transmissão *half-duplex* estiver ativa, mesmo sem a necessidade de posse do *token* de dados.

3.4.3 Gerência de diálogo

A gerência de diálogo opera nas quatro classes de serviço: transferência de dados *half-duplex*, sincronização de dados, controle de atividade e encerramento negociado.

Quando uma destas classes de serviço está ativa na sessão, a gerência de diálogo define que somente a entidade que possuir o *token* da classe de serviço poderá ter controle sobre ela.

Se um usuário necessita do *token*, ele pode solicitar ao seu interlocutor que lhe ceda o *token*, com o uso da primitiva S_TOKEN_PLEASE.

Se o outro usuário deseja ceder a posse do *token*, ele usa a primitiva de serviço S_TOKEN_GIVE, passando o identificador do *token* como parâmetro.

O controle de diálogo utiliza o mecanismo de passagem de *token* de forma simplificada, por exemplo, quando uma aplicação requer que a transferência de dados seja feita na forma *half-duplex*, ela deve negociar a utilização desta classe de serviço

durante o estabelecimento da conexão, e então o *token* de dados pode, ou não, ser enviado ao usuário que necessitá-lo, dependendo sempre da “vontade” da entidade que o possui.

O mecanismo de gerência de diálogo pode ser utilizado em aplicações multimídia que precisem dessa funcionalidade, como por exemplo, aplicações de videoconferência.

3.4.4 Pontos de sincronismo

O mecanismo de sincronização é feito de forma simplificada, de forma que apenas um ponto de sincronismo seja inserido como fronteira da informação, tendo como princípio o uso apenas dos pontos de sincronização principal definidos pelo RM-OSI.

O ponto de sincronização possibilita a transmissão de um fluxo de dados (normais, expressos ou tipados) contínuo, delimitados por unidades de diálogo.

A re-sincronização é usada para que a conexão seja retomada em um ponto de sincronização anterior, ou seja, todos os dados enviados a partir deste ponto serão retransmitidos.

Uma vez que o servidor receba a confirmação de que o cliente recebeu um novo ponto de sincronismo, os dados enviados anteriormente a este ponto podem, ou não, ser descartados.

3.4.5 Gerência de atividade

Devido ao fato de que o modelo de gerenciamento de múltiplas atividades e as máquinas de estado deste serviço não foram descritas, este trabalho não levará em consideração a troca de mensagens de sessão para o gerenciamento de atividades. Portanto a gerência de atividade não será implementada explicitamente apenas será prevista como trabalho futuro.

3.4.6 Encerramento de uma sessão

O encerramento pode ser feito de maneira negociada ou abrupta. Na forma negociada, somente a entidade detentora do *token* de encerramento pode solicitar o

encerramento da conexão. Já no modo de encerramento abrupto, a iniciativa pode ser de qualquer uma das partes comunicantes, tanto usuários como entidades provedoras de serviço, mas isso pode implicar em perda de dados.

3.5 Adaptação das aplicações existentes

Para que as aplicações já existentes possam operar normalmente sem utilizar os mecanismos de controle de sessão, foram implementados os serviços de *pass through* que encaminham os dados diretamente ao protocolo de transporte desejado.

4 Implementação do TCP/IP

Para ilustrar melhor e para justificar a modelagem do sistema até o presente momento serão esboçados de forma bem resumida os principais aspectos de implementação do TCP/IP no *Berkley Distribution Software*, colocando em foco apenas aqueles que são considerados mais relevantes para o desenvolvimento deste trabalho.

O TCP/IP foi inicialmente implementado no sistema operacional BSD4.2, lançado em 1983, para que as aplicações pudessem utilizar a arquitetura da maneira mais fácil possível os desenvolvedores criaram uma API independente de protocolos. conhecida como *Berkeley Sockets*, ou *BSD Sockets*, a partir da qual as aplicações poderiam utilizar os serviços dos protocolos de diferentes arquiteturas de rede, como TCP/IP, OSI, Xerox *Network Systems* (XNS), etc. Na versão 4.4 do BSD, ou *4.4BSD-Lite*, a implementação do TCP/IP era chamada de *BSD Networking Software*, release 3.0, ou simplesmente *Net/3*, que será tratado desta forma neste trabalho. O sistema original BSD é atualmente continuado e distribuído livremente em diferentes projetos, principalmente *FreeBSD*, *NetBSD*, *OpenBSD* e *BSD/OS*.

A arquitetura TCP/IP é construída seguindo o paradigma Cliente-Servidor, onde um processo, ou entidade de aplicação oferece seus serviços para clientes remotos que eventualmente queiram utilizá-los.

É importante ressaltar que todo o código fonte do BSD, e de seus sucessores, se encontra descrito na linguagem de programação C, e que será tratado neste trabalho como sendo de domínio público e de conhecimento comum.

Nos sistemas operacionais da atualidade, normalmente as aplicações fazem uso das funcionalidades de E/S⁴ das camadas inferiores, por meio de rotinas de sistema (*system calls*). Estas rotinas sempre são chamadas explicitamente e as camadas inferiores não têm acesso direto às funcionalidades da aplicação, ou seja, uma entidade TCP não tem acesso às rotinas das entidades de aplicação. Desta forma, a transmissão de dados é feita de maneira síncrona orientada pela aplicação. Um novo conjunto de rotinas de sistema, definido na edição 1993 do Posix⁵ (PASC, 2003), permite que

⁴ E/S: Entrada e Saída, referenciada também como I/O, do inglês *Input and Output*.

⁵ Portable Operating System Interface: conjunto de padronizações de interfaces utilizadas no desenvolvimento de sistemas operacionais. (PASC, 2003)

aplicações realizem operações de E/S de maneira assíncrona (STEVENS, 1998, p. 148), mas poucos sistemas da atualidade implementam estas rotinas, portanto elas serão desconsideradas no desenvolvimento deste trabalho.

O Net/3 é organizado em 3 camadas distintas: *socket*, protocolos e interface, e 3 conjuntos de filas, conforme ilustrado na figura 4.1. A camada de *sockets* oferece uma interface comum de chamadas de sistema onde as aplicações podem utilizar os serviços de estabelecimento e encerramento de conexão, envio e recebimento de dados, tarefas de controle, etc. A camada de protocolos implementa diferentes famílias de protocolos, XNS, OSI, Unix, TCP/IP, entre outras, que podem ser subdivididos em mais sub-camadas. A camada de interface implementa os protocolos de controle de enlace e drivers específicos para os dispositivos de interface de rede, ou *Network Interface Cards* (NICs).

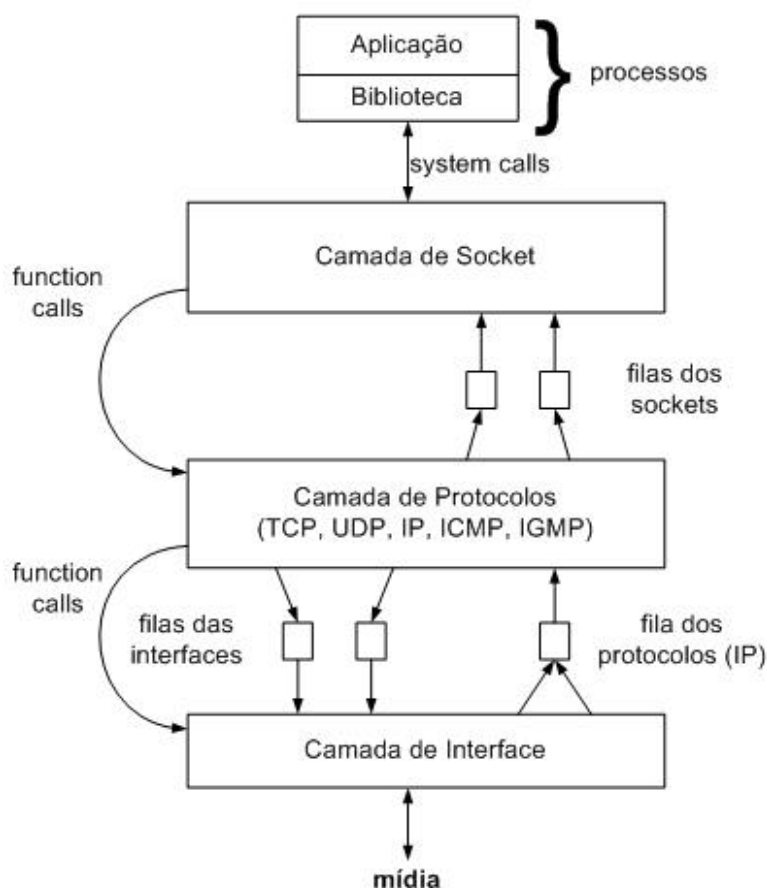


Figura 4.1 – Comunicação entre as camadas na arquitetura do Net/3 (WRIGHT, 1995).

Como um dos objetivos deste trabalho é observar o presente modelo com a finalidade de adicionar o protocolo SESP em seu contexto, analisaremos sucintamente a implementação e a utilização da camada de *socket* e do protocolo TCP.

4.1 Memory Buffers

A peça principal na arquitetura de implementação do *Net/3* é a *memory buffer*, ou *mbuf*, que é uma estrutura de dados definida para armazenar *Protocol Data Units* (PDUs), enquanto estas trafegam internamente nos protocolos do *Net/3*. Ela é utilizada com a finalidade de otimizar, da maneira mais eficiente possível, o uso de recursos computacionais, de forma que os dados possam ser anexados e extraídos de cada PDU, minimizando a quantidade de dados que deva ser copiada por cada procedimento. A estrutura de uma *mbuf* pode ser vista na figura 4.2.

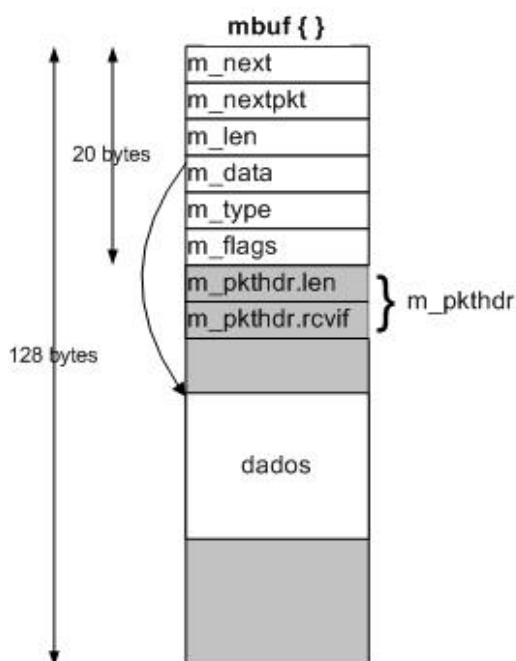


Figura 4.2 – Estrutura da Memory Buffer (WRIGHT, 1995).

As *mbufs* podem ser encadeadas, para que cada protocolo possa efetuar ordenamento, roteamento, entre outras funcionalidades (veja figura 4.3).

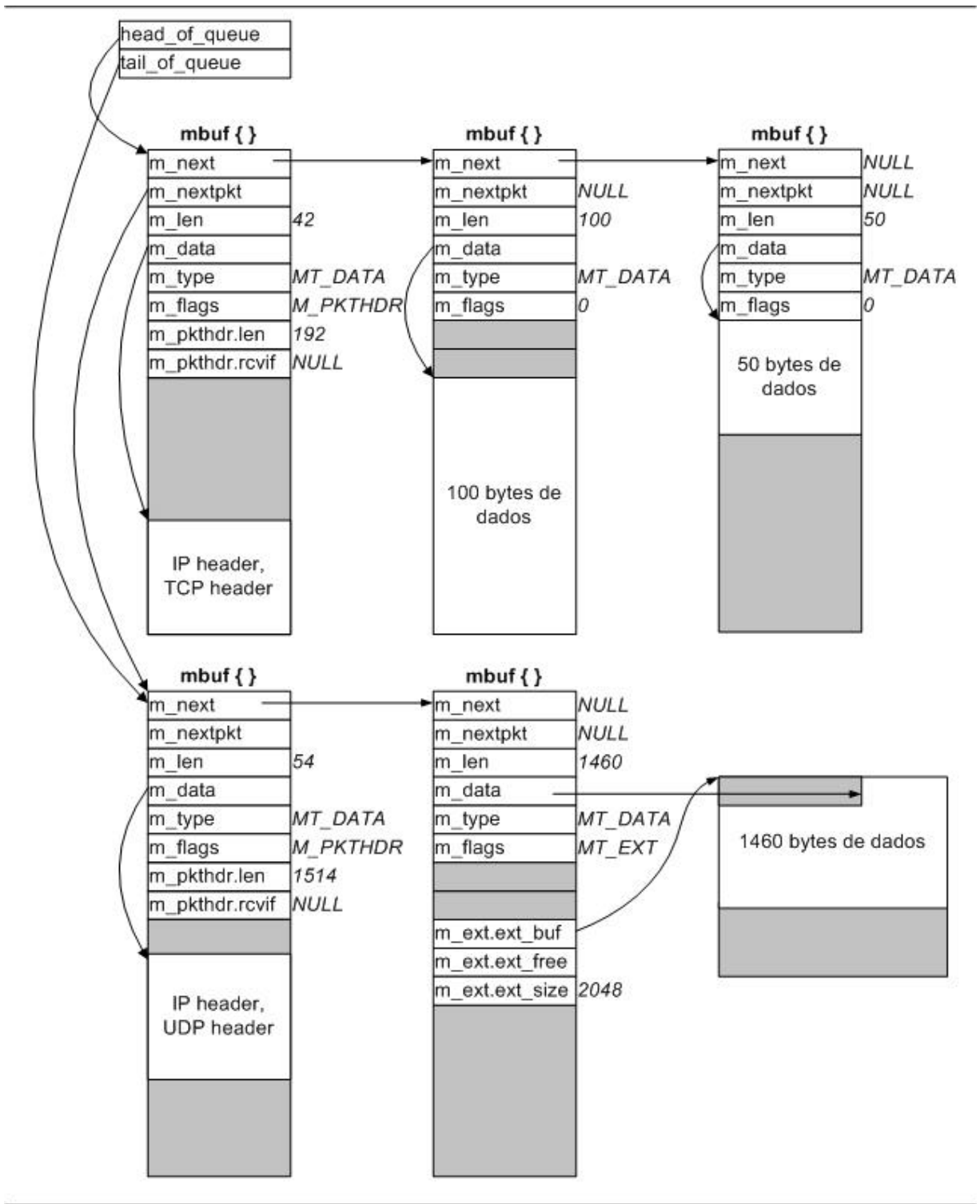


Figura 4.3 – Fila de Memory Buffers (WRIGHT, 1995).

O espaço de dados de uma `mbuf` pode conter 108 bytes, se possuir somente dados de aplicação, ou 100 bytes se possuir um cabeçalho de protocolo. No caso da quantidade de dados exceder o tamanho da estrutura, em até 208 bytes de dados de aplicação, as `mbufs` são aninhadas em cadeias indexadas, onde uma *memory buffer* referencia a próxima, mas todas elas contendo dados do mesmo pacote. Quando o valor dos dados da aplicação for maior do que 208 bytes, então uma das estruturas indicará um *buffer* externo, chamado *cluster*, que pode armazenar até 2048 bytes de dados (veja figura 4.3).

O tipo de dado armazenado em uma estrutura é informado no campo `m_flag`, e estas estruturas são divididas em quatro tipos, de acordo com o tipo de dado que contém:

- buffer de dados, `m_flag = 0`, contendo de 0 a 108 bytes de dados;
- somente cabeçalho de pacote, `m_flag = M_PKTHDR`, sendo a primeira `m_buf` que descreve um pacote de dados e pode conter de 0 a 100 bytes de cabeçalhos de pacotes/datagramas (ex. cabeçalho TCP + cabeçalho IP);
- somente cluster de dados, `m_flag = M_EXT`, usado para indicar um buffer externo chamado cluster de dados, podendo conter de 208 a 2048 bytes de dados;
- cabeçalho de pacote e cluster de dados, `m_flag = M_PKTHDR | M_EXT`, contem um buffer externo no qual também estão os cabeçalhos de protocolos.

Uma vez que os dados da aplicação foram encapsulados e alocados em memória em uma cadeia de *memory buffers*, isto feito pela camada de *socket* ou pela de interface, a única informação que deverá trafegar entre as demais camadas e protocolos são ponteiros para a localização da estrutura. Os demais dados, relativos aos cabeçalhos de cada protocolo, deverão apenas ser anexados ou removidos do primeiro `mbuf` da cadeia, cujo `m_flag` é definido como `M_PKTHDR` (veja figura 4.3).

Neste trabalho a estrutura de *memory buffers* foi abstraída em um modelo mais simples, apenas utilizando conceitos que auxiliassem a implementação do protocolo, o que será visto nos capítulos seguintes.

4.2 Camada de Socket

A API de BSD *sockets* é um sistema de comunicação de processos que consiste em abstrair o sistema de arquivos convencional para a comunicação de dados via rede. Nestes sistemas, um *socket* é representado por um descritor de arquivos; porém os dados escritos por meio deste descritor não são alocados em disco, mas são enviados para uma entidade par, representada por um outro descritor de arquivos, local ou remoto, pelo qual os dados podem ser lidos.

A integração entre as *system calls* do sistema de arquivos para a API de *sockets* permite que cada *socket* seja tratado como um arquivo comum do sistema, que é aberto, lido e escrito da forma que mais convém ao processo usuário, além de oferecer rotinas próprias para tratamento e configuração da comunicação via rede.

Para garantir a entrega das mensagens no *socket* destino abaixo da camada de *sockets* são implementados protocolos de comunicação, e estes protocolos são agrupados em diferentes domínios:

- UNIX – *Unix domain socket*
- INET – Internet via TCP/IP
- AX25 – Radio amador X25
- IPX – Novell IPX
- APPLETALK – Appletalk DDP
- X25 – X25

```

<sys/socket.h>

/*
 * Structure used by kernel to store most
 * addresses.
 */
struct sockaddr {
    u_char          sa_len;          /* total length */
    sa_family_t    sa_family;      /* address family */
    char           sa_data[14];    /* actually longer; address value */
};

```

<sys/socket.h>

Figura 4.4 – A estrutura de endereçamento genérica *sockaddr* (código fonte do FreeBSD 4.8)

Cada domínio de protocolos constitui uma família de endereçamento distinta, onde todas as famílias descendem de uma estrutura genérica de endereçamento de *socket* chamada `sockaddr`: A família de endereçamento da arquitetura TCP/IP (domínio INET) é representada pela estrutura `sockaddr_in`:

```

<netinet/in.h>
/*
 * Socket address, internet style.
 */
struct sockaddr_in {
    u_char    sin_len;
    u_char    sin_family;
    u_short   sin_port;
    struct    in_addr sin_addr;
    char      sin_zero[8];
};

```

<netinet/in.h>

Figura 4.5 – A estrutura de endereçamento *socket* para Internet IPv4.

Como os dados que vêm das camadas inferiores à camada de *socket* são enviados assíncronamente para as camadas superiores, cada *socket* armazena os dados, vindos da entidade par, em *buffers*, como ilustrado na figura 4.1. Desta forma, o recebimento de dados fica orientado às requisições do processo usuário, como será visto adiante.

4.2.1 Socket System Calls

Como já mencionado, na introdução deste capítulo, o uso de *sockets* se dá através de um conjunto definido de rotinas de sistema, inclusive sendo algumas destas comuns ao sistema de arquivos. As principais rotinas de sistema que operam na interface de *sockets* são ilustradas resumidamente na figura 4.6.

Categoria	Nome	Função
Configuração	<code>socket()</code>	Cria um novo <code>socket</code> , sem endereçamento, que operará em um determinado domínio de comunicação, representado pela família de endereçamento conferida ao <code>socket</code> .
	<code>bind()</code>	Designa um endereço (IP e porta, no caso do INET) ao <code>socket</code> .
Servidor	<code>listen()</code>	Informa ao <code>socket</code> para entrar em estado de "escuta", preparando-o para receber conexões.
	<code>accept()</code>	Aguarda e aceita conexões externas.
Cliente	<code>connect()</code>	Estabelece uma conexão com um <code>socket</code> remoto.
Entrada	<code>read()</code>	Recebe os dados em um único <code>buffer</code> .
	<code>readv()</code>	Rebebe os dados em múltiplos <code>buffers</code> .
	<code>recv()</code>	Recebe os dados, similarmente a <code>read()</code> , só que permite especificar opções de recebimento.
	<code>recvfrom()</code>	Recebe os dados e o endereço do remetente.
	<code>recvmsg()</code>	Recebe os dados em múltiplos <code>buffers</code> , com controle da informação, endereço do remetente, e permite especificar opções de recebimento.
Saída	<code>write()</code>	Envia os dados de um único <code>buffer</code> .
	<code>writerv()</code>	Envia os dados de múltiplos <code>buffers</code> .
	<code>send()</code>	Envia os dados permitindo a especificação de opções de envio.
	<code>sendto()</code>	Envia os dados para um endereço específico.
	<code>sendmsg()</code>	Envia os dados em múltiplos <code>buffers</code> , com controle da informação, endereço do remetente, e permite especificar opções de envio.
Condição	<code>select()</code>	Espera por condições de E/S.
Encerramento	<code>shutdown()</code>	Encerra a conexão em uma ou ambas as direções.
	<code>close()</code>	Encerra a conexão e libera o <code>socket</code> .
Administração	<code>fcntl()</code>	Modifica a semântica de E/S.
	<code>Ioctl()</code>	Altera diferente opções do <code>socket</code> .
	<code>setsockopt()</code>	Define opções sobre o <code>socket</code> ou os protocolos.
	<code>getsockopt()</code>	Informa as opções definidas sobre o <code>socket</code> ou os protocolos.
	<code>getsockname()</code>	Recebe o endereço designado ao <code>socket</code> local.
<code>getpeername()</code>	Recebe o endereço designado ao <code>socket</code> remoto.	

Figura 4.6 – Chamadas de Sistema para Comunicação em Redes no Net/3 (WRIGHT, 1995, p. 445)

Resumidamente, para que a conexão TCP possa utilizar a API de `socket`, ela deve seguir o seguinte procedimento:

- Lado Servidor:

1º. Cria-se um `socket` através da rotina `socket()`, onde serão especificados a família de endereçamento e o tipo do protocolo que

se deseja utilizar. Através desta rotina, o `socket` criado será ligado a uma estrutura de controle do protocolo desejado;

- 2º. Usa-se a rotina `bind()` para associar um endereço IP e uma porta (TCP ou UDP) ao `socket`;
 - 3º. Coloca-se a conexão inativa, representada pelo `socket`, em estado de escuta com o uso da rotina `listen()`;
 - 4º. Através da rotina `accept()` bloqueia-se o processo associado ao `socket` (se ele estiver em modo de operação “`nonblock = false`”), até que uma conexão seja estabelecida com sucesso, neste caso retornando um novo `socket` que será usado para o envio e recepção dos dados;
 - 5º. Inicia-se o processo de envio e recepção de dados com o uso das rotinas de entrada e saída ilustradas na figura 4.6;
- Lado Cliente:
 - 1º. Cria-se um `socket` da mesma forma que no lado servidor, onde a família de endereçamento e o tipo de protocolo devem ser os mesmos da entidade com a qual se deseja conectar;
 - 2º. Através da rotina `connect()` se conduz o procedimento de estabelecimento de conexão, TCP three-way handshake (STEVENS, 1998, p. 34), com a entidade remota especificada pelo endereço fornecido a rotina.
 - 3º. Inicia-se o processo de envio e recepção de dados com o uso das rotinas de entrada e saída da API de `socket`;
 - 4º. Após o término da transmissão, encerra-se a conexão, chamando-se a rotina `close()`, que seguirá com o procedimento de encerramento de conexão confirmado, e removerá o descritor do `socket` da tabela de arquivos usados pelo processo.

Um `socket` pode operar em dois modos de E/S: *blocked*, ou *nonblocked*; isto determina que quando o processo requisitar operações de recepção de dados/conexão (através das rotinas `accept()`, `select()`, `recv()` e demais procedimentos de

entrada), ele poderá ou não ser bloqueado até que hajam dados na fila de recepção do `socket`, ou conexões recentemente estabelecidas.

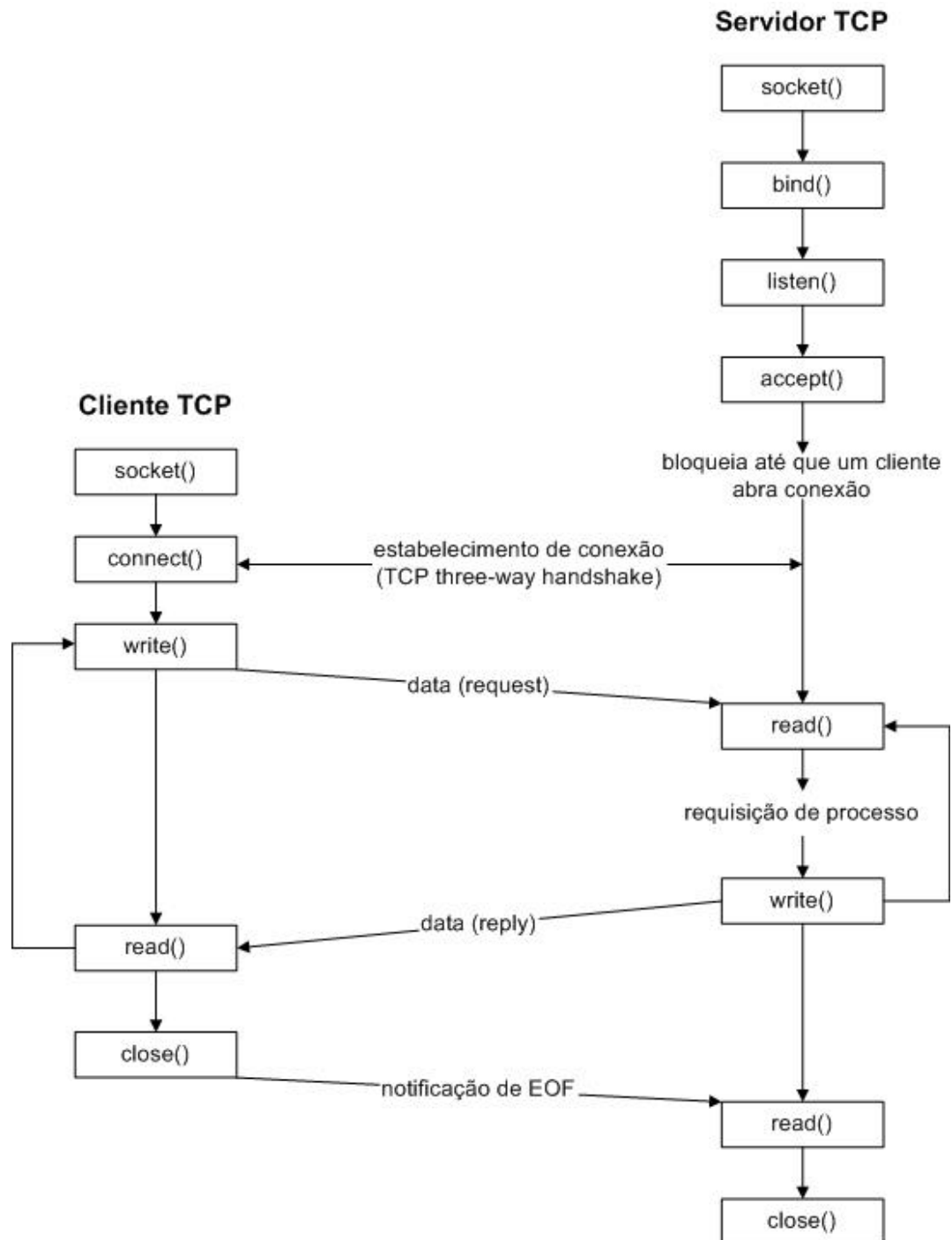


Figura 4.7 – Uso de *sockets* para uma aplicação cliente-servidor simples (STEVENS, 1998, p. 86)

4.3 Camada de Protocolos

A camada de protocolos, analogamente ao RM-OSI, consiste em um ou mais protocolos dispostos em camadas, cada um oferecendo seus serviços aos processos usuários ou às entidades dos protocolos superiores e usando os serviços dos protocolos inferiores. Cada protocolo oferece cinco principais pontos de acesso às demais entidades:

- `pr_output`: recebe e trata dos dados vindo de entidades de nível superior;
- `pr_ctloutput`: trata de informações de controle dos protocolos de nível superior;
- `pr_inputI`: recebe e trata dos dados vindos de entidades de nível superior;
- `pr_ctlinput`: trata de informações de controle dos protocolos de nível inferior;
- `pr_usrreqs`: que trata de toda e qualquer requisição de processos, sendo esta a interface utilizada pelas chamadas da camada de sockets.

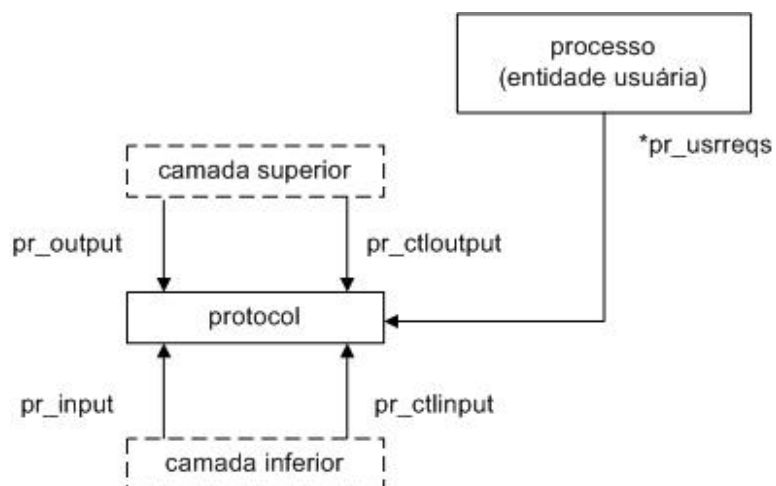


Figura 4.8 – Pontos de acesso a protocolos (WRIGHT, 1995, p. 190)

Outras rotinas auxiliares também são definidas por cada protocolo, como `pr_init`, `pr_sysctl` e `pr_flags`, mas este trabalho apenas enfocará as rotinas de interface.

```

"protosw.h"

struct protosw {
    short    pr_type;                /* socket type used for */
    struct   domain *pr_domain;      /* domain protocol a member of */
    short    pr_protocol;           /* protocol number */
    short    pr_flags;              /* see below */
    /* protocol-protocol hooks */
    void     (*pr_input) ();         /* input to protocol (from below) */
    int      (*pr_output) ();        /* output to protocol (from above) */
    void     (*pr_ctlinput) ();      /* control input (from below) */
    int      (*pr_ctloutput) ();     /* control output (from above) */
    /* user-protocol hook */
    void     *pr_ousrreq;
    /* utility hooks */
    void     (*pr_init) ();          /* initialization hook */
    void     (*pr_fasttimo) ();      /* fast timeout (200ms) */
    void     (*pr_slowtimo) ();      /* slow timeout (500ms) */
    void     (*pr_drain) ();         /* flush any excess space possible */
    struct   pr_usrreqs *pr_usrreqs; /* supersedes pr_usrreq() */
};

[...]
```

```

struct pr_usrreqs {
    int      (*pru_abort) ();
    int      (*pru_accept) ();
    int      (*pru_attach) ();
    int      (*pru_bind) ();
    int      (*pru_connect) ();
    int      (*pru_connect2) ();
    int      (*pru_control) ();
    int      (*pru_detach) ();
    int      (*pru_disconnect) ();
    int      (*pru_listen) ();
    int      (*pru_peeraddr) ();
    int      (*pru_rcvd) ();
    int      (*pru_rcvoob) ();
    int      (*pru_send) ();
    int      (*pru_sense) ();
    int      (*pru_shutdown) ();
    int      (*pru_sockaddr) ();
};

```

"protosw.h"

Figura 4.9 – As estruturas *protosw* e *pr_usrreqs* do BSD. (código fonte do FreeBSD 4.8)

Durante a inicialização do sistema os domínios e protocolos são registrados junto ao mesmo, para informar de que forma suas rotinas poderão ser acessadas quando necessárias.

Como a camada de sockets opera sob uma diversidade de famílias de endereçamento e de protocolos, todos os protocolos operam seguindo uma mesma estrutura de interface chamada de `protosw`, que define um conjunto de rotinas que todos os protocolos devem oferecer para as camadas/protocolos superiores (veja figura 4.9 que ilustra parte do código fonte do FreeBSD 4.8).

```

"__in_proto.h"

struct ipprotosw inetsw[] = {

  {}, {}, /* inetsw[0] = IP e inetsw[1] = UDP (...) */

  { SOCK_STREAM,
    &inetdomain,
    IPPROTO_TCP,
    PR_CONNREQUIRED|PR_IMPLOPCL|PR_WANTRCVD,
    tcp_input,
    0,
    tcp_ctlinput,
    tcp_ctloutput,
    0,
    tcp_init,
    0,
    tcp_slowtimo,
    tcp_drain,
    &tcp_usrreqs
  },

  /* abaixo seguem demais protocolos (RAW, ICMP, IGMP, etc...) */

};

```

"__in_proto.h"

Figura 4.10 – Variável de sistema `inetsw` – exemplo de definição da estrutura `protosw`.

Desta forma, cada domínio deve utilizar a estrutura `protosw` para identificar-se junto a camada de `socket`, informando quais as rotinas reais implementadas por cada protocolo. Como por exemplo temos na matriz `inetsw` definida pelo domínio INET os diversos protocolos desta família, em `inetsw[2]` o domínio define as entradas do protocolo TCP (veja figura 4.10), onde somente as rotinas informadas são implementadas pelo protocolo e oferecidas ao restante do sistema. Os valores de `protosw` preenchidos com “0” não são implementados pelo protocolo.

Dentre as rotinas de protocolo deve-se destacar a estrutura `pr_usrreqs`, que como mencionado anteriormente, agrega um conjunto de ponteiros para funções, são estas funções que respondem por requisições de processos usuários. Portanto, a estrutura `pr_usrreqs` é a interface comum que as rotinas da camada de *socket* buscam para acessar os serviços dos protocolos. Cada protocolo irá definir em sua implementação uma estrutura `pr_usrreqs` que indicará os nomes reais das rotinas a serem apontadas, como será apresentado no capítulo seguinte.

4.3.1 Interface do Protocolo TCP

Como visto na seção anterior, para que o TCP possa operar adequadamente dentro da arquitetura de implementação *Net/3*, ele precisa declarar suas interfaces para que os demais protocolos e, principalmente, para que a camada de *sockets* possa utilizá-la.

Durante a inicialização do domínio INET o *array* `inetsw` é montado (veja figura 4.10). Portanto, em `inetsw[2]` o protocolo TCP é declarado juntamente com sua interface implementada, e o mais importante é que a variável `tcp_usrreqs` contém os endereços reais das rotinas implementadas pelo protocolo que serão utilizadas pela camada de *socket* (veja figura 4.11).

```

_____  

                                     "tcp_usrreq.c"  

struct pr_usrreqs tcp_usrreqs = {  

tcp_usr_abort, tcp_usr_accept, tcp_usr_attach, tcp_usr_bind, tcp_usr_connect, pru_connect2_notstupp, in_control,  

tcp_usr_detach, tcp_usr_disconnect, tcp_usr_listen, in_setpeeraddr, tcp_usr_rcvd, tcp_usr_rcvoob, tcp_usr_send,  

pru_sense_null, tcp_usr_shutdown, in_setsockaddr, sosend, soreceive, sopoll  

};  

_____  

                                     "tcp_usrreq.c"

```

Figura 4.11 – Declaração de `tcp_usrreqs` (código fonte do FreeBSD 4.8)

Na figura 4.8 foram ilustrados os cinco principais pontos de acesso de um protocolo, que são usados pelos outros protocolos/processos. No caso do protocolo TCP, os pontos de acesso inferiores são usados pelos protocolos IP e ICMP⁶ e os pontos de acesso superiores pelos processos usuários por meio da camada de *socket*.

⁶ Internet Control Message Protocol – protocolo de troca/levantamento de informações estatísticas.

A figura 4.12 ilustra a forma com que as cinco principais rotinas da interface do protocolo TCP são usadas pelas demais entidades dentro do núcleo do sistema. Pode-se observar que as chamadas a *socket*, feitas por processos usuários, sempre são transmitidas ao TCP pelo uso das rotinas descritas em *tcp_usrreqs*. A rotina administrativa *tcp_ctloutput* é usada para as alterações de propriedades ou requisições de informações do protocolo.

É importante comentar que quando dados são recebidos na interface de redes, uma interrupção aciona o protocolo IP que verifica o destino deles, se o destino for um dos endereços da máquina local então os mesmos são encaminhados ao protocolo TCP para ordenação e posteriormente entregues ao *socket* apropriado, armazenando na sua fila de recepção. Se o *socket* este estiver operando no modo *blocked* o procedimento *tcp_input* então irá solicitar ao gerenciador de processos que desbloqueie o(s) processo(s) bloqueados, que aguardavam operações de E/S no *socket*.

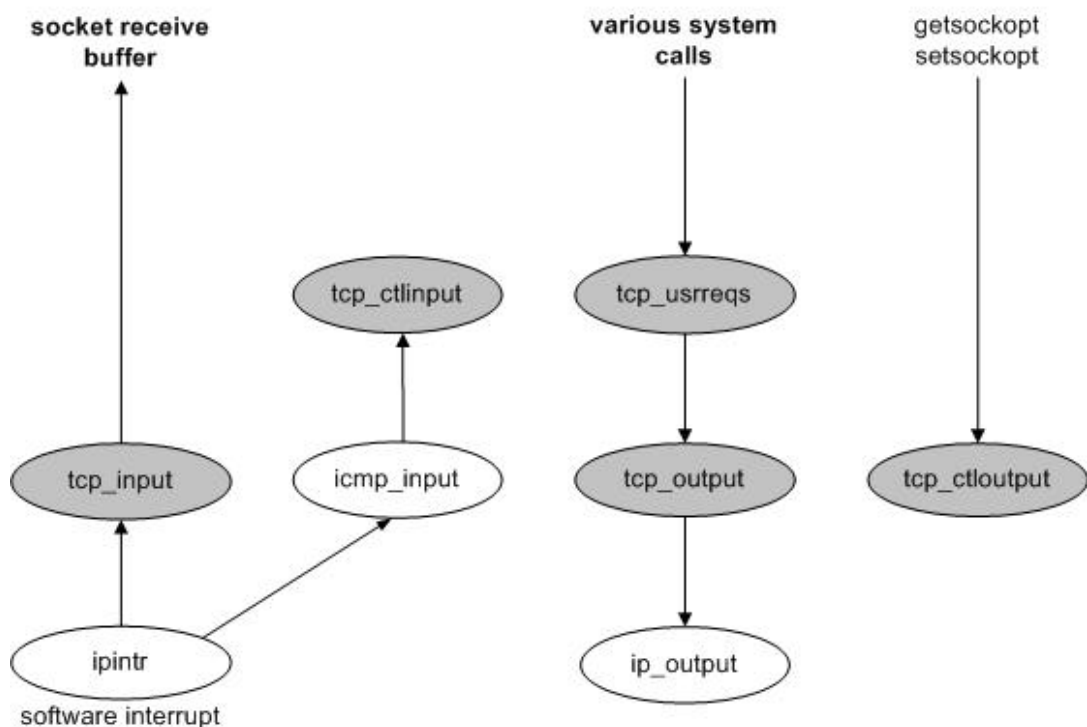


Figura 4.12 – Uso das principais rotinas TCP dentro do núcleo (WRIGHT, 1995, p. 796)

Como já foi citados na seção 4.1, todos os dados de aplicações que trafegam na arquitetura *Net/3* são encapsulados em *memory buffers* que armazenam estes dados juntamente com os cabeçalhos dos protocolos utilizados. Em alguns casos, estas estruturas também podem conter parâmetros de recepção/envio para estes dados. Como exemplo, pode-se citar a rotina `pr_usrreqs.pru_send()`, definida no arquivo "*protosw.h*", que possui o parâmetro `control`, que não possui dados de aplicação, apenas informações para controle do envio dos dados:

```
int(*pru_send) __P((struct socket *so, int flags, struct mbuf
                    *m, struct sockaddr *addr, struct mbuf
                    *control, struct proc *p));
```

5 Implementação do SESP

Antes que o protocolo SESP fosse efetivamente implementado, diferentes modelos de implementação foram idealizados, para que fosse visualizado de que forma o SESP poderia se tornar um *framework* para as aplicações atuais, operando acima ou abaixo da API de BSD *Sockets*. O processo de modelagem teve como guia o modelo principal de implementação do TCP/IP, o *Net/3*.

Após o desenvolvimento dos modelos foi implementado o primeiro protótipo do SESP, que será abordado na seção 5.2.

5.1 Modelos de implementação do SESP

No primeiro modelo, o protocolo SESP é visto integrado ao *Net/3*, operando na camada de protocolos fazendo uso das funcionalidades dos protocolos de transporte e oferecendo seus serviços para as aplicações usuárias, por meio da interface de *socket*.

O segundo modelo foi desenvolvido para validar o protocolo ao nível de usuário, através da implementação de uma biblioteca para programas aplicativos que necessitem de serviços de sessão.

Salienta-se que a principal finalidade do projeto de desenvolvimento do SESP é a sua implementação dentro da arquitetura real do TCP/IP (*Net/3*), ou seja, dentro do núcleo de um sistema operacional que descenda do 4.4 BSD-Lite.

Para facilitar a portabilidade futura no *kernel*, o segundo modelo de implementação foi idealizado como uma abstração das camadas do *Net/3*, de uma forma similar e simplificada. O objetivo é que a interface da API de *Sockets* não seja alterada drasticamente, não se permitindo a remoção ou alteração de rotinas já usuais, mas apenas a adição de rotinas novas, chamadas de “rotinas de sessão”.

5.1.1 Rotinas de sessão

Como o protocolo SESP oferece serviços adicionais aos de simples envio/recepção de dados via *socket*, conforme Posix, definiu-se um conjunto adicional de rotinas, que serão integradas à interface de *socket*. São elas:

- `ssize_t send_typed (int, const void *, size_t, int)`

- Descrição: Semelhante ao procedimento `send()`; envia dados em conexão *half-duplex* sem a posse do token de dados.
- Parâmetros: *file descriptor do socket*, buffer de dados a serem enviados, número de bytes a serem enviados, flags de envio;
- Retorno: número de bytes enviados se houver sucesso, -1 se houver erro de transmissão;
- `ssize_t send_exp (int, const void *, size_t, int)`
 - Descrição: Semelhante ao procedimento `send()`; envia dados com prioridade de transporte e prioridade em filas de envio/recepção.
 - Parâmetros: *file descriptor do socket*, buffer de dados a serem enviados, número de bytes a serem enviados, flags de envio;
 - Retorno: número de bytes enviados se houver sucesso, -1 se houver erro de transmissão;
- `int token_please (int, u_short)`
 - Descrição: Solicitação de um determinado *token* ao par comunicante.
 - Parâmetros: *file descriptor do socket*, identificador do *token*.
 - Retorno: 0 se solicitação enviada, -1 se houver erro de transmissão.
- `int token_give (int, u_short)`
 - Descrição: Envio de um determinado *token* ao par comunicante.
 - Parâmetros: *file descriptor do socket*, identificador do *token*.
 - Retorno: 0 se solicitação enviada, -1 se houver erro de transmissão.
- `int control_give (int)`
 - Descrição: Envio de todos os *tokens* da sessão ao par comunicante.
 - Parâmetros: *file descriptor do socket*.
 - Retorno: 0 se solicitação enviada, -1 se houver erro de transmissão.
- `int sync (int)`
 - Descrição: Cria e envia novo ponto de sincronização a entidade par.
 - Parâmetros: *file descriptor do socket*.
 - Retorno: 0 se sincronização efetuada com sucesso, -1 se houver erro de transmissão.
- `int resync (int)`

- Descrição: Solicita reenvio dos dados enviados pela entidade par desde o ultimo ponto de sincronização.
- Parâmetros: *file descriptor* do *socket*.
- Retorno: 0 se solicitação enviada com sucesso, -1 se houver erro de transmissão.
- `int recv_token (int, int *,int *)`
 - Descrição: Verifica o recebimento de mensagens de solicitação/envio de *token*, retornando a primitiva de serviço da mensagem e o identificador do *token* nos dois parâmetros referenciados por ponteiro.
 - Parâmetros: *file descriptor* do *socket*, primitiva de serviço recebida (S_TOKEN_GIVE, S_TOKEN_PLEASE, S_CONTROL_GIVE), identificador do *token* solicitado/enviado.
 - Retorno: valor maior que 0 se houver mensagem, 0 se não houver mensagem, -1 se houver erro.
- `int recv_sync (int, int *)`
 - Descrição: Verifica o recebimento de um novo ponto de sincronização enviado pela entidade par.
 - Parâmetros: *file descriptor* do *socket*, valor do ponto de sincronização.
 - Retorno: valor maior que 0 se houver novo ponto de sincronização, 0 se não houver, -1 se houver erro.
- `int actvt_start (int)`
 - Descrição: Envia para a entidade par um sinal de inicio de atividade.
 - Parâmetros: *file descriptor* do *socket*.
 - Retorno: identificador da atividade (valor inteiro positivo) se o sinal for enviado com sucesso, -1 se houver erro.
- `int actvt_end (int, int)`
 - Descrição: Envia para a entidade par um sinal de fim de atividade.
 - Parâmetros: *file descriptor* do *socket*, identificador da atividade.
 - Retorno: 0 se o sinal for enviado com sucesso, -1 se houver erro.

- `int actvt_discard (int, int)`
 - Descrição: Envia para a entidade por um sinal de cancelamento de atividade.
 - Parâmetros: *file descriptor* do *socket*, identificador da atividade.
 - Retorno: 0 se o sinal for enviado com sucesso, -1 se houver erro.
- `int actvt_interrupt (int, int)`
 - Descrição: Envia para a entidade por um sinal de interrupção temporária de atividade.
 - Parâmetros: *file descriptor* do *socket*, identificador da atividade.
 - Retorno: 0 se o sinal for enviado com sucesso, -1 se houver erro.
- `int actvt_resume (int, int)`
 - Descrição: Envia para a entidade por um sinal de reinício da atividade.
 - Parâmetros: *file descriptor* do *socket*, identificador da atividade.
 - Retorno: 0 se o sinal for enviado com sucesso, -1 se houver erro.
- `int recv_actvt (int, int *, int *)`
 - Descrição: Verifica o recebimento de mensagens de controle de atividade, retornando a primitiva de serviço da mensagem e o identificador da atividade nos dois parâmetros referenciados por ponteiro.
 - Parâmetros: *file descriptor* do *socket*, primitiva de serviço recebida (`S_ACTIVITY_STAT`, `S_ACTIVITY_END`, `S_ACTIVITY_DISCARD`, `S_ACTIVITY_INTERRUPT` ou `S_ACTIVITY_RESUME`), identificador da atividade.
 - Retorno: valor maior que 0 se houver mensagem, 0 se não houver mensagem, -1 se houver erro.

Se estas rotinas forem oferecidas na camada de *socket* elas deverão ser requisitadas ao protocolo de sessão através de sua interface `pr_usrreqs` onde serão passadas como parâmetros a rotina `pru_send` para seu tratamento e envio. Isto se deve ao fato de que a interface entre protocolo e camada de *socket* deve seguir o padrão de especificação definida pela estrutura `protosw`.

5.1.2 Modelo de implementação integrado ao Net/3

Da mesma forma que a implementação do TCP, o SESP deverá oferecer cinco principais pontos de acesso: `sesp_input`, `sesp_output`, `sesp_ctlinput`, `sesp_ctloutput` e `sesp_usrreqs`. Onde o uso das primitivas de serviço descritas em `sesp_usrreqs` deverá funcionar de forma análoga ao das `tcp_usrreqs` pela API `socket`, veja figura 5.1.

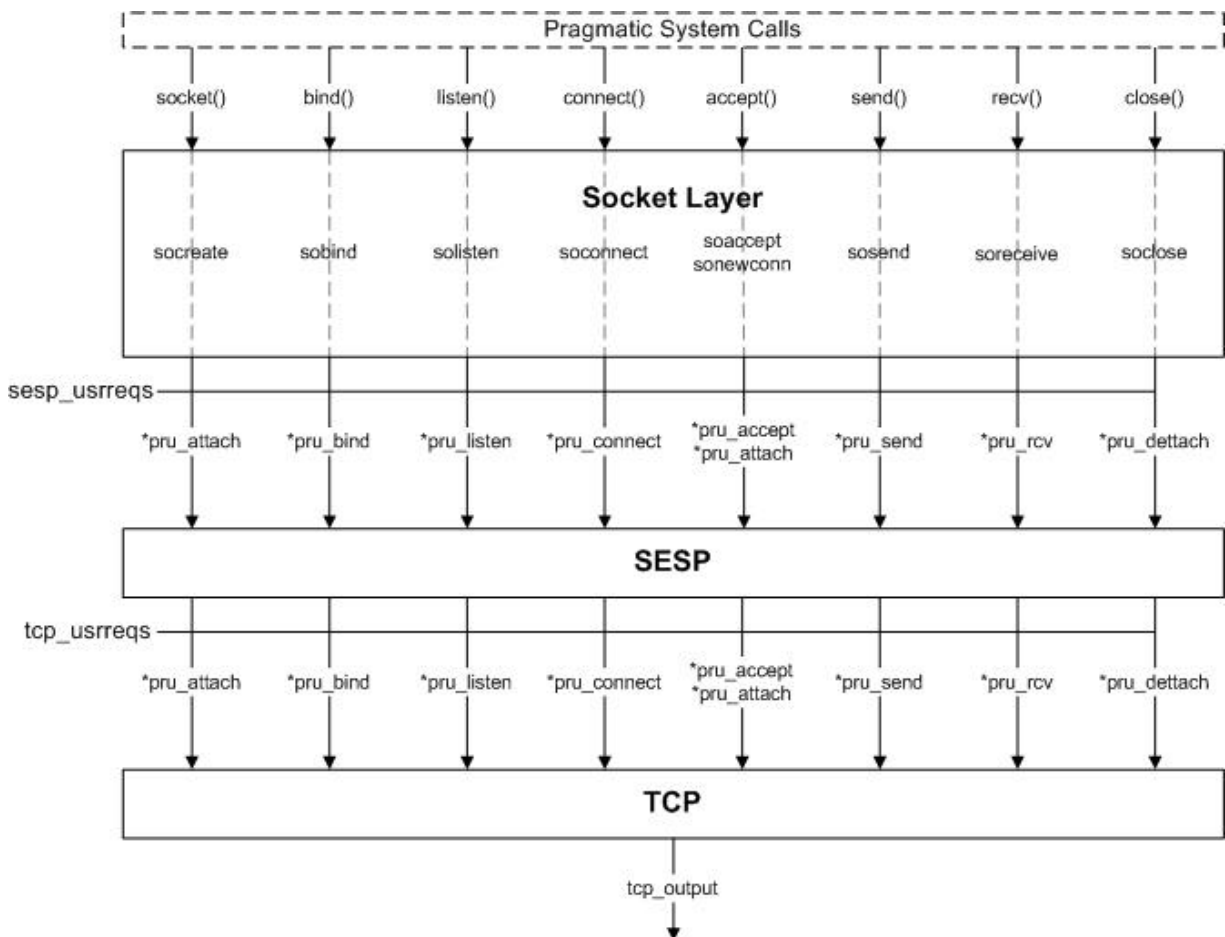


Figura 5.1 - Modelo de fluxo das requisições de usuário ao SESP no kernel do BSD

Uma das principais diferenças que será sentida pela aplicação com uso de um protocolo de sessão abaixo da camada de `socket`, será o fato de que desta forma o `socket` deve persistir enquanto a sessão estiver viva, só podendo ser encerrado a

partir do momento em que a sessão for encerrada normalmente, através do serviço de sessão S_RELEASE, ou for abortada por um dos processos comunicantes, através dos serviços S_P_ABORT ou S_U_ABORT; ao contrário do modelo atual que estipula que o `socket` é fechado a partir do momento que a conexão de transporte é encerrada.

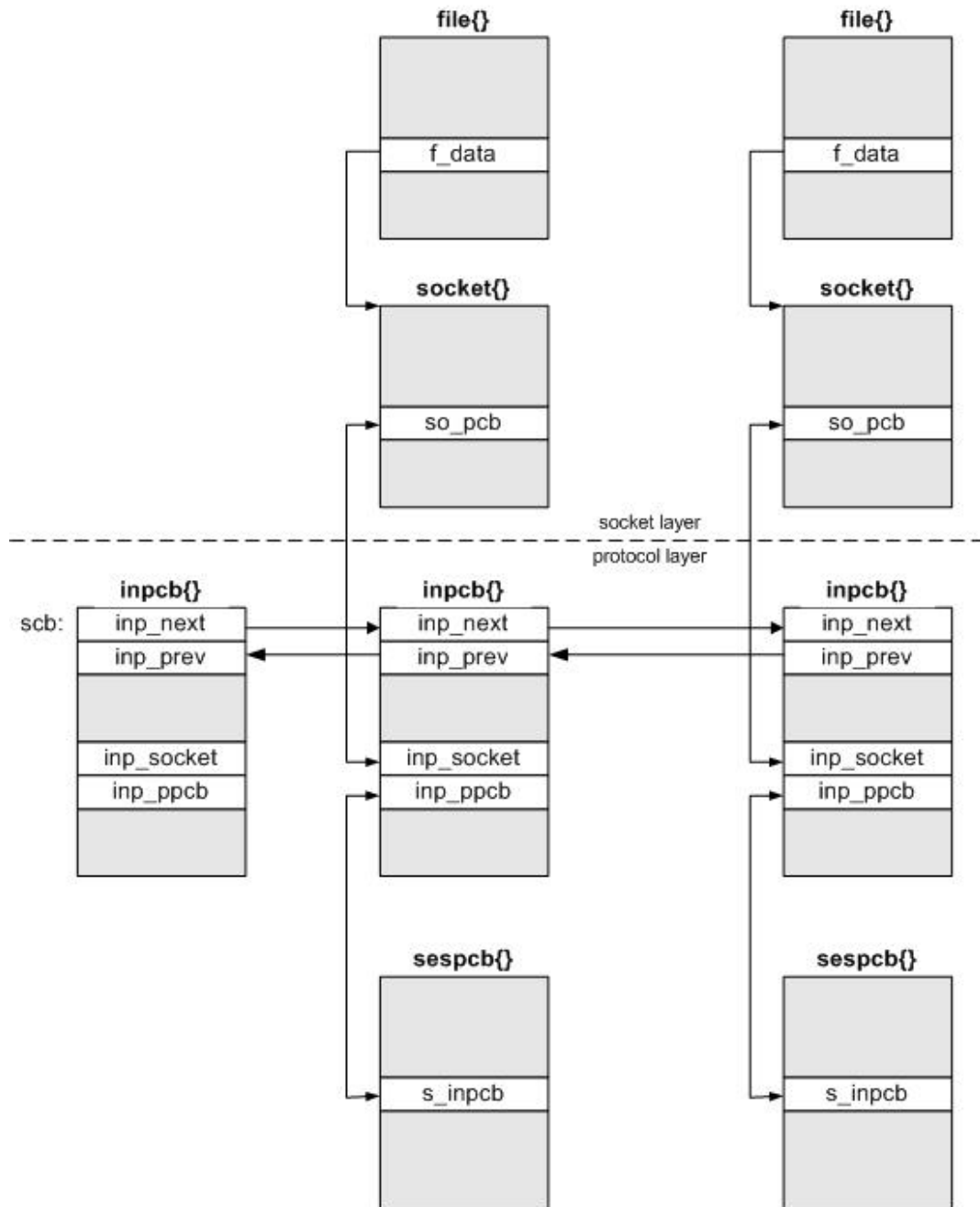


Figura 5.2 - SESP control buffers no kernel

No diagrama da figura 5.1 apenas observam-se as primitivas de serviço mais usuais para aplicações simples, mas o modelo deverá abranger todas as primitivas descritas por `pr_usrreqs`, sem exceções e sem alteração da interface, ou seja sem inclusão, exclusão ou alteração de qualquer um dos argumentos descritos por cada uma das rotinas.

Outro ponto importante para que as entidades de sessão sejam reconhecidas pela API de *socket*, são as questões relativas ao bloco de controle do protocolo SESP, que deverão seguir o modelo de PCBs do *Net/3*, como ilustrado na figura 5.2. Como será visto na seção 5.2, estes blocos de controle terão seus campos definidos em virtude das características do modelo implementado e das exigências observadas durante o processo de implementação.

5.1.3 Modelo de implementação como biblioteca de aplicação

Nesse modelo criou-se a proposta de implementar o protocolo SESP em um uma biblioteca auxiliar, que deve operar dentro do processo usuário, utilizando os serviços da API de *socket*, e oferecendo seus serviços para a aplicação por intermédio de uma camada *socket* simplificada, aqui definida como *Simplified SESP Socket Layer*.

Este modelo permite que as pequenas alterações feitas no protocolo sejam feitas, recompiladas e testadas rapidamente.

O modelo da biblioteca SESP se baseia na arquitetura de implementação do *Net/3*, análogo ao da implementação em *kernel*, isto para permitir as aplicações que façam uso das mesmas rotinas da interface de *socket* do sistema, como ilustrado na figura 5.3.

A implementação da camada de *socket* dentro da biblioteca abstrai ao máximo a arquitetura desta camada no sistema. Não há necessidade de oferecer todos os serviços da interface *socket* do sistema, já que as rotinas que não afetam os serviços de sessão podem ser utilizadas diretamente da interface convencional. Logo, o descritor de arquivos é associado ao processo da mesma forma. Uma estrutura de controle, abstraída da estrutura `socket` do *kernel*, armazena os dados relativos ao `socket` de sistema dentro da biblioteca, como o valor do descritor de arquivo do sistema, família de endereçamento, tipo de protocolo, entre outros. O motivo deste

procedimento é permitir que as solicitações feitas aos protocolos que não as necessitem, ou que não sejam suportados pelo protocolo de sessão, possam ser encaminhadas diretamente ao sistema.

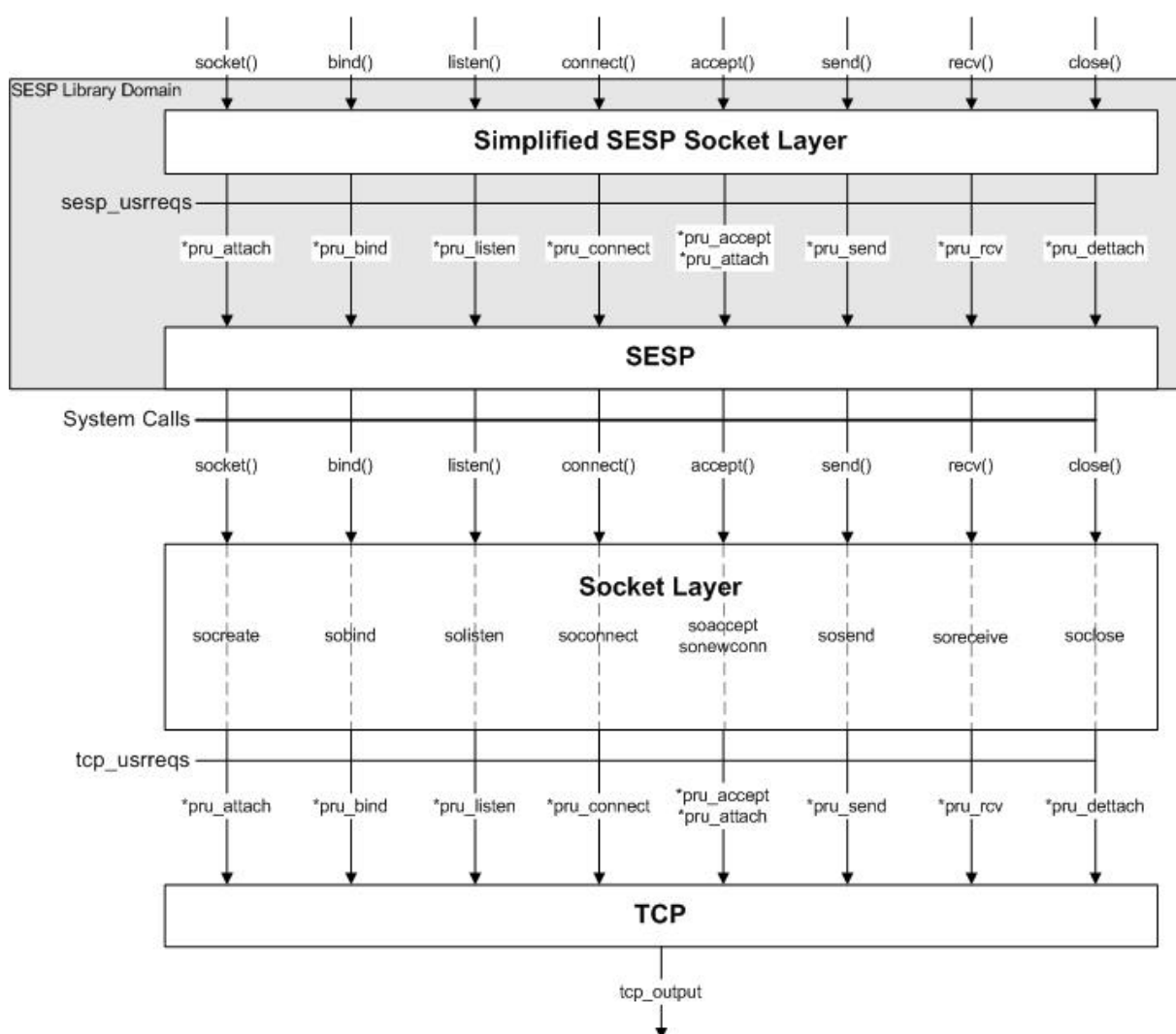


Figura 5.3 - Modelo de funcionamento da biblioteca SESP

O protocolo de sessão, também foi simplificado, de modo que implementa apenas as rotinas `pr_usrreqs`, pois a única entidade que irá acessá-lo será o processo usuário. A interface desta rotina é flexível, e pode ser definida de acordo com as necessidades definidas durante o processo de implementação. Observando-se as declarações no arquivo "`protosw.h`" fica evidente que certos parâmetros, tais como o

“processo usuário” no método `pru_connect`, podem ser abstraídos e outros, tais como “referência a buffer para recepção” na `pru_recv`, deverão ser adicionados.

Da mesma forma que o modelo de implementação no *kernel* as primitivas de serviços de sessão (sincronização, controle de *tokens* e controle de atividades) são oferecidas por um conjunto adicional de rotinas que, a princípio, são declaradas na interface de *socket* e requisitadas ao protocolo de sessão pela rotina `pru_send()`, como mencionado no capítulo seção 5.1.1.

5.2 Prototipação do SESP

Avaliando os modelos propostos optou-se por implementar o protótipo inicial do SESP seguindo o modelo de implementação como biblioteca de aplicação. A primeira versão de testes do SESP foi implementada em linguagem C. Nas compilações desta biblioteca foi utilizado o compilador Gnu C Compiler (GCC) sobre o sistema operacional Linux, distribuições RedHat 7.3 e 9.0. O modo de formatação escolhido para a biblioteca foi o Executable and Linking Format (ELF), que permite que a biblioteca seja ligada dinamicamente às aplicações teste em tempo de execução, evitando que estas aplicações sejam recompiladas a cada alteração da biblioteca.

Os códigos fontes descritos neste capítulo constam nos apêndices deste trabalho.

5.2.1 PDU do SESP

Todos os dados enviados por meio do protocolo de sessão são precedidos pelo cabeçalho da PDU do SESP, já descrito no capítulo 3. A estrutura que define este cabeçalho é a `sesphdr`, ilustrada na figura 5.4

```

struct sesphdr {
    u_int  sh_sid;           /* session id */
    u_char sh_serv;        /* session service */
    u_short sh_sclasses:13, /* in negotiation service classes */
           sh_category:3; /* INET session-transport category */
    u_short sh_protocol;   /* server application protocol type */
    u_int  sh_datalen;     /* data's size (in bytes) */
};

```

Figura 5.4 - Cabeçalho da PDU do SESP

Dentro do arquivo “*sesp.h*” também são definidas as constantes associadas aos campos do cabeçalho e serviços do protocolo, veja apêndice B.

5.2.2 Alocação de dados no protocolo SESP

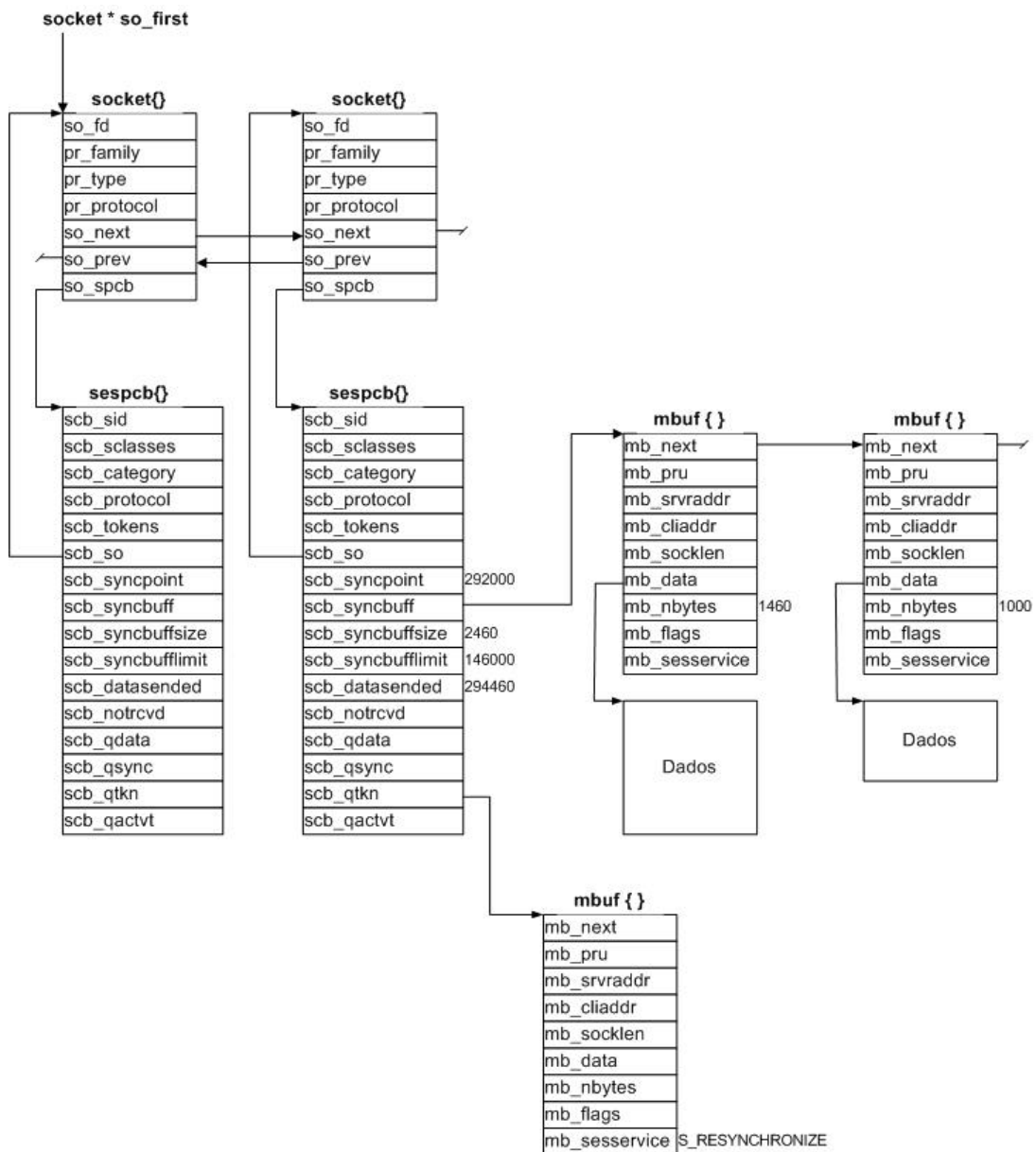


Figura 5.5 – Visão geral das estruturas do SESP

As estruturas de dados foram idealizadas como abstrações das estruturas utilizadas no *Net#3*, incluindo algumas características simplificadas, próprias para o uso dentro de processos usuários.

Foram criadas três estruturas principais: *memory buffer* ou `mbuf{}`, *socket descriptor* ou `socket{}` e *sesp control block* ou `sespcb{}`. Que se inter-relacionam como ilustrado na figura 5.5.

5.2.2.1 SESP Memory Buffer

Da mesma forma que no *Net#3* os *memory buffers*, ou simplesmente `mbufs`, da biblioteca SESP têm o objetivo de encapsular os dados que trafegam pelas camadas do modelo. A figura 5.6 mostra o modelo da estrutura, e seu código está descrito no arquivo "*mbuf.h*", que se encontra no apêndice C.

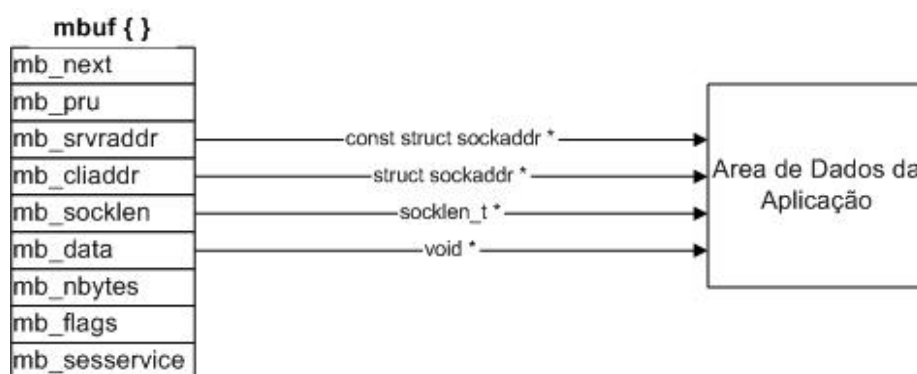


Figura 5.6 - SESP *memory buffer*

O primeiro campo do SESP `mbuf`, `mb_next`, é um ponteiro para `mbuf`, permitindo que estas estruturas sejam aninhadas em filas, ou cadeias de *memory buffers*.

O campo `mb_pru` indica que qual o tipo de serviço está sendo solicitado pela camada de *socket* ao protocolo SESP. Isto é necessário porque algumas rotinas podem processar mais de um tipo de requisição do usuário. Todas as PRUs são definidas por constantes declaradas no arquivo "*mbuf.h*", ilustrado no apêndice C.

Os campos `mb_srvraddr`, `mb_cliaddr`, `mb_socklen`, `mb_data`, `mb_nbytes`, e `mb_flags` armazenam, ou referenciam, os parâmetros de diferentes funções da API de *socket*. Para os parâmetros informados por referência, a área de dados continua aquela alocada pela aplicação. Se houver necessidade de armazená-los por mais tempo, como

por exemplo, no caso da montagem de um *buffer* para sincronização de dados, esta área de dados é copiada para uma área do mesmo tamanho, alocada pelo protocolo SESP.

E por ultimo, o campo `mb_sesservice`, é usado quando há necessidade de informar à aplicação que um determinado serviço de sessão, ou mensagem de sessão, foi recebido e processado pelo protocolo.

5.2.2.2 SESP Socket Descriptor

Para poder representar o *socket* do sistema na camada de *socket* da biblioteca observou-se a necessidade de criar uma estrutura que abstraísse as informações mais pertinentes de um *socket* do sistema. A estrutura foi modelada seguindo os modelos do Net/3, onde se encontrou a necessidade de enfileirar estas estruturas para que pudessem ser encontradas a partir do valor de seu descritor de arquivo. Portanto, se descartou a necessidade de implementar as estruturas `inpcb`, ilustradas na figura 5.2, que em uma implementação fora do *kernel* apenas teriam a finalidade de enfileirar os *control blocks* do protocolo de sessão.

A lista de *sockets* é duplamente encadeada, ou seja, uma estrutura aponta para a seguinte e para a anterior, sendo que a ordem que os *sockets* são encadeados é determinada por sua criação, onde o último a ser criado vai sempre para o fim da lista.

A figura 5.7 mostra o modelo da estrutura `socket`, e seu código está descrito no arquivo “`socket_var.h`” que se encontra no apêndice A.

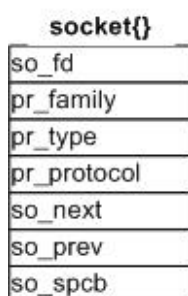


Figura 5.7 – Estrutura do SESP `socket{}`

O identificador do `socket` é seu `so_fd`, ou *socket file descriptor*, que armazena o valor do descritor de arquivo do *socket* real do sistema.

Os campos `pr_family`, `pr_type` e `pr_protocol` indicam as características do protocolo utilizado pelo `socket`, que são informadas no momento da sua criação. Para o caso de um `socket` que utilize o protocolo de sessão, sua família de endereçamento será a `AF_INET`, o tipo de protocolo será `SESSTREAM`, e seu campo `protocol` é informado como 0.

Como as estruturas são encadeadas duplamente, os campos `so_next` e `so_prev` referenciam o `socket` seguinte e o posterior, respectivamente.

A associação de uma sessão com um `socket` é feita referenciando-se o bloco de controle desta sessão no campo `so_spcb`.

5.2.2.3 SESP Control Block

A estrutura responsável por armazenar o estado de uma sessão é o *session protocol control block*, ou simplesmente `sespcb`. Ele é ligado diretamente ao seu `socket` sem necessidade de intermédio de uma estrutura geral, como no caso do `inpcb` do núcleo do BSD.

sespcb{}
scb_sid
scb_sclasses
scb_category
scb_protocol
scb_tokens
scb_so
scb_syncpoint
scb_syncbuff
scb_syncbuffsize
scb_syncbufflimit
scb_dataended
scb_notrcvd
scb_qdata
scb_qsync
scb_qtkn
scb_qactvt

Figura 5.8 - Estrutura do `sespcb{}`

No momento em que a aplicação chama a função `socket()` para que o `socket` seja criado, um `sescpcb` é associado a ele na camada de sessão, onde o valor 0 é atribuído como identificador desta sessão que, desta forma, fica definida como inativa.

A figura 5.8 mostra o modelo do bloco de controle, e seu código está descrito no arquivo “`sesp_var.h`” que se encontra no apêndice B.

As informações relativas à configuração da sessão corrente, descritas na PDU do protocolo de sessão, são armazenadas nos campos:

- `scb_sid` – identificador de sessão;
- `scb_classes` – classes de serviço negociadas;
- `scb_category` – categoria de transporte usada para a sessão;
- `scb_protocol` – classificação do tipo de aplicação.

Os *tokens* que as entidades de sessão podem possuir são armazenados em `scb_tokens`.

Da mesma forma que no modelo de implementação em *kernel*, a implementação em biblioteca estabelece uma referência a partir do bloco de controle para o seu `socket` associado, feita pelo campo `scb_so`.

Uma aplicação pode utilizar o serviço opcional de sincronização. Para estabelecer o controle deste serviço o bloco de controle possui os seguintes campos:

- `scb_syncpoint` – valor do último ponto de sincronização estabelecido;
- `scb_syncbuff` – referência para o *buffer* de dados que serão reenviados em uma possível re-sincronização;
- `scb_syncbuffsize` – quantidade de dados armazenados no *buffer* de sincronização, em número de *bytes*;
- `scb_syncbufflimit` – tamanho máximo que poderá ser admitido ao *buffer* de sincronização, usado para evitar problemas de *memory leak*;
- `scb_dataended` – indica a quantidade de dados já enviados desde o início da sessão, em número de *bytes*.

Durante o processo de implementação foram encontrados alguns problemas relativos à recepção de dados, que são comentados detalhadamente na seção 5.2.6

deste capítulo. Para resolução destes problemas foram estabelecidos alguns campos, que possivelmente só serão utilizados se implementados fora do *kernel*, são eles:

- `scb_notrcvd` – quantidade de dados ainda não recebidos, em bytes;
- `scb_qdata` – fila de dados recebidos que ainda não foram lidos pela aplicação;
- `scb_qsync` – fila de pontos de sincronização recebidos, porem não verificados pela aplicação;
- `scb_qtkn` – fila de mensagens de controle de *tokens* recebidos e ainda não verificados pela aplicação;
- `scb_qtkn` – fila de mensagens de controle de atividades recebidos, mas ainda não verificados pela aplicação.

5.2.3 SimplifiedSESP Socket API

A API de *socket* do protocolo de sessão implementou as funções mais usuais da API do sistema e as rotinas adicionais descritas na seção 5.1.1, com exceção dos serviços relativos ao controle de atividade.

O *header file* da biblioteca que define a interface da API *socket* de sessão é o “*socket.h*”, contido no apêndice A. As funções desta API receberam o prefixo “*sp_*” para evitar que sejam confundidas com as chamadas de sistema com o mesmo nome, mas apesar deste detalhe a passagem de parâmetros não foi modificada e é feita seguindo as definições de Posix.

5.2.3.1 Common API

A API de sessão da biblioteca oferece as seguintes rotinas compatíveis com a API de *socket*:

- `int sp_socket (int, int, int)`
 - Descrição: corresponde à rotina de sistema `socket()`, que é usada para criar um novo socket.
 - Parâmetros:

- Família de endereçamento: deve ser sempre definida como `AF_INET`, a camada *socket* da biblioteca não suporta as outras famílias disponíveis no na API do sistema;
 - Tipo de *socket*: podem ser os 3 tipos usuais da API do sistema, ou `SOCK_SEQPACKET` para uso do protocolo de sessão orientado a conexão;
 - Protocolo: normalmente definido como 0.
- `int sp_listen (int, int)`
 - Descrição: corresponde à rotina de sistema `listen()`, que é usada para definir o *socket* inativo como *socket* passivo, ou ouvinte.
 - Parâmetros:
 - Valor do descritor de *socket*;
 - Número máximo de conexões que podem aguardar na fila de espera.
 - `int sp_accept (int, struct sockaddr *, socklen_t *)`
 - Descrição: corresponde à rotina de sistema `accept()`, que é usada para aceitar conexões, respondendo o procedimento de negociação de sessão com o par solicitante.
 - Parâmetros:
 - Valor do descritor de *socket*;
 - Endereço do par que solicitou a conexão;
 - Tamanho da estrutura onde o endereço foi armazenado;
 - `int sp_bind (int, const struct sockaddr *, socklen_t)`
 - Descrição: corresponde à rotina de sistema `bind()`, que é usada para atribuir o endereço IP e a porta ao *socket*.
 - Parâmetros:
 - Valor do descritor de *socket*;
 - Endereço da máquina;
 - Tamanho da estrutura onde o endereço foi informado.
 - `int sp_connect (int, const struct sockaddr *, socklen_t)`

- Descrição: corresponde à rotina de sistema `connect()`, que é usada para conectar o *socket* a uma entidade par, servidor, que esteja no estado ouvinte. Inicia o procedimento de negociação de serviços de sessão com a entidade de sessão do servidor.
- Parâmetros:
 - Valor do descritor de *socket*;
 - Endereço da máquina e porta de conexão da entidade par, servidor, que se deseja conectar;
 - Tamanho da estrutura onde o endereço foi informado.
- `int sp_getpeername (int, struct sockaddr *, socklen_t *)`
 - Descrição: corresponde à rotina de sistema `getpeername()`, que é usada para retornar o endereço do par remoto.
 - Parâmetros:
 - Valor do descritor de *socket*;
 - Endereço do par conectado;
 - Tamanho da estrutura onde o endereço foi informado.
- `int sp_getsockname (int, struct sockaddr *, socklen_t *)`
 - Descrição: corresponde à rotina de sistema `getsockname()`, que é usada para retornar o endereço associado ao *socket* local.
 - Parâmetros:
 - Valor do descritor de *socket*;
 - Endereço associado ao *socket*;
 - Tamanho da estrutura onde o endereço foi informado.
- `int sp_getsockopt (int, int, int, void *, socklen_t *)`
 - Descrição: corresponde à rotina de sistema `getsockopt()`, que é usada para retornar opções do *socket* ou dos protocolos.
 - Parâmetros:
 - Valor do descritor de *socket*;
 - Nível, onde o nível do protocolo SESP é definido como `IPPROTO_SESP`;

- Nome da Opção, onde o protótipo define até o momento as opções `SESP_SID`, que retorna o valor do identificador da sessão, e `SESP_SERVCLASS`, que retorna as classes de serviço ativas na sessão;
 - Ponteiro para retorno do valor da opção, onde para a opção `SESP_SID` deve apontar para uma variável do tipo `u_int` e para a opção `SESP_SERVCLASS` para uma variável do tipo `u_short`;
 - Tamanho da estrutura de retorno do valor;
- `int sp_setsockopt (int, int, int, const void *, socklen_t)`
 - Descrição: corresponde à rotina de sistema `setsockopt()`, que é usada para configurar opções no *socket* ou nos protocolos.
 - Parâmetros:
 - Valor do descritor de *socket*;
 - Nível, onde o nível do protocolo `SESP` é definido como `IPPROTO_SESP`;
 - Nome da Opção, onde o protótipo define até o momento a opção `SESP_SERVCLASS`, que pode ser usada antes da sessão ficar ativa para configurar as classes de serviço que se deseja negociar no estabelecimento de conexão da sessão;
 - Ponteiro para o valor da opção, onde a opção `SESP_SERVCLASS` deve ser o endereço de uma variável do tipo `u_short`;
 - Tamanho da estrutura onde consta o valor da opção;
- `int sp_close(int)`
 - Descrição: corresponde à rotina de sistema `close()`, que aqui é usada para efetuar o encerramento de uma sessão, que pode ser negociado ou não.
 - Parâmetros:
 - Valor do descritor de *socket*;
- `int sp_shutdown (int, int)`

- Descrição: corresponde à rotina de sistema `shutdown()`, encerrando a conexão de sessão sem fechar o *socket*.
- Parâmetros:
 - Valor do descritor de *socket*;
 - *Howto*, informa se encerra leitura, ou escrita, ou ambos.
- `ssize_t sp_recv (int, void *, size_t, int)`
 - Descrição: corresponde à rotina de sistema `recv()`, recebe dados que estejam na fila do *socket* do sistema.
 - Parâmetros:
 - Valor do descritor de *socket*;
 - *Buffer* onde os dados serão armazenados;
 - Tamanho do *buffer* de recepção, ou quantidade de dados a serem recebidos;
 - *Flags* de recepção.
- `ssize_t sp_send (int, const void *, size_t, int)`
 - Descrição: corresponde à rotina de sistema `send()`, que envia dados para o par comunicante.
 - Parâmetros:
 - Valor do descritor de *socket*;
 - *Buffer* contendo os dados a serem enviados;
 - Tamanho do *buffer*, ou quantidade de dados a serem enviados;
 - *Flags* de envio.

Se a sessão estiver operando em serviço *half-duplex* a rotina `sp_send()` fica restrita à entidade que possuir o *token* de envio de dados.

Da mesma forma a rotina `sp_close()` fica restrita à entidade que possuir o *token* de encerramento se a sessão estiver operando com o serviço de encerramento negociado.

5.2.3.2 SESP *Additional*API

Para complementar as rotinas do núcleo do protocolo SESP esta biblioteca suporta as seguintes rotinas de sessão:

- `ssize_t sp_send_typed (int, const void *, size_t, int)`
 - Descrição: semelhante à rotina `sp_send()`, só que os dados são enviados, em uma sessão *half-duplex*, mesmo sem a posse do *token* de dados pela entidade.
 - Parâmetros:
 - Valor do descritor de *socket*;
 - *Buffer* contendo os dados a serem enviados;
 - Tamanho do *buffer*, ou quantidade de dados a serem enviados;
 - *Flags* de envio.
- `int sp_token_request (int, u_short)`
 - Descrição: envia uma mensagem de requisição de *token* a entidade par.
 - Parâmetros:
 - Valor do descritor de *socket*;
 - Identificador(es) do(s) *token(s)* requisitado(s).
- `int sp_token_give (int, u_short)`
 - Descrição: envia *token(s)* a entidade par.
 - Parâmetros:
 - Valor do descritor de *socket*;
 - Identificador(es) do(s) *token(s)* enviado(s).
- `int sp_control_give (int)`
 - Descrição: envia todos os *tokens* a entidade par.
 - Parâmetros:
 - Valor do descritor de *socket*.
- `int sp_sync (int)`
 - Descrição: estabelece um novo ponto de sincronização com a entidade par; só pode ser feito com a posse do *token* de sincronização.
 - Parâmetros:

- Valor do descritor de *socket*.
- `int sp_resync (int, int)`
 - Descrição: solicita reenvio dos dados a partir do ponto de sincronismo informado.
 - Parâmetros:
 - Valor do descritor de *socket*;
 - Identificador do ponto de sincronismo.
- `int sp_recv_token (int, int *, int *)`
 - Descrição: verifica se existem mensagens de controle de *token* não lidas na entidade de sessão; a mensagem só será processada a partir do momento que ela for lida pela aplicação.
 - Parâmetros:
 - Valor do descritor de *socket*;
 - Serviço associado a mensagem;
 - Identificador do *token*.
- `int sp_recv_sync (int, int *)`
 - Descrição: verifica se existem novos pontos de sincronismo recebidos.
 - Parâmetros:
 - Valor do descritor de *socket*;
 - Identificador do ponto de sincronismo.

5.2.4 Estabelecimento de Conexão e Negociação de Serviços

Como visto no capítulo 3, na fase de estabelecimento de conexão o SESP aguarda que a conexão de transporte seja estabelecida para iniciar a negociação dos serviços que serão utilizados durante a sessão. O lado cliente requisita o uso de determinadas categorias de serviços e o servidor retorna quais destas categorias ele disponibiliza.

Uma vez que a sessão se torne ativa ela não poderá mais alterar seus parâmetros de serviços disponíveis, portanto as aplicações devem fazer uso da função `sp_setsockopt()` para definir que serviços desejam que sua entidade de sessão utilize antes que a conexão seja estabelecida. Portanto o servidor deverá fazer isto

antes de chamar a rotina `sp_accept()` e o cliente antes de chamar a rotina `sp_connect()`.

Os parâmetros informados em `sp_setsockopt()` para configurar os serviços deverão ser, o valor do identificador do *socket*, o nível `IPPROTO_SESP`, o nome da opção `SESP_SERVCLASS` e os valores dos serviços desejados, que são:

- `SC_THALF`: transferência de dados no modo *half-duplex*;
- `SC_DTSYNC`: sincronização da transferência de dados;
- `SC_ACTIVITY`: gerência de atividade (não implementado ainda);
- `SC_CLOSE`: encerramento negociado.

5.2.5 Transmissão de Dados

Todas as requisições de envio de dados e envio de mensagens de controle de *token* ou de sincronização são submetidos à rotina `sesp_usr_send()` do protocolo de sessão. Nesta rotina o protocolo verifica o campo `mb_pru` da *memory buffer*, que foi montado e enviado pela camada de socket, para identificar qual serviço de envio de dados a aplicação está requisitando podendo, assim, efetuar a análise de permissão para uso do serviço e, em seguida, montar o cabeçalho da PDU de sessão apropriado para o serviço.

Quando requisita o envio de dados ou outros serviços por meio do protocolo SESP, este protocolo deve sempre checar se a entidade de sessão possui os *tokens* que permitem a ela utilizar tais serviços.

Se o serviço de sincronização estiver ativo, após a confirmação da transmissão dos dados pelo protocolo de transporte, a entidade deve criar uma cópia do `mbuf` utilizado no envio e armazená-lo no *buffer* de sincronização do bloco de controle.

5.2.5.1 Armazenamento de Dados para Sincronização

O armazenamento dos dados enviados, para possibilitar o re-sincronismo da sessão, é uma das características mais importantes do uso do protocolo SESP, principalmente para aplicações multimídia, mas este serviço levanta questões sérias em

torno de qual a quantidade máxima de dados e pontos de sincronismo que a sessão pode armazenar, e também onde estes dados serão armazenados.

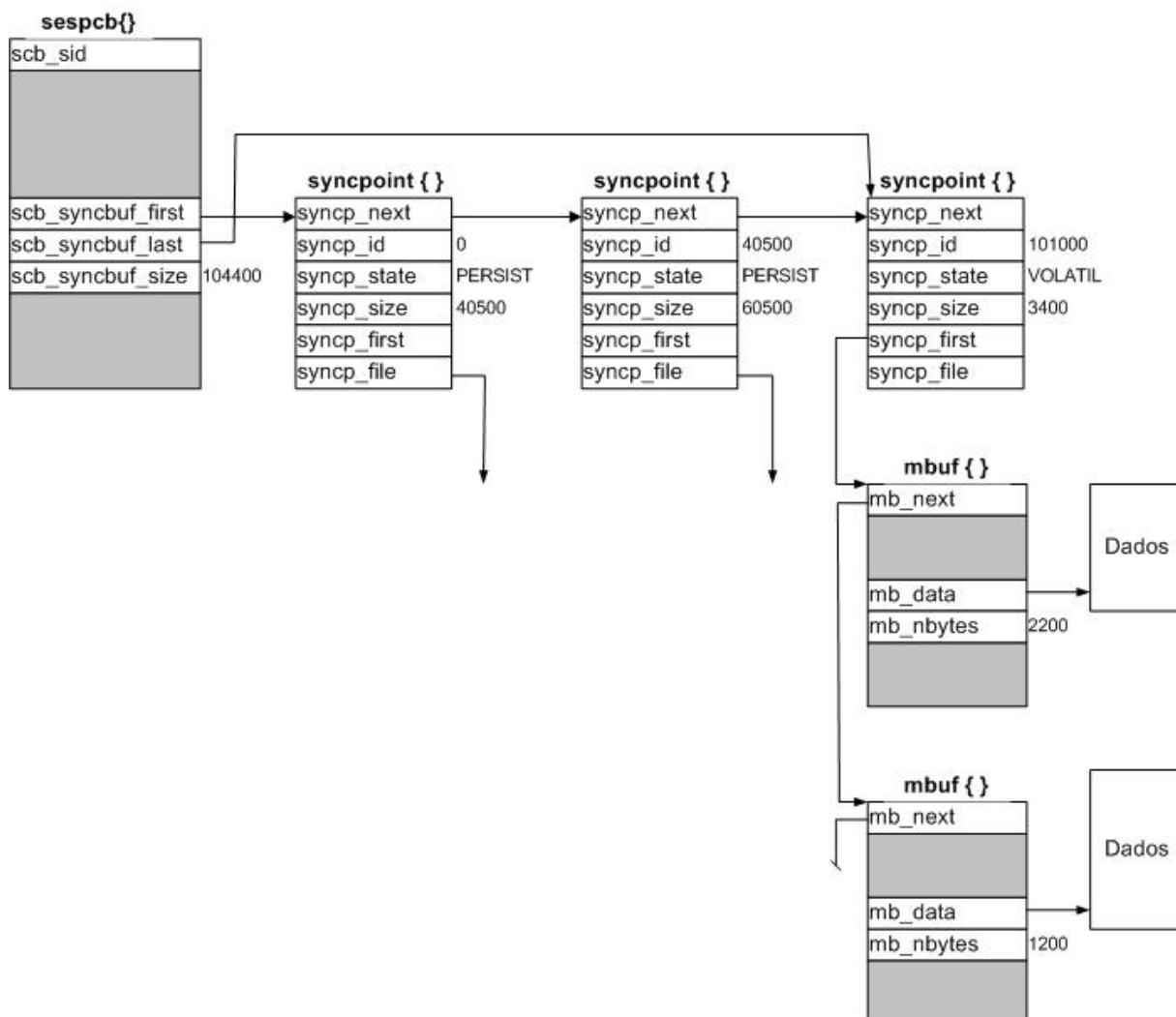


Figura 5.9 - SESP suportando mais de um ponto de sincronização

Como o modelo do protocolo SESP não havia considerado as questões relativas ao uso excessivo de recursos do sistema até o momento, e deixa algumas lacunas em branco quando ao controle dos dados armazenados, a implementação disponibilizou este serviço da maneira mais simples e menos custosa para o sistema, onde o *buffer* somente armazena os dados enviados a partir do último ponto de sincronismo estabelecido, como ilustrado na figura 5.5. Se nenhum ponto de sincronismo for enviado antes que o *buffer* atinja sua capacidade limite, as mensagens mais antigas

contidas neste *buffer* são eliminadas. Este procedimento é necessário para evitar que eventualmente ocorram *memory leaks*⁷ no sistema por causa da má implementação das aplicações que utilizam o serviço.

Ao longo do processo de modelagem do protótipo se pensou em armazenar os dados de pontos de sincronismo antigos em memória secundária, no sistema de arquivos, como ilustrado na figura 5.9. Neste tipo de solução o valor do campo `syncp_file` pode ser uma estrutura `inode` ou o valor de um descritor de arquivo, se o protocolo for implementado em *kernel*, ou simplesmente o *path* do arquivo dentro do sistema de arquivos. No entanto, alguns problemas ainda pendentes na modelagem do protocolo justificam a não implementação desta proposta neste primeiro protótipo:

- Como e onde estes dados serão armazenados?
- O administrador do sistema poderá configurar o modo e local que o SESP armazena estes dados? Considerou-se que nem todos os sistemas dispõem de espaço em disco rígido suficiente para armazenar quantidades grandes de pontos de sincronismo.
- Se o SESP não puder armazenar uma grande quantidade de dados qual o critério de seleção para a alocação dos dados?
- Se uma aplicação deseja armazenar dados para sincronismo e o sistema de arquivos estourar o que acontece com a aplicação, e o que acontece com o sistema?
- A relação custo benefício compensaria a implementação de um modelo mais complexo para armazenar os pontos de sincronismo?
- Qual a quantidade de aplicações e com que freqüência elas necessitarão fazer re-sincronização de dados?

Possivelmente o uso do SESP em seu modelo mais simples e futuros testes com levantamento de dados estatísticos poderão indicar a necessidade ou não de uma arquitetura mais complexa para o armazenamento destes dados.

⁷ *Memory leak*, ou estouro da pilha de memória, pode ocorrer se recursos da pilha de memória principal do sistema forem alocados intermitentemente por um processo até que toda memória livre seja exaurida, o que faz com que o sistema trave, ou em alguns casos mate o processo nocivo, liberando a memória alocada.

5.2.6 Recepção de Dados

Com relação à recepção de dados, foram encontrados alguns problemas relativos à grande quantidade de serviços oferecidos pelo protocolo de sessão e ao fato de o protocolo ter sido implementado fora do *kernel*. Para isto alguns tratamentos de dados necessitaram ser implementados na rotina de recepção de dados `sesp_usr_rcvd()`.

Três tipos de serviços podem ser recebidos pela API de *socket* da biblioteca SESP: dados, *token*, pontos de sincronização. Como nem sempre a primeira mensagem na fila do *socket* do sistema é do tipo que a aplicação deseja no momento, o protocolo SESP armazena esta mensagem em uma fila, para entrega posterior. As três filas são: `scb_qdata`, `scb_qsync` e `scb_qtkn`.

A API de *socket* do sistema só garante que os pacotes TCP recebidos corretamente sejam entregues, e sendo que o fluxo de recepção dos dados não é contínuo, ou seja, por exemplo, quando uma aplicação solicita receber uma mensagem de 20 *kbytes* somente metade destes pode ter chegado corretamente e estar armazenada no *buffer* do *socket*. Tendo em mente esta deficiência, cada entidade do protocolo SESP mantém um registro de dados não recebidos. Assim, quando uma nova PDU de dados chega, o protocolo SESP lê o campo `sh_dataalen` do cabeçalho da PDU e tenta ler todos os dados contidos nesta PDU, se a PDU não chegou por completo, a quantidade de bytes que falta ser recebida pela sessão será informada no campo `scb_notrcvd` do bloco de controle da sessão.

Quando a aplicação usa uma das rotinas de recepção de dados definidas pela API `sp_rcv()`, `sp_rcv_token()`, ou `sp_rcv_sync()`, a rotina `sesp_usr_rcvd()` é chamada pela camada de *socket*. Primeiramente esta rotina irá verificar se existem dados ainda não lidos da PDU de sessão anterior, indicado pelo campo `scb_notrcvd`. Em seguida irá conferir se a aplicação possui alguma mensagem do tipo requerido em uma das filas de entrega posterior. Não havendo mensagem, ela irá chamar a rotina `rcv()`, da API de *socket* do sistema, para pegar o cabeçalho da próxima mensagem. Depois de reconhecido o cabeçalho, a rotina irá encaminhar os dados para a aplicação ou para uma das filas de entrega posterior.

Se a aplicação requerer a recepção de dados, normais ou tipados, e o tamanho do seu *buffer* de recepção for menor que a quantidade de dados da mensagem, o

protocolo SESP entregará somente a quantidade de dados suportada pela aplicação, armazenando o restante na fila de espera `scb_qdata`.

É importante frisar que estes problemas serão contornados com maior facilidade na implementação do protocolo dentro do *kernel* do sistema, e muitos dos quais nem sequer existirão se a interação entre TCP e SESP for feita da maneira adequada.

5.2.7 Tratamento de Erros

Para o tratamento de erros a biblioteca de sessão usa a variável global do sistema `errno`, onde Stevens (1998, p. 12) descreve que, “quando um erro ocorre em uma função Unix (como uma das funções *socket*), a variável global `errno` é definida como um valor positivo que indica o tipo de erro e a função normalmente retorna o valor `-1`”.

Para suportar este tipo de tratamento no arquivo “`errno.h`”, da biblioteca SESP, foram definidos as constantes dos erros relativos ao protocolo de sessão, sendo eles:

- `ESERVNA = 125`: serviço não permitido, retornado quando a aplicação deseja usar um serviço sem possuir a permissão necessária.
- `EDTOOBL = 126`: dados maiores que o limite do *buffer*, é retornado quando uma aplicação utiliza uma estrutura de dados de tamanho muito pequeno para armazenar algum tipo de dado que não pode ser segmentado, como *tokens*, e pontos de sincronismo.
- `ENOSESPDT = 150`: cabeçalho SESP inválido, é um erro usado para depuração do protocolo e é informado quando o cabeçalho recebido pelo protocolo SESP possui valores inválidos.
- `ESESPFAULT = 151`: falha no SESP, também é um erro usado somente para depuração, e é usado de forma genérica para identificar possíveis falhas.

Os valores foram atribuídos a estes erros de forma a não conflitarem com os já existentes no Linux ou no FreeBSD, onde os máximos definidos são 124 e 86 respectivamente.

O problema ao se utilizar esta abordagem de tratamento de erros fora do sistema é que as funções apropriadas para lidarem com mensagens de erro, `perror()`,

`strerror()`, `strerror_r()`, `sys_errlist()`, `sys_nerr()`, não reconhecerão os erros descritos na biblioteca.

6 Conclusão

Com a implementação do primeiro protótipo do SESP, este trabalho contribuiu para que o projeto de desenvolvimento do *Session Protocol*, a partir das formulações teóricas definidas até o momento, possa ter uma visão mais empírica, voltando-se para sua validação experimental.

Através da pesquisa da arquitetura de implementação do TCP/IP, com o levantamento de sua estrutura de dados, com o mapeamento do fluxo de dados das rotinas de E/S e com o entendimento de seu funcionamento, foi possível desenvolver um modelo de como o SESP pode ser integrado à estrutura atual do Net/3, permitindo que o primeiro protótipo do SESP ficasse o mais próximo possível de sua real implementação no *kernel* do BSD. Se não fosse esta contribuição o protocolo SESP seria projetado como mais um protocolo da camada de aplicação.

O trabalho também contribuiu para que características ainda não analisadas fossem incorporadas, ou reavaliadas, no modelo do protocolo SESP, como:

- Ajuste dos tamanhos dos campos do cabeçalho da PDU de sessão;
- Utilizando-se o protocolo TCP para transporte, o protocolo de sessão não necessita fazer a confirmação do envio de cada uma de suas PDUs;
- A recepção de dados na arquitetura TCP/IP é coordenada por requisições da aplicação, portanto é necessário que o protocolo de sessão possua rotinas que permitam a recepção de mensagens de controle de diálogo, sincronização e atividade.
- Definição dos valores que identificam as primitivas de serviço, *tokens*, classes de serviço e categorias de transporte.

Para efetuar este trabalho de conclusão de curso foram aplicados e aprofundados os conhecimentos de diversas áreas; dentre elas destacam-se: redes de computadores, sistemas operacionais, engenharia de software, métodos formais e protocolos de comunicação.

6.1 Dificuldades encontradas

Ao longo do desenvolvimento do trabalho percebeu-se que a implementação do *Net#3* é amarrada a uma arquitetura de duas camadas (protocolos - *socket*), e sendo o TCP o protocolo superior da camada de protocolos, sua interação é feita diretamente com os *sockets*. Esta característica diminui bastante a modularidade da arquitetura do *Net#3*, dificultando a inserção do SESP dentro dele.

Levando-se em conta que: compilar e realizar o *debug* de uma pequena aplicação com poucas linhas de código é bem mais simples do que efetuar as mesmas operações em meio a toda a complexidade do núcleo do sistema, e também que a cada alteração realizada todo o *kernel* deve ser recompilado e o sistema deve ser reiniciado para que se possam realizar os testes. Após diversas tentativas frustradas de tentar alterar o código fonte do sistema FreeBSD e conseguir compilá-lo, julgou-se necessária a elaboração de um modelo mais prático e flexível que a implementação direta no *kernel*, para que a prototipação do protocolo, sem sua validação formal, fosse implementada mais rapidamente. Por isto, a primeira versão de testes do SESP foi desenvolvida seguindo uma abordagem de biblioteca de aplicação, mas abstraída como se estivesse implementada no *kernel*.

6.2 Trabalhos futuros

Para que a implementação integrada ao *kernel* seja feita adequadamente, um dos pontos mais importantes, e possivelmente o que demandará maior tempo de pesquisa será re-orientar o fluxo e o tratamento de dados entrantes no sistema, ou seja, permitir que os dados que se destinam ao sistema passem pelo protocolo de sessão antes de chegarem à aplicação de destino. Para isto, deverá ser feita uma análise detalhada do código do *Net#3*, procurando todas as rotinas da camada de *socket* que são utilizadas pelos protocolos TCP onde, além de localizar as chamadas, deverá se alterar o código dos protocolos de transporte para que eles enviem os dados vindos de sua interface de entrada (descrita por `pr_input`) ao protocolo de sessão, ao invés de encaminharem aos buffers da camada de *socket*. Para isto o recebimento de dados do protocolo SESP precisa ser claramente definido e validado formalmente em seu modelo, pois as

tentativas de depuração de erros do modelo poderão ser altamente custosas e inviáveis uma vez que o mesmo esteja operando dentro do *kernel*.

Para que haja persistência de uma entidade de sessão, ou seja, ela deverá manter-se ativa mesmo quando as entidades comunicantes estiverem inativas por um longo período, ela deve armazenar seu estado periodicamente em algum local no sistema de arquivos, em memória secundária. Esta possibilidade gerou um impasse, pois se houver acessos muito freqüentes ao disco rígido o desempenho deve cair sensivelmente no transporte de dados por meio do uso do SESP, e se o tempo entre gravação do estado da sessão em disco for muito grande, no pior dos casos, pode ocorrer inconsistência de dados entre as entidades de sessão comunicantes. Técnicas de consistência e *log*, utilizadas em sistemas de gerência de bancos de dados, poderão ser estudadas e aplicadas no caso da persistência das entidades de sessão, buscando encontrar uma solução viável para a sua implementação.

Este primeiro protótipo do protocolo SESP foi implementado sem oferecer alguns serviços que ainda se encontram em processo de modelagem. Portanto, também são deixados como referência a trabalhos futuros: a implementação dos procedimentos de controle de atividades e a implementação do modelo de *bufferização* de múltiplos pontos de sincronismo.

Referências Bibliográficas

- BARRETO, Fernando. Protocolo de comunicação para multicomputador. 2002. Dissertação (Mestrado em Ciências da Computação) – Departamento de Informática e Estatística, Universidade Federal de Santa Catarina, Florianópolis.
- BARLOW, Daniel. The Linux ELF HOWTO. [S.l.]: [s.n], 1996. Disponível em: <<http://www.sgmltools.org/HOWTO/ELF-HOWTO/t1.html>>. Acesso em: 05 jan. 2004.
- BUDAG, Karlos H. Implementação do núcleo do sistema operacional distribuído A Crux. 2002. Dissertação (Mestrado em Ciências da Computação) – Departamento de Informática e Estatística, Universidade Federal de Santa Catarina, Florianópolis.
- CARVALHO, Daniel Balparda. Curso de Linguagem C. Belo Horizonte: Núcleo de Ensino à Distância da Escola de Engenharia da Universidade Federal de Minas Gerais, 2000. Disponível em: <<http://ead1.eee.ufmg.br/cursos/C/>>. Acesso em: 05 jan. 2004.
- DANTAS, Mario. Tecnologias de Redes de Comunicação e Computadores. Rio de Janeiro: Axcel Books, 2002. ISBN 85-7323-269-6.
- FREEBSD ARCHITECTURE Handbook, The. [S.l.]: The FreeBSD Documentation Project, 2003. Disponível em: <http://www.freebsd.org/doc/en_US.ISO8859-1/books/arch-handbook/index.html>. Acesso em: 23 nov. 2003.
- FREEBSD DEVELOPER's Handbook, The. [S.l.]: The FreeBSD Documentation Project, 2003. Disponível em: <http://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook/index.html>. Acesso em: 23 nov. 2003.
- FREEBSD MAN Pages: [S.l.]: The FreeBSD Project, 2003. Disponível em: <<http://www.freebsd.org/cgi/man.cgi>>. Acesso em: 20 jan. 2004.
- FREE SOFTWARE FOUNDATION. Introdução ao Projeto GNU. Tradução de Fernando Lozano. 1999. Disponível em: <<http://www.gnu.org/gnu/gnu-history.pt.html>>. Acesso em: 29 jun. 2003.
- GNU's Not Unix: the GNU Project and the Free Software Foundation. Boston: Free Software Foundation, Inc. Disponível em: <<http://www.gnu.org>>. Acesso em: 29 jun. 2003.
- ITU-T. Telecommunication Standardization Sector. Recommendation X.200: Open System Interconnection – Basic Reference Model: The Basic Model. [S.l.]: ITU-T, 1994.

ITU-T. Telecommunication Standardization Sector. Recommendation X.215: Information Technology - Open System Interconnection – Session Service Definition. [S.I.]: ITU-T, 1995.

ITU-T. Telecommunication Standardization Sector. Recommendation X.225: Information Technology - Open System Interconnection – Connection-Oriented Session Protocol: Protocol Specification. [S.I.]: ITU-T, 1995.

LINUX Online. Direitos autorais de Linux Online Inc. Disponível em: <<http://www.linux.org>>. Acesso em: 29 jun. 2003.

LU, Hongjiu. ELF: From The Programmer's Perspective. White Plains: NYNEX Science & Technology, Inc., 1995. Disponível em: <http://linux4u.jinr.ru/usoft/WWW/www_debian.org/Documentation/elf/elf.html>. Acesso em: 19 jan. 2004.

MARSHALL, Dave. Programming in C: UNIX System Calls and Subroutines using C. Cardiff: School of Computer Science of the Cardiff University, 1999. Disponível em: <<http://www.cs.cf.ac.uk/Dave/C/CE.html>>. Acesso em: 10 jan 2004.

PARK, Alfred. C Programming Tutorial. The Sherrill Group of Georgia Institute of Technology, 2001. Disponível em: <<http://vergil.chemistry.gatech.edu/resources/programming/c-tutorial/toc.html>>. Acesso em: 10 jan. 2004.

PASC. Portable Applications Standards Committee (including POSIX) of the Institute for Electrical and Electronics Engineers Computer Society. Disponível em: <<http://www.pasc.org/standing/sd11.html>>. Acesso em: 30 out. 2003.

PENDLETON, Stephen. BSD Sockets Tutorial. [S.I.]: C-Scene, 2001. Disponível em: <<http://cscene.unitycode.org/topics/unix/cs2-13.xml.html>>. Acesso em: 11 dez. 2004.

POSTEL, J. B. Internet Protocol: RFC 792. Marina Del Rey: Information Sciences Institute of University of Southern California, 1981. Disponível em: <<http://www.ietf.org/rfc/rfc0793.txt?number=792>>. Acesso em: 28 abr. 2003.

POSTEL, J. B. Transmission Control Protocol: RFC 793. Marina Del Rey: Information Sciences Institute of University of Southern California, 1981. Disponível em: <<http://www.ietf.org/rfc/rfc0793.txt?number=793>>. Acesso em: 28 abr. 2003.

RUSLING, David A. The Linux kernel. Berkshire, UK: [s.n.], 1999. Disponível em: <<http://metalab.unc.edu/LDP/LDP/tlk/tlk.html>>. Acesso em: 29 jun. 2003.

SALZMAN, Peter Jay; POMERANTZ, Ori. The Linux kernel module programming guide. [S.I.: s.n.], 2003. Disponível em: <<http://www.tldp.org/LDP/lkmpg/index.html>>. Acesso em: 29 jun. 2003.

SOARES, Luis Fernando G.; LEMOS, Guido; COLCHER, Sérgio. Redes de Computadores, Das LANs MANs e WANs às Redes ATM. 2. ed. Rio de Janeiro: Campus, 1995. ISBN 85-7001-954-8.

SILBERSCHATZ, Abraham; GALVIN, Peter Baer. Sistemas operacionais: conceitos. 5. ed. São Paulo: Prentice Hall, 2000. ISBN 85-87918-02-8

STEVENS, W. Richard. Unix Network Programming: Networking APIs - Sockets and XTI, Volume 1. 2. ed. New Jersey: Prentice Hall PTR, 1998.

THE LINUX Documentation Project. Disponível em: <<http://www.tldp.org>>. Acesso em: 29 jun. 2003.

WRIGHT, Gary R.; STEVENS, W. Richard TCP/IP Illustrated, Volume 2: The Implementation. Massachusetts, EUA: Addison-Wesley Publishing Company, 1995. ISBN 0-201-63354-X.

TANENBAUM, Andrew S. Computer Networks. 3. ed. Prentice-Hall.

YOSHIRO, Ben. Make - a tutorial. Honolulu: College of Engineering of the University of Hawaii, 2001. Disponível em: <<http://www.eng.hawaii.edu/Tutor/Make/>>. Acesso em: 02 jan. 2004.

Apêndice A – SESP Socket API

```

/* -----
   Rafael Henchen
   Jan/2004
   Socket Interface (socket.h)
   ----- */

#ifndef _SESP_SOCKET_H_
#define _SESP_SOCKET_H_

#ifndef _BSD_SOURCE
#define _BSD_SOURCE 1
#endif
#include <features.h>
#include <sys/types.h>
#include <sys/socket.h>

#ifndef _KERNEL

#include <sys/cdefs.h>

/*
 * Types
 */
#ifndef SOCK_STREAM
#define SOCK_STREAM 1 /* stream socket */
#define SOCK_DGRAM 2 /* datagram socket */
#define SOCK_RAW 3 /* raw-protocol interface
 */
#define SOCK_RDM 4 /* reliably-delivered
message */
#define SOCK_SEQPACKET 5 /* sequenced packet
stream */
#endif
#define SOCK_SESSTREAM 6 /* stream socket with
session */
#define SOCK_SESDGRAM 7 /* datagram socket with
session */

__BEGIN_DECLS
int sp_accept (int, struct sockaddr *, socklen_t *);
int sp_bind (int, const struct sockaddr *, socklen_t);
int sp_connect (int, const struct sockaddr *, socklen_t);
int sp_getpeername (int, struct sockaddr *, socklen_t *);
int sp_getsockname (int, struct sockaddr *, socklen_t *);
int sp_getsockopt (int, int, int, void *, socklen_t *);

```

```

int sp_listen (int, int);
ssize_t sp_recv (int, void *, size_t, int);
ssize_t sp_send (int, const void *, size_t, int);
int sp_setsockopt (int, int, int, const void *, socklen_t);
int sp_shutdown (int, int);
int sp_socket (int, int, int);
int sp_socketpair (int, int, int, int *);

/* -----
   ----- SESP Additional Socket Procedures -----
   ----- */

/* Send Typed Data
   param: int sockfd, const void *buff, size_t nbytes, int flags
   return: number of bytes written if OK, -1 on error
   */
ssize_t sp_send_typed (int, const void *, size_t, int);

/* Ask for Session Token
   param: int sockfd, int tokenid
   return: 0 if OK, -1 on error
   */
int sp_token_please (int, u_short);

/* Give Session Token
   param: int sockfd, int tokenid
   return: 0 if OK, -1 on error
   */
int sp_token_give (int, u_short);

/* Give All Session Tokens
   param: int sockfd
   return: 0 if OK, -1 on error
   */
int sp_control_give (int);

/* Send Synchronization Point
   param: int sockfd
   return: 0 if OK, -1 on error
   */
int sp_sync (int);

/* Ask for resynchronization with syncpoint
   param: int sockfd, int syncpoint
   return: 0 if OK, -1 on error
   */

```

```

int  sp_resync (int, int);

/* Check for session token control requests
   param: int sockfd, int *sesservice, int *tknvalue
   return: number of bytes in token if request arrived, 0 if
   no requests arrived, -1 on error
   */
int  sp_rcv_token (int, int *,int *);

/* Check for session sync point signal arrival
   param: int sockfd, int *syncpoint
   return: number of bytes of syncpoint if sync arrived, 0 if
   no sync points arrived, -1 on error
   */
int  sp_rcv_sync (int, int *);

/* ----- Not implemented procedures ----- */

/* Send Expedited Data
   param: int sockfd, const void *buff, size_t nbytes, int flags
   return: number of bytes written if OK, -1 on error
   */
ssize_t  sp_send_exp (int, const void *, size_t, int);

/* Send Activity Start Signal
   param: int sockfd
   return: 0 if OK, -1 on error
   */
int  sp_actvt_start (int);

/* Send Activity End Signal
   param: int sockfd
   return: 0 if OK, -1 on error
   */

int  sp_actvt_end (int, int);

/* Send Activity Discard Signal
   param: int sockfd
   return: 0 if OK, -1 on error
   */
int  sp_actvt_discard (int);

/* Send Activity Interrupt Signal
   param: int sockfd
   return: 0 if OK, -1 on error
   */

```

```

int sp_actvt_interrupt (int);

/* Send Activity Resume Signal
   param: int sockfd
   return: 0 if OK, -1 on error
*/
int sp_actvt_resume (int);

__END_DECLS

#endif /* !_KERNEL */

#endif /* !_SESP_SOCKET_H_ */

/* -----
   Rafael Henchen
   Jan/2004
   Socket Auxiliary Structures (socket_var.h)
   ----- */

#ifndef _SESP_SOCKET_VAR_H_
#define _SESP_SOCKET_VAR_H_

#ifndef _BSD_SOURCE
#define _BSD_SOURCE 1
#endif
#include <features.h>
#include <sys/types.h>
#include <sys/socket.h>

#include "sesp_var.h"

#ifndef _KERNEL

#include <sys/cdefs.h>

struct socket {
    int so_fd; /* socket file
descriptor */
    short pr_family; /* domain protocol a
member of */
    short pr_type; /* socket type used for
*/

```

```

    short          pr_protocol;          /* protocol number */
    struct socket  *so_next;
    struct socket  *so_prev;
    struct sespcb  *so_spcb;
};

/* Protocol Option Requests - copy from BSD 4.4 Source
Code in <sys/protosw.h>*/

/*
 * The arguments to ctloutput are:
 *      (*protosw[].pr_ctloutput)(req, so, level, optname,
optval, p);
 * req is one of the actions listed below, so is a (struct socket
*),
 * level is an indication of which protocol layer the option is
intended.
 * optname is a protocol dependent socket option request,
 * optval is a pointer to a mbuf-chain pointer, for value-return
results.
 * The protocol is responsible for disposal of the mbuf chain
*optval
 * if supplied,
 * the caller is responsible for any space held by *optval, when
returned.
 * A non-zero return from usrreq gives an
 * UNIX error number which should be passed to higher level
software.
 */

#define PRCO_GETOPT      0
#define PRCO_SETOPT     1
#define PRCO_NCMDS      2

#ifdef PRCOREREQUESTS
char    *prcorequests[] = {
        "GETOPT", "SETOPT",
};
#endif

/* Socket auxiliary procedures */

__BEGIN_DECLS
int socreate(int, struct socket **, int, int);
int sonewconn(int, int, int, int, struct socket **);
void soalloc(struct socket *);

```

```

void soisconnected(struct socket *);
int getsocket(int, struct socket **);
void sofree(struct socket *);
__END_DECLS

#endif /* !_KERNEL */

#endif /* !_SESP_SOCKET_VAR_H_ */

/* -----
Rafael Henchen
Jan/2004
Socket Functions (socket.c)
----- */

#include <stdio.h>

#include <stdlib.h>
#include <sys/errno.h>

#include "socket.h"
#include "socket_var.h"
#include "sesp_var.h"
#include "mbuf.h"

static struct socket *so_first = NULL;

int sp_socket(family, type, protocol)
    int family;
    int type;
    int protocol;
{
    struct socket *so;

    /* Create new socket and associate new SESP control buffer */

    if ( socreate(family, &so, type, protocol) == -1 )
        return -1;

    /* Allocate new socket in the chain */
    soalloc(so);

```

```

printf("SOCKET.c : socket opened = %d \n",so->so_fd);

return so->so_fd;
}

int sp_accept(sockfd,cliaddr,addrlen)
    int sockfd;
    struct sockaddr *cliaddr;
    socklen_t *addrlen;
{
    struct socket *so;
    struct mbuf *buff;

    /* Find the socket by sockfd */
    if ( getsocket(sockfd,&so) == -1 )
        return -1;

    if ( (buff = malloc( sizeof(struct mbuf) )) == NULL )
        return -1;

    buff->mb_pru = PRU_ACCEPT;
    buff->mb_cliaddr = cliaddr;
    buff->mb_socklen = addrlen;

    return sesp_usr_accept(so,buff);
}

int sp_bind(sockfd, myaddr, addrlen)
    int sockfd;
    const struct sockaddr *myaddr;
    socklen_t addrlen;
{
    struct socket *so;
    struct mbuf *buff;

    /* Find the socket by sockfd */
    if ( getsocket(sockfd,&so) == -1 )
        return -1;

    if ( (buff = malloc( sizeof(struct mbuf) )) == NULL )
        return -1;

    buff->mb_pru = PRU_BIND;
    buff->mb_srvraddr = myaddr;
    buff->mb_socklen = &addrlen;
}

```

```

    return sesp_usr_bind(so, buff);
}

int sp_connect(sockfd, servaddr, addrlen)
    int sockfd;
    const struct sockaddr *servaddr;
    socklen_t addrlen;
{
    struct socket *so;
    struct mbuf *buff;

    /* Find the socket by sockfd */
    if ( getsocket(sockfd,&so) == -1 )
        return -1;

    if ( (buff = malloc( sizeof(struct mbuf) )) == NULL )
        return -1;
    buff->mb_pru = PRU_CONNECT;
    buff->mb_srvraddr = servaddr;
    buff->mb_socklen = &addrlen;

    return sesp_usr_connect(so, buff);
}

int sp_getpeername(sockfd, peeraddr, addrlen)
    int sockfd;
    struct sockaddr *peeraddr;
    socklen_t *addrlen;
{
    return getpeername(sockfd, peeraddr, addrlen);
}

int sp_getsockname(sockfd, localaddr, addrlen)
    int sockfd;
    struct sockaddr *localaddr;
    socklen_t *addrlen;
{
    return getsockname(sockfd, localaddr, addrlen);
}

int sp_getsockopt(sockfd, level, optname, optval, optlen)
    int sockfd;
    int level;
    int optname;
    void *optval;
    socklen_t *optlen;

```



```

{
    struct socket *so;
    struct mbuf *buff;

    /* Find the socket by sockfd */
    if ( getsocket(sockfd,&so) == -1 )
        return -1;

    /* Create and set the mbuffer */
    if ( (buff = malloc( sizeof(struct mbuf) )) == NULL )
        return -1;

    buff->mb_pru = PRU_SOCKETOPT;
    buff->mb_socklen = optlen;
    buff->mb_data = optval;

    return sesp_ctloutput(PRCO_GETOPT, so, level, optname, buff);
}

int sp_listen(sockfd, backlog)
    int sockfd;
    int backlog;
{
    struct socket *so;
    struct mbuf *buff;

    /* Find the socket by sockfd */
    if ( getsocket(sockfd,&so) == -1 )
        return -1;

    if ( (buff = malloc( sizeof(struct mbuf) )) == NULL )
        return -1;
    buff->mb_pru = PRU_LISTEN;
    buff->mb_flags = backlog; /* utiliza o mb_flags para passar a
informacao backlog que tbm eh do tipo inteiro*/

    return sesp_usr_listen(so, buff);
}

int sp_close(sockfd)
    int sockfd;
{
    struct socket *so;

    /* Find the socket by sockfd */
    if ( getsocket(sockfd,&so) == -1 )

```

```

        return -1;

    /* PENDENCIA - Tem que verificar se o socket esta sendo usado
    por mais algum processo */

    if ( sesp_usr_detach(so) == -1)
        return -1;

    sofree(so);

    return 0;
}

ssize_t  sp_recv(sockfd, buff, nbytes, flags)
    int sockfd;
    void *buff;
    size_t nbytes;
    int flags;
{
    struct socket *so;
    struct mbuf *mbuf;

    /* Find the socket by sockfd */
    if ( getsocket(sockfd,&so) == -1 )
        return -1;

    if ( (mbuf = malloc( sizeof(struct mbuf) )) == NULL )
        return -1;
    mbuf->mb_pru = PRU_RECV;
    mbuf->mb_data = buff;
    mbuf->mb_nbytes = nbytes;
    mbuf->mb_flags = flags;

    return sesp_usr_rcvd(so, mbuf);
}

ssize_t  sp_send(sockfd, buff, nbytes, flags)
    int sockfd;
    const void *buff;
    size_t nbytes;
    int flags;
{
    struct socket *so;
    struct mbuf *mbuf;

```

```

/* Find the socket by sockfd */
if ( getsocket(sockfd,&so) == -1 )
    return -1;

if ( (mbuf = malloc( sizeof(struct mbuf) )) == NULL )
    return -1;
mbuf->mb_next = NULL;
mbuf->mb_pru = PRU_SEND_NORMAL;
(const void *)mbuf->mb_data = buff;
mbuf->mb_nbytes = nbytes;
mbuf->mb_flags = flags;

return sesp_usr_send(so, mbuf);
}

int sp_setsockopt(sockfd, level, optname, optval, optlen)
    int sockfd;
    int level;
    int optname;
    const void *optval;
    socklen_t optlen;
{
    struct socket *so;
    struct mbuf *buff;

    /* Find the socket by sockfd */
    if ( getsocket(sockfd,&so) == -1 )
        return -1;

    /* Create and set the mbuffer */
    if ( (buff = malloc( sizeof(struct mbuf) )) == NULL )
        return -1;

    buff->mb_pru = PRU_SOCKETOPT;
    (const void *)buff->mb_data = optval;
    buff->mb_socklen = &optlen;

    return sesp_ctloutput(PRCO_SETOPT, so, level, optname, buff);
}

int sp_shutdown(sockfd, howto)
    int sockfd;
    int howto;
{
    struct socket *so;

```

```

/* Find the socket by sockfd */
if ( getsocket(sockfd,&so) == -1 )
    return -1;

/* PENDENCIA - Tem que verificar se o socket esta sendo usado
por mais algum processo */

if ( sesp_usr_shutdown(so,howto) == -1)
    return -1;

sofree(so);

return 0;
}

ssize_t  sp_send_typed (sockfd, buff, nbytes, flags)
    int sockfd;
    const void *buff;
    size_t nbytes;
    int flags;
{
    struct socket *so;
    struct mbuf *mbuf;

    /* Find the socket by sockfd */
    if ( getsocket(sockfd,&so) == -1 )
        return -1;

    if ( (mbuf = malloc( sizeof(struct mbuf) )) == NULL )
        return -1;
    mbuf->mb_next = NULL;
    mbuf->mb_pru = PRU_SEND_TYPED;
    (const void *)mbuf->mb_data = buff;
    mbuf->mb_nbytes = nbytes;
    mbuf->mb_flags = flags;

    return sesp_usr_send(so, mbuf);
}

int  sp_token_please (sockfd, tokenid)
    int sockfd;
    u_short tokenid;
{
    struct socket *so;

```

```

struct mbuf    *mbuf;

/* Find the socket by sockfd */
if ( getsocket(sockfd,&so) == -1 )
    return -1;

if ( (mbuf = malloc( sizeof(struct mbuf) )) == NULL )
    return -1;
mbuf->mb_pru = PRU_PLEASE_TKN;
(u_short *)mbuf->mb_data = tokenid;
mbuf->mb_nbytes = sizeof(u_short);
mbuf->mb_flags = 0;

return sesp_usr_send(so, mbuf);
}

int sp_token_give (sockfd, tokenid)
    int sockfd;
    u_short tokenid;
{
    struct socket *so;
    struct mbuf    *mbuf;

    /* Find the socket by sockfd */
    if ( getsocket(sockfd,&so) == -1 )
        return -1;

    if ( (mbuf = malloc( sizeof(struct mbuf) )) == NULL )
        return -1;
    mbuf->mb_pru = PRU_GIVE_TKN;
    (u_short *)mbuf->mb_data = tokenid;
    mbuf->mb_nbytes = sizeof(u_short);
    mbuf->mb_flags = 0;

    return sesp_usr_send(so, mbuf);
}

int sp_control_give (sockfd)
    int sockfd;
{
    struct socket *so;
    struct mbuf    *mbuf;

    /* Find the socket by sockfd */
    if ( getsocket(sockfd,&so) == -1 )

```

```

    return -1;

    if ( (mbuf = malloc( sizeof(struct mbuf) )) == NULL )
        return -1;
    mbuf->mb_pru = PRU_GIVE_CTRL;
    mbuf->mb_flags = 0;

    return sesp_usr_send(so, mbuf);
}

int sp_sync (sockfd)
    int sockfd;
{
    struct socket *so;
    struct mbuf *mbuf;

    /* Find the socket by sockfd */
    if ( getsocket(sockfd,&so) == -1 )
        return -1;

    if ( (mbuf = malloc( sizeof(struct mbuf) )) == NULL )
        return -1;
    mbuf->mb_pru = PRU_SYNC;
    mbuf->mb_flags = 0;

    return sesp_usr_send(so, mbuf);
}

int sp_resync (sockfd)
    int sockfd;
{
    struct socket *so;
    struct mbuf *mbuf;

    /* Find the socket by sockfd */
    if ( getsocket(sockfd,&so) == -1 )
        return -1;

    if ( (mbuf = malloc( sizeof(struct mbuf) )) == NULL )
        return -1;
    mbuf->mb_pru = PRU_RESYNC;
    mbuf->mb_flags = 0;

    return sesp_usr_send(so, mbuf);
}

```

```

int sp_rcv_token (sockfd, sesservice, tknvalue)
    int sockfd;
    int *sesservice;
    int *tknvalue;
{
    struct socket *so;
    struct mbuf *mbuf;

    /* Find the socket by sockfd */
    if ( getsocket(sockfd,&so) == -1 )
        return -1;

    if ( (mbuf = malloc( sizeof(struct mbuf) )) == NULL )
        return -1;

    mbuf->mb_pru = PRU_RCV_TKN;
    mbuf->mb_data = tknvalue;
    mbuf->mb_nbytes = sizeof(int);
    mbuf->mb_sesservice = sesservice;
    mbuf->mb_flags = 0;

    return sesp_usr_rcvd(so, mbuf);
}

int sp_rcv_sync (sockfd, syncpoint)
    int sockfd;
    int *syncpoint;
{
    struct socket *so;
    struct mbuf *mbuf;

    /* Find the socket by sockfd */
    if ( getsocket(sockfd,&so) == -1 )
        return -1;

    if ( (mbuf = malloc( sizeof(struct mbuf) )) == NULL )
        return -1;

    mbuf->mb_pru = PRU_RCV_SYNC;
    mbuf->mb_data = syncpoint;
    mbuf->mb_nbytes = sizeof(int);
    mbuf->mb_flags = 0;

    return sesp_usr_rcvd(so, mbuf);
}

```

```

}

/* -----
   ---- Auxiliary Procedures ----
   ----- */

int screate(family, aso, type, protocol)
    int family;
    struct socket **aso;
    int type;
    int protocol;
{
    struct socket *so;

    if ( family != AF_INET ){
        errno = EPROTONOSUPPORT;
        return -1;
    }

    /* Create new SESP Socket */
    if ( ( so = malloc( sizeof(struct socket) ) ) == NULL )
        return -1;
    so->pr_family = family;
    so->pr_type = type;
    so->pr_protocol = protocol;
    so->so_next = NULL;
    so->so_prev = NULL;

    /* Attach new socket to protocol control buffer */
    if ( sesp_usr_attach(so) == -1){
        free(so);
        return -1;
    }
    *aso = so;
    return so->so_fd;
}

/* Create a new empty socket structure */
int sonewconn(sofd, family, type, protocol, newso)
    int sofd;
    int family;
    int type;
    int protocol;
    struct socket **newso;
{
    struct socket *so;

```



```

if ( family != AF_INET ){
    errno = EPROTONOSUPPORT;
    return -1;
}

/* Create new SESP Socket */
if ( ( so = malloc( sizeof(struct socket) ) ) == NULL )
    return -1;
so->so_fd = sofd;
so->pr_family = family;
so->pr_type = type;
so->pr_protocol = protocol;
so->so_next = NULL;
so->so_prev = NULL;

*newso = so;
return so->so_fd;
}

void soisconnected(so)
    struct socket *so;
{
    soalloc(so);
}

/* Allocate new socket in the chain */
void soalloc(so)
    struct socket *so;
{
    struct socket *chain_next;

    if (so_first == NULL)
        so_first = so;
    else {
        chain_next = so_first;
        while ( chain_next->so_next != NULL)
            chain_next = chain_next->so_next;
        chain_next->so_next = so;
        so->so_prev = chain_next;
    }
}

void sofree(sock)
    struct socket *sock;

```

```
{
    if (sock->so_next != NULL)
        (sock->so_next)->so_prev = sock->so_prev;
    if (sock->so_prev != NULL)
        (sock->so_prev)->so_next = sock->so_next;
    free(sock);
}

int getsocket(sockfd,aso)
    int sockfd;
    struct socket **aso;
{
    struct socket *so;

    if ( so_first == NULL){
        errno = EBADF;
        return -1;
    }
    so = so_first;
    while ( so != NULL ){
        if (so->so_fd == sockfd){
            *aso = so;
            return so->so_fd;
        }
        so = so->so_next;
    }
    errno = EBADF;
    return -1;
}
```

Apêndice B – INET SESP

```

/* -----
   Rafael Henchen
   Jan/2004
   SESP Protocol Structures (sesp.h)
   ----- */

#ifndef _NETINET_SESP_H_
#define _NETINET_SESP_H_

#ifndef _BSD_SOURCE
#define _BSD_SOURCE 1
#endif
#include <features.h>
#include <sys/types.h>

/* Session Services */
#define S_CONNECT_R      0x01 /* S_CONNECT Request */
#define S_CONNECT_A      0x02 /* S_CONNECT Response */
#define S_RELEASE        0x03
#define S_U_ABORT        0x04
#define S_P_ABORT        0x05
#define S_NORMAL_DATA    0x06
#define S_EXPEDITED_DATA 0x07
#define S_TYPED_DATA     0x08
#define S_TOKEN_GIVE     0x09
#define S_TOKEN_PLEASE   0x0A
#define S_CONTROL_GIVE   0x0B
#define S_SYNC           0x0C
#define S_RESYNCHRONIZE  0x0D
#define S_ACTIVITY_START  0x0E
#define S_ACTIVITY_END    0x0F
#define S_ACTIVITY_DISCARD 0x10
#define S_ACTIVITY_INTERRUPT 0x11
#define S_ACTIVITY_RESUME 0x12
#define S_MAX             0x13

/*
 * SESP Header
 */
struct sesphdr {
    u_int    sh_sid;           /* session id */
    u_char   sh_serv;         /* session service */
    u_short  sh_sclasses:13, /* in negotiation service classes
 */

```

```

        sh_category:3;          /* INET session-transport category
*/
    u_short sh_protocol;        /* server application protocol,
identified by ??? - NOT IN USE YET */
    u_int    sh_datalen;        /* data's size (in bytes) */
};

/* Session Header Service Classes Constants */
#define SHSC_THALF    0x01          /* Half-Duplex Data
Transfer Mode*/
#define SHSC_DTSYNC    0x02          /* Data Transfer
Synchronization */
#define SHSC_ACTIVITY 0x04          /* Activity Management */
#define SHSC_CLOSE    0x08          /* Negotiated Closure */
#define SHSC_ALL      (SHSC_THALF | SHSC_DTSYNC | SHSC_ACTIVITY |
SHSC_CLOSE);

/* Session Header Categories Constants */
#define SHC_CONOR    0x01          /* connection oriented session
(TCP way) */
#define SHC_NOCON    0x02          /* data transfer with no arrival
garanty (UDP way)*/
#define SHC_PTTCP    0x06          /* pass throught TCP */
#define SHC_PTUDP    0x07          /* pass throught UDP */

/* Token IDs */
#define TKN_DATA    0x01          /* Data Token ID*/
#define TKN_SYNC    0x02          /* Synchronization Token ID*/
#define TKN_ACTVT    0x04          /* Activity Token ID*/
#define TKN_CLOSE    0x08          /* Negotiated Closure Token ID*/
#define TKN_ALL      (TKN_DATA | TKN_SYNC | TKN_ACTVT |
TKN_CLOSE )

/* Socket options OPTNAMEs for level IPPROTO_SESP*/
#define SESP_SID    0x01
#define SESP_SERVCLASS 0x02

/* Extension of netinet/in.h Standard well-defined IP protocols.
*/
#define IPPROTO_SESP 101

#endif

```

```

/* -----
   Rafael Henchen
   Jan/2004
   SESP Auxiliary Structures (sesp_var.h)
   ----- */

#ifndef _NETINET_SESP_VAR_H_
#define _NETINET_SESP_VAR_H_

#ifndef _BSD_SOURCE
#define _BSD_SOURCE 1
#endif
#include <features.h>
#include <sys/types.h>

#include "sesp.h"
#include "socket.h"
#include "socket_var.h"
#include "mbuf.h"

#define TMAX_WAIT 10
#define BUFMAX_SIZE1048576
#define BUFMIN_SIZE146000

struct sespcb {
    u_int    scb_sid;                /* session id */
    u_short  scb_sclasses:13,      /* negotiated session service
classes */
    u_short  scb_category:3;       /* INET transport category */
    u_short  scb_protocol;         /* server application
protocol, identified by known port numbers */
    u_short  scb_tokens;           /* tokens possessed by
session entity */

    struct socket *scb_so;         /* session socket */

    u_int    scb_syncpoint;        /* last synchronization point
sended */
    struct mbuf *scb_syncbuff;     /* first message in session
synchronization buffer */
    u_int    scb_syncbuffsize;     /* actual size of the sync
buffer */
    u_int    scb_syncbufflimit;    /* maximum byte number limit
of sync buffer */

    u_int    scb_datasended;       /* amount of data sended
since the begining of the sessio */

```

```

    u_int    scb_notrcvd;          /* data not received from TCP
*/

    /* Not checked messages queues */
    struct mbuf *scb_qdata;        /* queue for incoming data
*/
    struct mbuf *scb_qsync;        /* queue for incoming sync
points */
    struct mbuf *scb_qtkn;        /* queue for incoming tokens
*/
    struct mbuf *scb_qactvt;      /* queue for incoming
activity messages */
};

/* Session Protocol States */
#define CLOSED          0x01
#define LISTEN         0x02
#define TCPWAIT        0x03
#define SESPCONNWAIT  0x04
#define ESTABLISHED0x05

#ifdef  _KERNEL

#include <sys/cdefs.h>

__BEGIN_DECLS
/* --- pr_usrreqs sesp_usrreqs; */
int sesp_usr_attach(struct socket *);
int sesp_usr_detach(struct socket *);
int sesp_usr_bind(struct socket *, struct mbuf *);
int sesp_usr_listen(struct socket *, struct mbuf *);
int sesp_usr_connect(struct socket *, struct mbuf *);
int sesp_usr_disconnect(struct socket *);
int sesp_usr_accept(struct socket *, struct mbuf *);
int sesp_usr_shutdown(struct socket *, int);
int sesp_usr_rcvd(struct socket *, struct mbuf *);
int sesp_usr_send(struct socket *, struct mbuf *);
int sesp_usr_abort(struct socket *);
int sesp_ctloutput(int, struct socket *, int, int, struct mbuf
*);
__END_DECLS

#endif /* !_KERNEL */

#endif /* !_NETINET_SESP_VAR_H_ */

```

```

/* -----
Rafael Hennen
Jan/2004
SESP User Functions (sesp_usrreq.c)
----- */

#include <stdio.h>

#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <string.h>
#include <sys/time.h>
#include <sys/types.h>

#include "errno.h"
#include "sesp.h"
#include "sesp_var.h"

/* SESP Layer internal procedures */
static int sesp_attach(struct socket *);
static int sesp_send (struct socket *, struct mbuf *);
static int sesp_recv (struct socket *, struct mbuf *);
static int sesp_free(struct sespcb *);

/* Data Not Received pendant - Put incoming received data on
queue */
static int sesp_doqueue_dnr(struct socket *);
/* Verify if have data on queues to delivery according to buff
PDU */
static int sesp_verifyqueues(struct socket *, struct mbuf *);
/* Put next incoming data on proper queue according to sesp
header*/
static int sesp_doqueue(struct socket *, struct sesphdr *);
/* Clean the synchronization buffer */
static void cleanbuffer(struct sespcb *);

/* SESP Layer Global Variables */
//static struct sespcb *spcb_first = NULL;

/* -----
SESP Layer implemented procedures
----- */

```

```

int
sesp_usr_attach(so)
    struct socket *so;
{
    /* Create new System Trasnpot Socket (or BSD NET/3 Socket) */
    if (so->pr_type == SOCK_SESSTREAM){
        if ( (so->so_fd = socket(so->pr_family, SOCK_STREAM, so-
>pr_protocol)) <= 0){
            free(so);
            return -1;
        }
    }
    else if(so->pr_type == SOCK_SESDGRAM){
        free(so);
        errno = EPROTONOSUPPORT;
        return -1;
    }
    else{
        if ( (so->so_fd = socket(so->pr_family, so->pr_type, so-
>pr_protocol)) <= 0){
            free(so);
            return -1;
        }
    }

    return sesp_attach(so);
}

int
sesp_usr_detach(sock)
    struct socket *sock;
{
    struct sespcb *spcb = sock->so_spcb;

    /* If Session Active and Negotiated Closure Active and Not Have
TKN_CLOSE return ERROR Service Not Allowed */
    if ( ( spcb->scb_sid != 0 ) &&
        ( ( spcb->scb_category & SHSC_CLOSE) == SHSC_CLOSE) &&
        ( (spcb->scb_tokens & TKN_CLOSE) != TKN_CLOSE) ) ){
        errno = ESERVNA;
        return -1;
    }

    /* PENDENCIA - Procedimento de S_RELEASE -- Ver c/ Beth e Dayna
*/

```



```

sesp_free(spcb);

/* Normal close transport socket and return */
return close(sock->so_fd);
}

int
sesp_usr_bind(sock, nam)
    struct socket *sock;
    struct mbuf *nam;
{
    int error;

    if (sock->pr_type <= SOCK_SESSTREAM){
        error = bind(sock->so_fd, nam->mb_srvraddr, *(nam-
>mb_socklen));
        free(nam);
        return error;
    }

    if ( bind(sock->so_fd, nam->mb_srvraddr, *(nam->mb_socklen)) ==
-1){
        free(nam);
        return -1;
    }
    free(nam);
    return 0;
}

int
sesp_usr_listen(sock, largs)
    struct socket *sock;
    struct mbuf *largs;
{
    int error;

    if (sock->pr_type <= SOCK_SESSTREAM){
        error = listen(sock->so_fd, largs->mb_flags);
        free(largs);
        return error;
    }

    if ( listen(sock->so_fd, largs->mb_flags) == -1){
        free(largs);

```

```

        return -1;
    }

    free(largs);
    return 0;
}

int
sesp_usr_accept(sock, nam)
    struct socket *sock;
    struct mbuf *nam;
{
    static u_int sidcount = 2004;

    int newsofd;
    int error;
    fd_set rset;
    struct timeval timeout;
    struct socket *newso;
    struct sesphdr conhdr;

    if (sock->pr_type < SOCK_SESSTREAM){
        error = accept(sock->so_fd, nam->mb_cliaddr, nam->mb_socklen);
        free(nam);
        return error;
    }

    /* Accept the establishment of a new TCP connection */
    if ( (newsofd = accept(sock->so_fd, nam->mb_cliaddr, nam->mb_socklen)) == -1){
        free(nam);
        return -1;
    }
    free(nam);

    /* Create a new socket to the new TCP connection accepted */
    if (sonewconn(newsofd, sock->pr_family, sock->pr_type, sock->pr_protocol, &newso) == -1){
        shutdown(newsofd, 2);
        return -1;
    }

    /* Attach new socket on a new SESP control buffer*/
    if (sesp_attach(newso) == -1){
        shutdown(newso->so_fd, 2);
    }
}

```

```

    sofree(newso);
    return -1;
}

/* Specify the properties of the new SESP control buffer based
on parent SESP Control Buffer */
newso->so_spcb->scb_sid = sidcount++;
newso->so_spcb->scb_sclasses = sock->so_spcb->scb_sclasses;
newso->so_spcb->scb_category = sock->so_spcb->scb_category;
newso->so_spcb->scb_protocol = sock->so_spcb->scb_protocol;
newso->so_spcb->scb_tokens = sock->so_spcb->scb_tokens;

/* Wait for the Client's S_CONNECT Request */
FD_ZERO(&rset);
FD_SET(newso->so_fd, &rset);

timeout.tv_sec = TMAX_WAIT; /* Wait a TMAX_WAIT time to the
reception of SESP S_CONNECT PDU */
timeout.tv_usec = 0;

error = select(newso->so_fd + 1, &rset, NULL, NULL, &timeout);
if ( error == -1 ){
    shutdown(newso->so_fd, 2);
    sofree(newso);
    return -1;
}
else if (error == 0){
    shutdown(newso->so_fd, 2);
    sofree(newso);
    errno = ETIMEDOUT;
    return -1;
}

/* Receive the S_CONNECT Request */
if (recv(newso->so_fd, &conhdr, sizeof(struct sesphdr),
MSG_WAITALL) == -1){
    shutdown(newso->so_fd, 2);
    sofree(newso);
    return -1;
}

/* Process S_CONNECT Request and mount PDU for S_CONNECT
Response */
if ( ( conhdr.sh_serv != S_CONNECT_R ) ||
    ( conhdr.sh_category != newso->so_spcb->scb_category ) ||
    ( conhdr.sh_datalen > 0 ) ) {
    shutdown(newso->so_fd, 2);
}

```

```

    sofree(newso);
    errno = ESESPFAULT;
    return -1;
}
newso->so_spcb->scb_sclasses &= conhdr.sh_sclasses;

conhdr.sh_sid = newso->so_spcb->scb_sid;
conhdr.sh_serv = S_CONNECT_A;
conhdr.sh_sclasses = newso->so_spcb->scb_sclasses;
conhdr.sh_category = newso->so_spcb->scb_category;
conhdr.sh_protocol = newso->so_spcb->scb_protocol;
conhdr.sh_datalen = 0;

/* Send the S_CONNECT Response */
if ( send(newso->so_fd,(struct sesphdr *)&conhdr, sizeof(struct
sesphdr),0) == -1){
    shutdown(newso->so_fd,2);
    sesp_free(newso->so_spcb);
    sofree(newso);
    return -1;
}

/* Connection established */
soisconnected(newso);

return newso->so_fd;
}

int
sesp_usr_connect(sock, nam)
    struct socket *sock;
    struct mbuf *nam;
{
    int error;
    fd_set rset;
    struct timeval timeout;
    struct sesphdr conhdr;

    if (sock->pr_type < SOCK_SESSTREAM){
        error = connect(sock->so_fd, nam->mb_srvraddr, *(nam-
>mb_socklen));
        free(nam);
        return error;
    }
}

```

```

/* Establish a TCP connection with the server */
if (connect(sock->so_fd,nam->mb_srvraddr,*(nam->mb_socklen)) ==
-1 ){
    free(nam);
    return -1;
}
free(nam);

/* Mount PDU for S_CONNECT Request */
conhdr.sh_sid = 0;
conhdr.sh_serv = S_CONNECT_R;
conhdr.sh_sclasses = sock->so_spcb->scb_sclasses;
conhdr.sh_category = sock->so_spcb->scb_category;
conhdr.sh_protocol = sock->so_spcb->scb_protocol;
conhdr.sh_datalen = 0;

/* Send the S_CONNECT Request */
if ( send(sock->so_fd,(struct sesphdr *)&conhdr, sizeof(struct
sesphdr),0) == -1){
    shutdown(sock->so_fd,2);
    sesp_free(sock->so_spcb);
    sofree(sock);
    return -1;
}

/* Receive the S_CONNECT Response */
FD_ZERO(&rset);
FD_SET(sock->so_fd, &rset);

timeout.tv_sec = TMAX_WAIT; /* Wait a TMAXX_WAIT time to the
reception of SESP S_CONNECT PDU */
timeout.tv_usec = 0;

error = select(sock->so_fd + 1, &rset, NULL, NULL, &timeout);
if ( error == -1 ){
    shutdown(sock->so_fd,2);
    return -1;
}
else if (error == 0){
    shutdown(sock->so_fd,2);
    errno = ETIMEDOUT;
    return -1;
}

if (recv(sock->so_fd, &conhdr, sizeof(struct sesphdr),
MSG_WAITALL) == -1){
    shutdown(sock->so_fd,2);

```

```

    return -1;
}

if ( ( conhdr.sh_serv != S_CONNECT_A ) ||
      ( conhdr.sh_category != sock->so_spcb->scb_category ) ||
      ( conhdr.sh_datalen > 0 ) ) {
    shutdown(sock->so_fd,2);
    errno = ESESPFAULT;
    return -1;
}

/* Set the Session Server Attributes on the cliente SESP control
buffer*/
sock->so_spcb->scb_sid = conhdr.sh_sid;
sock->so_spcb->scb_sclasses = conhdr.sh_sclasses;
sock->so_spcb->scb_protocol = conhdr.sh_protocol;
sock->so_spcb->scb_tokens = 0;

return 0;
}

int sesp_ctloutput(op, so, level, optname, mb)
    int op;
    struct socket *so;
    int level, optname;
    struct mbuf *mb;
{
    int error = 0;
    struct sespcb *spcb = so->so_spcb;

    if (so->pr_type < SOCK_SESSTREAM){
        error = setsockopt(so->so_fd, level, optname ,mb->mb_data,
*(mb->mb_socklen));
        free(mb);
        return error;
    }

    /* If level are not IPPROTO_SESP do normal set/getsockopt()
operation*/
    if (level != IPPROTO_SESP){
        switch(op){
            case PRCO_SETOPT:
                error = setsockopt(so->so_fd, level, optname, mb->mb_data,
*(mb->mb_socklen));
                free(mb);
                return error;

```

```

        break;
    case PRCO_GETOPT:
        error = getsockopt(so->so_fd, level, optname, mb->mb_data,
mb->mb_socklen);
        free(mb);
        return error;
        break;
    }
}

switch(op){
case PRCO_SETOPT:
    switch (optname){
    case SESP_SERVCLASS:
        if (spcb->scb_sid > 0){ /* The Set SERVCLASS operation is
only permitted before connection procediment */
            free(mb);
            errno = EPERM;
            return -1;
        }
        if (*(mb->mb_socklen) > sizeof(u_short)){
            free(mb);
            errno = EOVERFLOW;
            return -1;
        }
        spcb->scb_sclasses = *( (u_short *) mb->mb_data);
        free(mb);
        return 0;
        break;
    default:
        errno = ENOPROTOOPT;
        free(mb);
        return -1;
    }
    break;
case PRCO_GETOPT:
    switch (optname){
    case SESP_SERVCLASS:
        *((u_short *) mb->mb_data) = spcb->scb_sclasses;
        *(mb->mb_socklen) = sizeof(u_short);
        free(mb);
        return 0;
        break;
    case SESP_SID:
        *((u_int *) mb->mb_data) = spcb->scb_sid;
        *(mb->mb_socklen) = sizeof(u_int);
        free(mb);

```

```

        return 0;
        break;
    default:
        errno = ENOPROTOOPT;
        free(mb);
        return -1;
    }
    break;
}
free(mb);
return -1;
}

int
sesp_usr_disconnect(sock)
    struct socket *sock;
{
    errno = ESESPFAULT;
    return -1;
}

int
sesp_usr_shutdown(sock,howto)
    struct socket *sock;
    int howto;
{
    /* PENDENCIA - Verificar protocolo de S_?_ABORT c/ Dayna e
    Beth*/

    seSP_free(sock->so_spcb);
    return shutdown(sock->so_fd,howto);
}

int
sesp_usr_rcvd(sock, buff)
    struct socket *sock;
    struct mbuf *buff;
{
    int size, recvdt = 0;
    struct seSPcb *spcb = sock->so_spcb;
    struct seSPhdr rcvhdr;
    struct mbuf hmb;

    if ( (sock->pr_type <= SOCK_SESSTREAM) && ( buff->mb_pru ==
    PRU_RECV) ){

```



```

        size = recv(sock->so_fd, buff->mb_data, buff->mb_nbytes,
buff->mb_flags);
        free(buff);
        return size;
    }
    else if ( (sock->pr_type <= SOCK_SESSTREAM) && ( buff->mb_pru
!= PRU_RECV) ){
        errno = ESERVNA;
        free(buff);
        return -1;
    }
}

/* If have not received data pendant put them on the scb_qdata
queue */
if (spcb->scb_notrcvd > 0){
    size = sesp_doqueue_dnr(sock);
    if (size == -1){
        free(buff);
        return -1;
    }
    if (spcb->scb_notrcvd > 0)
        return 0;
}

/* If the user required data are in some queue return this data
*/
size = sesp_verifyqueues(sock, buff);
if (size > 0)
    return size;

/* Get SESP PDU Header */
hmb.mb_data = &rcvhdr;
hmb.mb_nbytes = sizeof(rcvhdr);
hmb.mb_flags = 0;

/* Get and Process Data from TCP */
size = sesp_recv(sock, &hmb);
if (size == -1){
    free(buff);
    return -1;
}
else if(size == 0){
    errno = EAGAIN;
    free(buff);
    return -1;
}
}

```

```

/* Verify for valid SESP PDU Header */
if ((rcvhdr.sh_sid != spcb->scb_sid ) ||
    (rcvhdr.sh_serv <= 0) ||
    (rcvhdr.sh_serv > S_MAX) ){
    free(buff);
    errno = ENOESPDT;
    return -1;
}

/* Indicate on the receive memory buffer the incoming service
request*/

switch (buff->mb_pru){
case PRU_RECV:
    if( (rcvhdr.sh_serv & S_NORMAL_DATA ) ||
        (rcvhdr.sh_serv & S_TYPED_DATA ) ||
        (rcvhdr.sh_serv & S_EXPEDITED_DATA ) ){
        spcb->scb_notrcvd = rcvhdr.sh_datalen; /* Tem-se que
garantir que todos os dados enviados serão recebidos corretamente
*/
        if ( buff->mb_nbytes > rcvhdr.sh_datalen ) /* Deve receber
no maximo a quantidade de dados especificada no cabeçalho */
            buff->mb_nbytes = rcvhdr.sh_datalen;
        recvdt = sesp_recv(sock, buff);
        spcb->scb_notrcvd = spcb->scb_notrcvd - recvdt;
    }
    else{
        free(buff);
        if (sesp_doqueue(sock,&rcvhdr) == -1)
            return -1;
    }
    break;
case PRU_RCV_SYNC:
    if( (rcvhdr.sh_serv & S_SYNC ) ||
        (rcvhdr.sh_serv & S_RESYNCHRONIZE ) ){
        if ( buff->mb_nbytes < sizeof(u_short) ){ /* Minimum buffer
to receive SYNC pointssize */
            free(buff);
            errno = EDTOOBL;
            return -1;
        }
        recvdt = sesp_recv(sock, buff);
    }
    else {
        free(buff);
        if (sesp_doqueue(sock,&rcvhdr) == -1)

```

```

    return -1;
}
break;
case PRU_RCV_TKN:
    if( (rcvhdr.sh_serv & S_TOKEN_GIVE ) ||
        (rcvhdr.sh_serv & S_TOKEN_PLEASE ) ||
        (rcvhdr.sh_serv & S_CONTROL_GIVE ) ){
        if ( buff->mb_nbytes < sizeof(u_short) ){ /* Minimum buffer
to receive SYNC pointssize */
            free(buff);
            errno = EDTOOBL;
            return -1;
        }
        recvdt = sesp_recv(sock, buff);
        *(buff->mb_sesservice) = rcvhdr.sh_serv;
        printf("RECV - MB_SESSERVICE %d \n",
            *(buff->mb_sesservice));
    }
    else {
        free(buff);
        if (sesp_doqueue(sock,&rcvhdr) == -1)
            return -1;
    }
    break;
default:
    errno = ESESPFAULT;
    return -1;
    break;
}

printf("RECV - MB_DATA %s\n",
    (char *)buff->mb_data);

return recvdt;
}

int sesp_usr_send(sock, buff)
    struct socket *sock;
    struct mbuf *buff;
{
    int size;
    void *data, *olddata;
    struct sesphdr sphdr;
    struct mbuf *chain_next;
    struct sespcb *spcb = sock->so_spcb;

```

```

    if ( (sock->pr_type <= SOCK_SESSTREAM) && ( buff->mb_pru ==
PRU_SEND) ){
        size = send(sock->so_fd, buff->mb_data, buff->mb_nbytes,
buff->mb_flags);
        free(buff);
        return size;
    }
    else if ( (sock->pr_type <= SOCK_SESSTREAM) && ( buff->mb_pru
!= PRU_SEND) ){
        errno = ESERVNA;
        free(buff);
        return -1;
    }

    if ( buff->mb_nbytes > spcb->scb_synchbufflimit ){
        free(buff);
        errno = EDTOOBL;
        return -1;
    }

    sphdr.sh_sid = spcb->scb_sid;
    sphdr.sh_sclasses = spcb->scb_sclasses;
    sphdr.sh_category = spcb->scb_category;
    sphdr.sh_protocol = spcb->scb_protocol;
    sphdr.sh_dataalen = buff->mb_nbytes;

    /* Process the user request and mount SESP PDU */
    switch (buff->mb_pru){
    case PRU_SEND_NORMAL:
        if ( (spcb->scb_sclasses & SHSC_THALF ) && !(spcb->scb_tokens
& TKN_DATA) ) {
            free(buff);
            errno = ESERVNA;
            return -1;
        }
        sphdr.sh_serv = S_NORMAL_DATA;
        break;
    case PRU_SEND_TYPED:
        sphdr.sh_serv = S_TYPED_DATA;
        break;
    case PRU_SYNC:
        if ( !(spcb->scb_sclasses & SHSC_DTSYNC) ||
( (spcb->scb_sclasses & SHSC_DTSYNC) && !(spcb->scb_tokens
& TKN_SYNC) ) ){
            free(buff);
            errno = ESERVNA;
            return -1;
        }

```

```

    }
    if ( ( buff->mb_data = malloc(sizeof(u_int)) ) == NULL){
        free(buff);
        return -1;
    }
    *((u_int *) (buff->mb_data)) = spcb->scb_datasended;
    sphdr.sh_serv = S_SYNC;
    buff->mb_nbytes = sizeof(u_int);
    break;
case PRU_RESYNC:
    if ( !(spcb->scb_sclasses & SHSC_DTSYNC) ){
        free(buff);
        errno = ESERVNA;
        return -1;
    }
    sphdr.sh_serv = S_RESYNCHRONIZE;
    buff->mb_data = NULL;
    buff->mb_nbytes = 0;
    break;
case PRU_PLEASE_TKN:
    if ( !( *(u_short *) (buff->mb_data) & TKN_ALL) ){
        free(buff);
        errno = ESERVNA;
        return -1;
    }
    sphdr.sh_serv = S_TOKEN_PLEASE;
    buff->mb_nbytes = sizeof(u_short);
    break;
case PRU_GIVE_TKN:
    if ( !(spcb->scb_tokens & *(u_short *) (buff->mb_data)) ){
        free(buff);
        errno = ESERVNA;
        return -1;
    }
    sphdr.sh_serv = S_TOKEN_GIVE;
    buff->mb_nbytes = sizeof(u_short);
    break;
case PRU_GIVE_CTRL:
    if ( !(spcb->scb_tokens & TKN_ALL) ){
        free(buff);
        errno = ESERVNA;
        return -1;
    }
    sphdr.sh_serv = S_CONTROL_GIVE;
    if ( ( buff->mb_data = malloc(sizeof(u_short)) ) == NULL){
        free(buff);
        return -1;
    }

```

```

    }
    *((u_short *)(buff->mb_data)) = TKN_ALL;
    buff->mb_nbytes = sizeof(u_short);
    break;
default:
    free(buff);
    errno = ESESPFAULT;
    return -1;
    break;
}

if (( data = malloc(buff->mb_nbytes + sizeof(sphdr)) ) ==
NULL){
    free(buff);
    return -1;
}

printf("\nSESP_HDR - SID %d, SERV %d, DTLEN %d.\n",
    sphdr.sh_sid,
    sphdr.sh_serv,
    sphdr.sh_datalen);
printf("MB_DATA %s\n", (char *)buff->mb_data);

/* Append the SESP PDU Header on message to send */
memcpy(data, &sphdr, sizeof(sphdr));
memcpy((data + sizeof(sphdr)), buff->mb_data, buff->mb_nbytes);
olddata = buff->mb_data;
buff->mb_data = data;
buff->mb_nbytes = buff->mb_nbytes + sizeof(sphdr);

/* Send the SESP header + user request */
size = sesp_send(sock, buff);
if (size == -1){
    free(data);
    free(buff);
    return -1;
}

/* ----- Tratamentos após PRU enviada ----- */

printf("PRU_SEND %d\n\n", ((struct sesphdr *)buff->mb_data)-
>sh_datalen);

/* Remove tokens after token sended */
if ( buff->mb_pru == PRU_GIVE_TKN )
    spcb->scb_tokens &= ~( *((u_short *)olddata) );
/* Remove all tokens after token sended */

```

```

else if ( buff->mb_pru == PRU_GIVE_CTRL )
    spcb->scb_tokens = 0;
/* Set new syncpoint and clear the buffer on PRU_SYNC */
else if ( buff->mb_pru == PRU_SYNC){
    spcb->scb_syncpoint = spcb->scb_datasended;
    cleanbuffer(spcb);
}
/* Incremente the scb_datasended on PRU_SEND */
else if (buff->mb_pru & PRU_SEND) {
    spcb->scb_datasended += size;
}
/* If PRU_SEND and data sync enabled and session entity have
synchronization token - then Bufferize the SESP PDU */
else if ( (buff->mb_pru & PRU_SEND) &&
          (spcb->scb_sclasses & SHSC_DTSYNC) &&
          (spcb->scb_tokens & TKN_SYNC) ){
    if (spcb->scb_syncbuff == NULL)
        spcb->scb_syncbuff = buff;
    else{

        /* Dont Let Buffer Exceed Sync Buffer Limit - If Exceed
Discard first mbuf */
        if ( (spcb->scb_syncbuffsize + buff->mb_nbytes ) > spcb-
>scb_syncbufflimit ){
            chain_next = spcb->scb_syncbuff->mb_next;
            free(spcb->scb_syncbuff);
            spcb->scb_syncbuff = chain_next;
        }
        chain_next = spcb->scb_syncbuff;
        while (chain_next->mb_next != NULL)
            chain_next = chain_next->mb_next;
        chain_next->mb_next = buff;
    }
    spcb->scb_syncbuffsize += buff->mb_nbytes;
}
else{
    free(data);
    free(buff);
}

return size;
}

int
sesp_usr_abort(sock)
    struct socket *sock;
{

```

```

    errno = ESESPFAULT;
    return -1;;
}

/* -----
   --- Auxiliary Procedures ---
   ----- */

static int
sesp_attach(so)
    struct socket *so;
{
    struct sespcb *spcb;

    /* Create new SESP Control Buffer*/
    if ( (spcb = malloc(sizeof(struct sespcb)) ) == NULL )
        return -1;
    spcb->scb_sid = 0;
    spcb->scb_sclasses = 0;
    switch ( so->pr_type ) {
    case SOCK_SESSTREAM:
        spcb->scb_category = SHC_CONOR;
        break;
    case SOCK_SESDGRAM:
        spcb->scb_category = SHC_NOCON;
        break;
    case SOCK_STREAM:
        spcb->scb_category = SHC_PTTCP;
        break;
    case SOCK_DGRAM:
        spcb->scb_category = SHC_PTUDP;
        break;
    default:
        free(spcb);
        errno = EPROTONOSUPPORT;
        return -1;
    }

    /* A princípio todos os serviços de sessão estão disponíveis
    aos usuários - Como o usuário irá informar quais os serviços que
    deseja oferecer no lado servidor (fornecedor de serviço)?? */
    spcb->scb_sclasses = SHSC_ALL;
    spcb->scb_tokens = TKN_ALL;
    spcb->scb_protocol = 0;
    spcb->scb_so = so;
    spcb->scb_syncpoint = 0 ;
}

```



```

    spcb->scb_syncbuff = NULL;
    spcb->scb_syncbuffsize = 0;
    spcb->scb_syncbufflimit = BUFMIN_SIZE;
    spcb->scb_datasended = 0 ;
    spcb->scb_notrcvd = 0 ;
    spcb->scb_qdata = NULL ;
    spcb->scb_qsync = NULL ;
    spcb->scb_qtkn = NULL ;
    spcb->scb_qactvt = NULL ;

    so->so_spcb = spcb;

    return spcb->scb_so->so_fd;
}

static int
sesp_send (so, buff)
    struct socket *so;
    struct mbuf *buff;
{
    return send(so->so_fd, buff->mb_data, buff->mb_nbytes, buff->mb_flags);
}

static int sesp_recv (so, buff)
    struct socket *so;
    struct mbuf *buff;
{
    return recv(so->so_fd, buff->mb_data, buff->mb_nbytes, buff->mb_flags);
}

static int sesp_free(spcb)
    struct sespcb *spcb;
{
    struct mbuf *buff;

    /* Clean Sync Buffer */
    while (spcb->scb_syncbuff != NULL){
        buff = spcb->scb_syncbuff;
        spcb->scb_syncbuff = buff->mb_next;
        free(buff);
    }
}

```

```

/* Deallocate SESP control buffer */
(spcb->scb_so)->so_spcb = NULL;
free (spcb);

return 0;
}

/* Data Not Received pendant - Put incoming received data on
queue */
static int sesp_doqueue_dnr(sock)
    struct socket *sock;
{
    int recvdt;
    struct sespcb *spcb = sock->so_spcb;
    struct mbuf *buff, *chain_next;

    printf("sesp_doqueue_dnr()\n");

    if ( (buff = malloc( sizeof(struct mbuf) )) == NULL )
        return -1;

    buff->mb_next = NULL;
    buff->mb_pru = PRU_RECV;
    buff->mb_nbytes = spcb->scb_notrcvd;
    if ( (buff->mb_data = malloc( spcb->scb_notrcvd )) == NULL ){
        free(buff);
        return -1;
    }
    buff->mb_flags = 0;

    recvdt = sesp_recv(sock, buff);
    spcb->scb_notrcvd = spcb->scb_notrcvd - recvdt;
    if ( spcb->scb_qdata == NULL )
        spcb->scb_qdata = buff;
    else{
        chain_next = spcb->scb_qdata;
        while (chain_next->mb_next != NULL ){
            chain_next = chain_next->mb_next;
        }
        chain_next->mb_next = buff;
    }

    printf("doqueue_dnr MB_DATA %s\n", (char *)buff->mb_data);

    return recvdt;
}

```

```

/* Verify if have data on queues to delivery according to buff
PDU */
static int sesp_verifyqueues(sock, buff)
    struct socket *sock;
    struct mbuf *buff;
{
    int recvdt;
    char *newdata;
    struct mbuf *chain_next;
    struct sespcb *spcb = sock->so_spcb;

    printf("sesp_verifyqueue()\n");

    switch (buff->mb_pru){
    case PRU_RECV:
        if ( spcb->scb_qdata == NULL) /* Empty queue return 0 */
            return 0;
        /* Copy the data from queue to buffer */
        chain_next = spcb->scb_qdata;
        /* If size of buffer is less that queue can reallocate the
remaining data */
        if (buff->mb_nbytes < chain_next->mb_nbytes ){
            memmove(buff->mb_data, chain_next->mb_data, buff-
>mb_nbytes);
            recvdt = buff->mb_nbytes;
            chain_next->mb_nbytes = chain_next->mb_nbytes - buff-
>mb_nbytes;
            if ( (newdata = malloc( chain_next->mb_nbytes )) == NULL ){
                free(buff);
                return -1;
            }
            memcpy(newdata, (chain_next->mb_data + buff->mb_nbytes),
chain_next->mb_nbytes);
            free(chain_next->mb_data);
            chain_next->mb_data = newdata;
            free(buff);
            return recvdt;
        }
        break;
    case PRU_RCV_SYNC:
        if ( spcb->scb_qsync == NULL) /* Empty queue return 0 */
            return 0;
        chain_next = spcb->scb_qsync;
        if (buff->mb_nbytes < chain_next->mb_nbytes ){
            errno = EDTOOBL;

```

```

        free(buff);
        return -1;
    }
    break;
case PRU_RCV_TKN:
    if ( spcb->scb_qtkn == NULL) /* Empty queue return 0 */
        return 0;
    chain_next = spcb->scb_qtkn;
    if (buff->mb_nbytes < chain_next->mb_nbytes ){
        errno = EDTOOBL;
        free(buff);
        return -1;
    }
    break;
default:
    errno = ESERVNA;
    free(buff);
    return -1;
    break;
}

/* Copy the data from queue to buffer */
memmove(buff->mb_data, chain_next->mb_data, chain_next->mb_nbytes);
recvdt = chain_next->mb_nbytes;
spcb->scb_qdata = chain_next->mb_next;
free(chain_next->mb_data);
free(chain_next);

printf("verifyqueue MB_DATA %s\n", (char *)buff->mb_data);

free(buff);
return recvdt;
}

/* Put next incoming data on proper queue acording to sesp header*/
static int sesp_doqueue(sock, sphdr)
    struct socket *sock;
    struct sesphdr *sphdr;
{
    int recvdt;
    struct sespcb *spcb = sock->so_spcb;
    struct mbuf *buff, *chain_next;

    printf("sesp_doqueue()\n");

```

```

/* Get the message described by sphdr */
if ( (buff = malloc( sizeof(struct mbuf) )) == NULL )
    return -1;
buff->mb_pru = PRU_VOID;
buff->mb_next = NULL;
buff->mb_nbytes = sphdr->sh_datalen;
*(buff->mb_sesservice) = sphdr->sh_serv;
if ( (buff->mb_data = malloc( sphdr->sh_datalen )) == NULL ){
    free(buff);
    return -1;
}
buff->mb_flags = 0;
recvdt = sesp_recv(sock, buff);
if ( (recvdt == -1) || (recvdt == 0) ){
    free(buff->mb_data);
    free(buff);
    return recvdt;
}

/* Alloc the received SESP message on the proper queue */
switch (sphdr->sh_serv){
case S_NORMAL_DATA:
    if ( spcb->scb_qdata == NULL ){
        spcb->scb_qdata = buff;
        return recvdt;
    }
    else
        chain_next = spcb->scb_qdata;
    break;
case S_SYNC:
    if ( spcb->scb_qsync == NULL ){
        spcb->scb_qsync = buff;
        return recvdt;
    }
    else
        chain_next = spcb->scb_qsync;
    break;
case S_TOKEN_PLEASE:
    if ( spcb->scb_qtkn == NULL ){
        spcb->scb_qtkn = buff;
        return recvdt;
    }
    else
        chain_next = spcb->scb_qtkn;
    break;
case S_TOKEN_GIVE:

```

```

    if ( spcb->scb_qtkn == NULL ){
        spcb->scb_qtkn = buff;
        return recvdt;
    }
    else
        chain_next = spcb->scb_qtkn;
    break;
case S_CONTROL_GIVE:
    if ( spcb->scb_qtkn == NULL ){
        spcb->scb_qtkn = buff;
        return recvdt;
    }
    else
        chain_next = spcb->scb_qtkn;
    break;
default:
    errno = ESESPFAULT;
    return -1;
    break;
}

/* If the queue is not empty alloc the buff in the end */
while (chain_next->mb_next != NULL ){
    chain_next = chain_next->mb_next;
}
chain_next->mb_next = buff;

printf("doqueue MB_DATA %s\n", (char *)buff->mb_data);

return recvdt;
}

static void cleanbuffer(spcb)
    struct sespcb *spcb;
{
    struct mbuf *chain_next;

    while (spcb->scb_syncbuff != NULL){
        chain_next = spcb->scb_syncbuff;
        spcb->scb_syncbuff = chain_next->mb_next;
        free(chain_next->mb_data);
        free(chain_next);
    }
    spcb->scb_syncbuffsize = 0;
}
}

```

Apêndice C – Estruturas Auxiliares (mbuf.h e errno.h)

```

/* -----
   Rafael Henchen
   Jan/2004
   SESP Memmory Buffer (mbuf.h)
   ----- */

#ifndef _SESP_MBUF_H_
#define _SESP_MBUF_H_

#ifndef _BSD_SOURCE
#define _BSD_SOURCE 1
#endif
#include <features.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>

#ifndef _KERNEL

#include <sys/cdefs.h>

struct mbuf{
    struct mbuf * mb_next;
    u_short    mb_prui;
    const struct sockaddr * mb_srvraddr;
    struct sockaddr * mb_cliaddr;
    socklen_t * mb_socklen;
    void *      mb_data;
    size_t      mb_nbytes;
    int         mb_flags;
    int         *mb_sesservice;
};

/* Mbuffer types */
#define PRU_VOID    0x00
#define PRU_ATTACH  0x01
#define PRU_BIND    0x02
#define PRU_LISTEN  0x03
#define PRU_ACCEPT  0x04
#define PRU_CONNECT 0x05
#define PRU_SOCKOPT 0x06
#define PRU_SEND_NORMAL 0x07
#define PRU_SEND_EXP  0x08
#define PRU_SEND_TYPED 0x09
#define PRU_RECV      0x0A

```

```

#define PRU_SYNC    0x0B
#define PRU_RESYNC 0x0C
#define PRU_RCV_SYNC    0X0D
#define PRU_PLEASE_TKN 0x0E
#define PRU_GIVE_TKN    0x0F
#define PRU_GIVE_CTRL   0x10
#define PRU_RCV_TKN0x11

#define PRU_SEND    ( PRU_SEND_NORMAL | PRU_SEND_EXP |
PRU_SEND_TYPED )

#endif /* !_KERNEL */
#endif /* !_SESP_MBUF_H_ */

/* -----
Rafael Henchen
Jan/2004
SESP Error Reporting (errno.h)
----- */

#ifndef _SESP_ERRNO_H
#define _SESP_ERRNO_H

#include <errno.h>
#include <asm/errno.h>

/* SESP PROTOCOL ERRORS */

#define ESERVNA          125 /* Error Service Not Allowed - dont
have token permission to use that service */
#define EDTOUBL          126 /* Error Data out of Buffer Limit
*/
#define ENOSESPDT  127 /* Error Invalid SESP PDU Header on
Incoming Data*/
#define ESESPFAULT 150 /* Debug Only - SESP Protocol Error */

#endif

```


Apêndice D – Makefile

```

# Makefile for SESP BSD Socket Dynamic (ELF) Library
VERSION = 0
SUBLEVEL = 1

# Source, Executable, Includes, Library Defines
INCL    = sesp.h sesp_var.h socket.h socket_var.h mbuf.h
SRC     = sesp_usrreq.c socket.c
OBJ     = $(SRC:.c=.o)
LIBS    =
#SONAME = libsesp.so.$(VERSION)
#MYLIB  = $(SONAME).$(SUBLEVEL)
MYLIB   = libsesp.so

# Compiler, Linker Defines
CC      = /usr/bin/gcc
CFLAGS  = -Wall -O2 -fPIC
LIBPATH = -L.
#LDFLAGS = -shared -Wl,-soname,$(SONAME) -o $(MYLIB) $(LIBPATH)
$(LIBS)
LDFLAGS = -shared -o $(MYLIB) $(LIBPATH) $(LIBS)
CFDEBUG = -Wall -g -DDEBUG $(LDFLAGS) -fPIC
RM       = /bin/rm -f

# Compile and Assemble C Source Files into Object Files
%.o: %.c
    $(CC) -c $(CFLAGS) $*.c

# Link all Object Files with external Libraries into Binaries
#$(MYLIB): $(OBJ)
sesp: $(OBJ)
    $(CC) $(LDFLAGS) $(OBJ)

# Objects depend on these Libraries
$(OBJ): $(INCL)

# Create a gdb/dbx Capable Executable with DEBUG flags turned on
debug:
    $(CC) $(CFDEBUG) $(SRC)

# Clean Up Objects, Executables, Dumps out of source directory
clean:
    $(RM) $(OBJ) $(MYLIB) core a.out *.h~ *.c~

```

Apêndice E – Artigo