Sessões de comunicações tolerantes a rupturas:
uma camada de Socket para aplicações cientes
de mobilidade na Internet

Bruno Yuji Lino Kimura

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

Sessões de comunicações tolerantes a rupturas: uma camada de Socket para aplicações cientes de mobilidade na Internet

Bruno Yuji Lino Kimura

Orientador: Prof. Dr. Edson dos Santos Moreira

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Doutor em Ciências - Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA.*

**USP – São Carlos
Dezembro de 2012**

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados fornecidos pelo(a) autor(a)

K49s Kimura, Bruno Yuji Lino
Sessões de comunicações tolerantes a rupturas: uma
camada de Socket para aplicações cientes de
mobilidade na Internet / Bruno Yuji Lino Kimura;
orientador Edson dos Santos Moreira. -- São Carlos,
2012.
100 p.

Tese (Doutorado - Programa de Pós-Graduação em
Ciências de Computação e Matemática Computacional) --
Instituto de Ciências Matemáticas e de Computação,
Universidade de São Paulo, 2012.

1. Sessões tolerantes a rupturas. 2.
Gerenciamento de Mobilidade IP. 3. Aplicações cientes
de mobilidade. 4. API de Sockets. 5. Computação Móvel.
I. Moreira, Edson dos Santos, orient. II. Título.

Para minha esposa, Giselle.

Agradecimentos

Agradeço a Deus e à N.Sra. Aparecida por me manterem perseverante face aos desafios enfrentados.

À minha esposa, Giselle, pelo seu apoio, compreensão, paciência e amor, que são fundamentais na minha vida.

Aos meus pais, Roberto e Cirene, meus irmãos, Gustavo e Rodrigo, e meus padrinhos, José Ramos e Dídima, pelo incentivo e pela confiança.

Aos professores que foram fundamentais nesta trajetória: Prof. Edson Moreira e Prof. Hélio Guardia, pela orientação durante a pesquisa, e Prof. João Carlos Morselli, por me iniciar na ciência lá na graduação.

Aos amigos Bruno Kuehne, Dalton Daher, Danilo Coimbra, Eder Godoy, Paulo Cereda, Rafael Galo, Ricardo Rios, Roberto Rigolin, Roberto Sadao, Rubens Campanini e Thiago Caproni.

Aos amigos e amigas do Departamento de Matemática e Computação da UNIFEI, que me ajudaram indiretamente.

À FAPESP e ao INCT-SEC pelo suporte financeiro.

*There are no limits. There are plateaus,
and you must not stay there, you must go
beyond them ... A man must constantly
exceed his level.*

Bruce Lee

Resumo

Com a heterogeneidade de tecnologias de comunicação sem fio presentes na borda de redes de acesso, serviços providos na Internet podem ser acessados de forma *quasi* ubíqua através de dispositivos móveis ou portáteis. O acesso a esses serviços, contudo, está associado a atrasos e rupturas frequentes na comunicação devido a razões inerentes à mobilidade do dispositivo, como: *i*) perda de sinal em locais onde há pouca ou nenhuma cobertura de acesso móvel; *ii*) erros no quadro de dados durante a transmissão e, conseqüentemente, perdas de pacotes, que podem ser ocasionados por interferência no sinal ou enfraquecimento deste pelo distanciamento do dispositivo em relação à Estação Base; *iii*) mudanças de endereços IP durante transmissões em andamento causadas pela migração do dispositivo entre diferentes redes. Como consequência, aplicações falham com a ruptura de comunicações orientadas a conexão.

Tratar a mobilidade de forma transparente à aplicação é um dos desafios da Computação Móvel e Ubíqua que vem sendo pesquisado ao longo da última década. Soluções foram propostas para operarem desde a Camada de Enlace à Aplicação. Muitas delas, entretanto, exigem modificações na pilha de protocolos TCP/IP e adição de infraestrutura específica de rede no suporte à comunicação fim-a-fim. Além de elevar o custo das etapas de implantação e manutenção, estratégias intrusivas e dependentes de infraestrutura adicional podem não apresentar desempenho satisfatório. Nesse contexto, propomos tratar a mobilidade no nível da própria aplicação através de Sessões de Comunicação que não falham com atrasos e desconexões. Operando somente nos nós-fim e de modo transparente às Camadas adjacentes de Aplicação e Transporte, as sessões não requerem infraestrutura adicional para intermediar ou controlar a comunicação entre pares, tampouco modificações em protocolos legados da pilha TCP/IP.

O conceito de *Sessões Tolerantes a Rupturas* é implementado através de uma API de propósito geral em sistemas Linux que estende a interface de Sockets. A API é, na prática, uma camada transparente sobre o Socket que provê *Ciência de Mobilidade* à aplicação através de mecanismos para: acompanhar a localização de nós ao longo da duração de uma sessão; detectar rupturas nas transmissões causadas pela mobilidade do nó ou de seu par remoto; suspender e retomar sessões de forma eficiente, segura e confiável. Experimentos conduzidos em ambientes emulados e reais com equipamentos de uso comercial mostram a eficiência das sessões. Além de introduzir baixa degradação na vazão fim-a-fim, rupturas na transmissão podem ser detectadas em microssegundos e sessões suspensas são reabertas em milissegundos. Com um desempenho superior a solução de mobilidade geral da Camada IP, as sessões não necessitam de adaptações de software em equipamentos de rede.

Palavras-chave: Computação Móvel, Gerenciamento de Mobilidade IP, Aplicações Cientes de Mobilidade, Sessões Tolerantes a Atrasos e Rupturas, API de Sockets.

Abstract

Nowadays services available on the Internet can be accessed from mobile devices while they roam across heterogeneous wireless networks. Due to the inherent reasons of device mobility, however, the access to such services is frequently involved with delay and disruptions. The most common reasons are: *i)* losing radio signal at places where mobile access coverage area is not available; *ii)* frame error, losses, and fading on the radio signal when the mobile device moves away from the Base Station; *iii)* changes on the device's IP address over ongoing transmission, while the mobile node migrates among different wireless networks. As result, networked application fails with disruptions on TCP connections established in the mobile user's path.

Handling seamlessly mobility on the Internet is a technical challenge of the Mobile Computing Paradigm. It has been widely researched over the last decade. Several solutions have been proposed to work from the Link Layer to the Application Layer. Most of them, however, work intrusively and require modifications in the classical TCP/IP protocol stack, as well as rely on additional network infrastructure to support mobile end-to-end communication. Besides increasing the cost of deployment and maintenance, intrusive and infrastructure dependent strategies may not present suitable performance. In this sense, we devised an architecture to handle mobility at the Application level by means of communication sessions that do not fail with delay, disruption or disconnection. Such sessions work only at the end-systems in a such way that: are fully transparent to the adjacent layers of Transport and Application; do not require additional network infrastructure to forward and manage the communication between two mobile peers; and do not impose any modification on the legacy protocols from the TCP/IP stack.

The concept of Disruption-Tolerant Sessions is implemented in Linux by means of a general purpose API extended from the Socket interface. Such API is a transparent layer placed on top of the Socket to provide mobility awareness to the Application Layer. To do so, session services are provided for: tracking mobile peers along the session duration; detecting disruptions over TCP connection caused by mobility of the local or remote peer; suspending and resuming sessions with efficiency, security and reliability. Experiments conducted in emulated and real systems (off-the-shelf hardware and open source software) showed the desired efficiency. Besides introducing little overhead on the *goodput*, disruptions are detected in a range of microseconds and suspended sessions are resumed in milliseconds. With performance greater than the general IP layer mobility solution, the proposed sessions do not require software adaptation in the core of the network infrastructure.

Keywords: Mobile Computing, IP Mobility Management, Mobility Aware Applications, Disruption-Tolerant Sessions, Sockets API.

Sumário

Resumo	vii
Abstract	ix
Lista de Figuras	xv
Lista de Tabelas	xvii
Lista de Abreviaturas	xix
1 Introdução	1
1.1 Problema	2
1.2 Objetivo	4
1.3 Hipótese	4
1.4 Motivações	4
1.5 Organização do texto	6
2 Mobilidade na Internet	7
2.1 Protocolos de mobilidade baseados em infraestrutura de rede	8
2.1.1 MIPv4	9
2.1.2 MIPv6	10
2.2 Protocolos de mobilidade fim-a-fim	12
2.2.1 HIP	13
2.2.2 TCP-Migrate	17
2.2.3 ROCKS	19
2.2.4 TIPS	21
3 Resultados Preliminares	23
3.1 Experimentos iniciais	23
3.1.1 Configuração do <i>testbed</i>	24

3.1.2	Latência de <i>handovers</i>	25
3.1.3	Análise de desempenho	27
3.1.4	Motivação para um gerenciamento otimizado	28
3.2	Definição dos requisitos para o gerenciamento de mobilidade fim-a-fim	29
4	Sessões de Comunicações Tolerantes a Rupturas	33
4.1	Abstração de sessões de comunicação	34
4.1.1	Uma camada de Socket de mobilidade	35
4.1.2	Elementos de uma Sessão	36
4.1.3	Informação de Sessão	37
4.2	Confiabilidade no nível da Aplicação por meio de <i>bufferização</i> no envio	41
4.2.1	Envio e Recebimento de mensagens na Aplicação	41
4.2.2	Perda de dados durante rupturas	43
4.2.3	<i>Bufferização</i> no envio	44
4.2.4	Recuperação e reenvio dos dados perdidos	46
4.3	Gerenciamento de Localização com o uso de DHTs	48
4.3.1	A interface BambooDHT	49
4.3.2	Registro de Localização	50
4.3.3	Consulta e atualização de registros	51
4.3.4	Detectando a presença de roteadores NAT	52
4.4	<i>Handshaking</i> e autenticação mútua de desafio-resposta	55
4.4.1	Gerando o Desafio	55
4.4.2	Computando a Resposta	57
4.4.3	Abertura de Sessões Fechadas	58
4.4.4	Reabertura de Sessões Suspensas	60
4.5	Gerenciamento de gatilhos de mobilidade	62
4.5.1	Sincronização baseada em Estado de Enlace e Estado de Sessão	62
4.5.2	Detecção de Mudança de Estado de Enlace - LSCD	64
4.5.3	Detecção de Par Indisponível - DPD	66
4.6	Implementação	67
4.6.1	Incorporando as Sessões no mecanismo TIPS	68
4.6.2	Vetor de Sessões	70
4.6.3	Encapsulamento da mensagem de controle	72
4.6.4	As bibliotecas e os serviços providos	72
5	Resultados obtidos	75
5.1	Latência de operações	75
5.1.1	Cenário e equipamentos utilizados	75
5.1.2	Abertura de Sessão Fechada	76
5.1.3	Gerenciamento de Localização	77
5.1.4	Reabertura de Sessão Suspensa	78

5.1.5	Avaliação do <i>handshaking</i>	79
5.2	Sobrecarga do mecanismo	81
5.2.1	Cenário emulado	82
5.2.2	Impacto na vazão fim-a-fim	83
6	Conclusões	85
6.1	Contribuições	86
6.2	Publicações geradas	88
6.3	Trabalhos futuros	89
6.3.1	Mobilidade do servidor entre redes privadas	89
6.3.2	Proatividade e IEEE 802.21	89
6.3.3	Otimização de transmissões após <i>handovers</i>	89
6.3.4	<i>Multihoming</i> e <i>Multipathing</i>	90
6.3.5	Cenários de comunicação altamente dinâmicos	90
	Referências Bibliográficas	93
	Glossário	99

Lista de Figuras

1.1	Mobilidade na Internet: (a) Cenário em que o nó móvel (MN) se comunica com um nó correspondente (CN) através de uma conexão TCP e, em meio a essa conexão, o nó MN é migrado entre diferentes Redes de Acesso Sem Fio (RASf); (b) Relação temporal entre os eventos no decorrer do <i>handover</i> entre duas Redes de Acesso Sem Fio.	3
2.1	Relação entre as soluções de mobilidade pertinentes a este trabalho classificadas por: gerenciamento, implementação e camada na pilha de protocolos.	8
2.2	Comunicação entre MN (<i>Mobile Node</i>) e CN (<i>Correspondent Node</i>) através dos Agentes de Mobilidade HA (<i>Home Agent</i>) e FA (<i>Foreign Agent</i>) no protocolo MIPv4 [45].	10
2.3	Otimização de Rotas no MIPv6.	11
2.4	Tunelamento Bidirecional no MIPv6.	11
2.5	Camada de Identificação HIP na Pilha de Protocolos TCP/IP [6].	14
2.6	Estabelecimento de comunicação HIP entre os nós I (<i>Initiator</i>) e R (<i>Responder</i>).	14
2.7	Atualização de Localização após mobilidade do nó I.	16
2.8	Migração de Conexões TCP [56].	18
2.9	Estados de um socket ROCKS em uma conexão TCP [69].	19
2.10	Comunicação entre Cliente e Servidor com TIPS [28].	22
3.1	Topologia do <i>Testbed</i> [32].	24
3.2	Histograma dos tempos de: (a) L_{TIPS} e (b) L_{MIPv6} [32].	27
3.3	Análise de Desempenho [32]. O desempenho é dado por $P = N/L$, onde N é a latência esperada perto de um valor ideal (assume-se $N = 500$ milissegundos), e L são as latências ordenadas ascendentemente.	28
4.1	Sessões Tolerantes a Rupturas [29].	35
4.2	Arquitetura: Uma camada de Socket de mobilidade.	36
4.3	Uso dos <i>flags</i> de controle durante a reabertura de uma sessão suspensa.	40
4.4	<i>Buffers</i> de envio e recebimento do Socket.	42
4.5	Envio confiável e tolerante a ruptura na Aplicação através da primitiva <code>m_send()</code>	46

4.6	Escrita no <i>buffer</i> da sessão.	46
4.7	Recuperação e reenvio dos dados perdidos com a primitiva <code>mt_recoverylostdata()</code>	47
4.8	Leitura a partir do <i>buffer</i> de Sessão.	47
4.9	Registro de Localização [29].	50
4.10	Gerenciamento de Localização em redes IPv4 [29].	51
4.11	Uso de registros de Localização no suporte à travessia NAT com a técnica <i>TCP Hole Punching</i> [59] entre os pares A e B em uma aplicação P2P.	54
4.12	Autenticação Mútua de Desafio-Resposta, RFC 6287 [43].	55
4.13	Geração do Desafio Q pelo par desafiante [31].	56
4.14	Verificação da assinatura do desafiante pelo desafiado [31].	56
4.15	Computação da Resposta R [31].	57
4.16	Geração do Segredo S no cliente [31].	57
4.17	Verificação da Resposta R [31].	58
4.18	Abertura de Sessão Fechada [31].	59
4.19	Reabertura de Sessão Suspensa [31].	61
4.20	Sincronização por Estado de Enlace e Estado de Sessão.	63
4.21	Monitoramento e Sincronização.	64
4.22	Detecção de Mudança de Estado de Enlace no nó móvel.	65
4.23	Detecção de par indisponível.	67
4.24	Operação TIPS: (a) <i>levemente intrusiva</i> , onde é necessário incluir a biblioteca <code>tips.h</code> e utilizar a funções da API; e (b) <i>Transparente</i> , no qual a biblioteca dinâmica <code>libtips.ld.so</code> é utilizada no filtro de captura de chamadas de sistema da aplicação, as quais são substituídas pelas equivalentes em TIPS.	69
4.25	<code>libtips.ld.so</code> : uma biblioteca dinamicamente carregada e anexada à aplicação alvo com <i>linker/loader ld.so</i> . As <i>syscalls</i> da API de Sockets (<code>socket.h</code>) são filtradas e substituídas pelas equivalentes na API em TIPS (<code>tips.h</code>), como indicam as setas pontilhadas.	70
4.26	Vetor de sessões <code>_session[]</code> : (a) declaração da estrutura em <code>session.h</code> ; (b) visualização do vetor de sessões com o uso de índices locais (descritores de socket).	71
4.27	Informação de Sessão: 41 bytes trocados entre os pares durante a (re) abertura de sessão. (a) Declaração das estruturas em <code>session.h</code> ; (b) Visualização do encapsulamento da Informação de Sessão na pilha de protocolos.	72
4.28	Dependência das bibliotecas do mecanismo de Sessão. Duas bibliotecas compartilhadas são geradas: <code>libtips.so</code> e <code>libtips.ld.so</code>	73
5.1	Probabilidade Cumulativa de operações no Gerenciamento de Localização [31]: (a) Persistentes (P); e (b) Não-Persistente (NP).	77
5.2	Probabilidade Cumulativa de protocolos de apresentação (<i>handshake</i>) [31].	80
5.3	Topologia da rede virtualizada no emulador Netkit.	82
5.4	<i>Goodput</i> da transmissão com e sem o uso das Sessões entre as máquinas cliente e servidor da rede virtualizada ilustrada na Figura 5.3.	83

Lista de Tabelas

2.1	Pacotes trocados durante o <i>handshake</i> HIP [20].	15
3.1	Resultados sumarizados das latências de <i>handover</i> (em segundos) de TIPS e MIPv6 sobre conexões TCP [32].	26
4.1	Notação utilizada na descrição dos <i>buffers</i> de envio e recebimento do Socket (adaptações a partir da terminologia apresentada nos trabalhos [49] e [10]). . .	42
4.2	Interface genérica <i>put/get/remove</i> provida pela BambooDHT [52].	49
4.3	Funções básicas de comunicação em TIPS.	68
5.1	Sumarização dos tempos (em milissegundos) da abertura de sessões fechadas. .	76
5.2	Sumarização dos tempos de resposta (em milissegundos) para a reabertura de sessão suspensa.	78
5.3	Sobrecarga do uso das sessões a partir dos blocos utilizados no envio.	84
6.1	Comparação entre as principais soluções de mobilidade de acordo com requisitos para gerenciamento de mobilidade fim-a-fim.	87

Lista de Abreviaturas

3G	<i>Third Generation</i>
ABC	<i>Always Best Connected</i>
API	<i>Application Programming Interface</i>
BGP	<i>Border Gateway Protocol</i>
CA	<i>Certificate Authority</i>
CN	<i>Correspondent Node</i>
CoA	<i>Care-of-Address</i>
DHCP	<i>Dynamic Host Configuration Protocol</i>
DHT	<i>Distributed Hash Table</i>
DNS	<i>Domain Name System</i>
DPD	<i>Dead Peer Detection</i>
FA	<i>Foreign Agent</i>
FQDN	<i>Fully Qualified Domain Name</i>
HA	<i>Home Agent ou Home Address</i>
HIP	<i>Host Identity Protocol</i>
HIT	<i>Host Identity Tag</i>
HTTP	<i>Hypertext Transfer Protocol</i>
IEEE	<i>Institute of Electrical and Electronics Engineers</i>
IP	<i>Internet Protocol</i>
IPSec	<i>Internet Protocol Security</i>
LSCD	<i>Link State Change Detection</i>
MANET	<i>Mobile Ad-Hoc Network</i>
MIPv4	<i>Mobile Ipv4</i>
MIPv6	<i>Mobile Ipv6</i>
MITM	<i>Man-in-the-middle</i>
MN	<i>Mobile Node</i>
MPTCP	<i>Multi-Path TCP</i>
MSS	<i>Maximum Segment Size</i>
MTTR	<i>Mean-Time-To-Recovery</i>
NAT	<i>Network Address Translation</i>
NEMO	<i>Network Mobility</i>
NGN	<i>Next Generation Networks</i>
OSPF	<i>Open Shortest Path First</i>
P2P	<i>Peer-to-Peer</i>

PEM	<i>Privacy Enhanced Mail</i>
PKI	<i>Public Key Infrastructure</i>
RASF	<i>Redes de Acesso Sem Fio</i>
RIP	<i>Routing Information Protocol</i>
ROCKS	<i>Reliable Sockets</i>
RST	<i>Connection Reset</i>
RVS	<i>Rendezvous Server</i>
SHA-1	<i>Secure Hash Algorithm 1</i>
SIP	<i>Session Initiation Protocol</i>
SNR	<i>Signal-to-noise ratio</i>
SPI	<i>Security Parameters Index</i>
SSH	<i>Secure Shell</i>
SSL	<i>Secure Sockets Layer</i>
TCP	<i>Transmission Control Protocol</i>
TIPS	<i>Transparent IP Sockets</i>
TLD	<i>Top Level Domain</i>
TTL	<i>Time to Live</i>
UML	<i>User Mode Linux</i>
VANET	<i>Vehicular Ad-Hoc Network</i>
VCN	<i>Vehicular Communication Network</i>
WLAN	<i>Wireless Local Area Network</i>
WMAN	<i>Wireless Metropolitan Area Network</i>

Introdução

Há mais de uma década o paradigma da Computação Móvel vislumbrou a hipótese de a atividade computacional ser executada a partir de dispositivos portáteis, permitindo que as pessoas acessassem aplicações e serviços em qualquer lugar e a qualquer momento. O paradigma se tornou predominante e atualmente está presente no cotidiano das pessoas. Há uma necessidade de as pessoas se manterem conectadas o tempo todo aos serviços disponíveis na Web. Isso foi possível devido a fatores, como: a ampliação e oferta de acesso provido por empresas de telefonia móvel; diversidade de tecnologias de transmissão sem fio nas redes periféricas da Internet; e a evolução do hardware e software dos dispositivos móveis, os quais são atualmente computadores pessoais miniaturizados que possuem múltiplas interfaces de comunicação sem fio.

O acesso móvel a partir das redes sem fio na borda da Internet, contudo, está associado a atrasos e rupturas frequentes na comunicação. Isso é devido a razões inerentes à mobilidade do dispositivo, como: *i)* perda de sinal em locais onde não há cobertura de acesso sem fio; *ii)* erros no quadro de dados durante a transmissão e perdas de pacotes, que são ocasionados por interferência no sinal ou seu enfraquecimento com distanciamento do dispositivo da Estação Base; *iii)* mudanças de endereços IP durante transmissões em andamento causadas pela migração do dispositivo entre diferentes redes de acesso. Como consequência, aplicações falham com a ruptura de comunicações orientadas a conexão.

A partir de resultados obtidos de estudos anteriores [26] [28] [27], é apresentado nesta tese um mecanismo de tratamento de mobilidade baseado em sessões de comunicações tolerante a rupturas e desconexões. O enfoque em sessões é devido ao fato de o TCP ser o protocolo de transporte preferencial de aplicações que requerem confiabilidade nas transmissões.

O mecanismo de sessão opera de forma não intrusiva à pilha de protocolos TCP/IP e é apropriado para aplicações baseadas em TCP que requeiram segurança no estabelecimento de conexões, confiabilidade na transmissão fim-a-fim e tolerância a rupturas e desconexões causadas pela mobilidade do nó. As sessões de comunicações possuem parâmetros, os quais são negociados no estabelecimento da sessão entre os nós móveis, bem como serviços que são providos para gerenciar de forma consistente o estado de aplicações na presença de rupturas de conexões. As sessões são implementadas através de uma API de propósito geral em sistemas Linux, a qual permite criar aplicações cientes de mobilidade.

Diferentemente de arquiteturas baseadas em sessões [69] [57] já propostas para a mesma finalidade na literatura, a abordagem de Sessões proposta possui um caráter inovador. Além de agrupar diferentes estratégias de suporte à mobilidade em um único objeto, o conceito de sessão é particularmente útil em limitar o escopo e duração de uma conexão, enquanto preserva a pilha de protocolos TCP/IP intacta [29].

As Sessões possuem funções de comunicações estendidas da interface de Sockets, as quais proveem serviços transparentes no nível da Aplicação para: *i)* rastreamento de nós móveis com o suporte de uma DHT (*Distributed Hash Table*) para recuperar registros de localização armazenados sob identificadores únicos de nó móvel; *ii)* detecção de ruptura de transmissão com o uso de um subconjunto de eventos especificados no padrão IEEE 802.21 [2] para detectar ruptura e restabelecimento do enlace; *iii)* suspensão de sessão através do bloqueio da transmissão da aplicação quando o enlace e a sessão não estiverem preparados para o uso; *iv)* e reabertura de sessões suspensas através da sincronização entre emissor e receptor com informação corrente de sessão em uma nova conexão TCP.

Uma camada abstrata de Sessão foi então elaborada tendo em vista os seguintes princípios de projeto: *a)* minimizar a necessidade de infraestrutura de rede adicional; *b)* prover tratamento à mobilidade com mínimo ou nenhum impacto às camadas adjacentes de Aplicação e Transporte; *c)* obter desempenho similar ou melhor aos protocolos de mobilidade de camada de Rede, como *Mobile IPv6*[24]; e *d)* prover continuação segura de sessão nas retomadas de comunicação após rupturas.

1.1 Problema

A Figura 1.1 ilustra sob duas perspectivas o problema o qual a pesquisa descrita nesta tese se propõe a resolver. A primeira delas, ilustrada na Figura 1.1(a), leva em consideração o cenário: dois nós na Internet, MN (*Mobile Node*) e CN (*Correspondent Node*), se comunicam através de uma conexão TCP; ao longo do percurso do usuário e em meio à conexão, o nó MN é migrado entre diferentes Redes de Acesso Sem Fio (RASf). Tais redes proveem conectividade IP e podem fornecer o acesso através de diferentes tecnologias de transmissão sem fio, o que

implica em *handovers* verticais de nível 3¹. O percurso do usuário e a escolha da rede de acesso de destino, contudo, estão fora do escopo desta tese.

Figura 1.1: Mobilidade na Internet: (a) Cenário em que o nó móvel (MN) se comunica com um nó correspondente (CN) através de uma conexão TCP e, em meio a essa conexão, o nó MN é migrado entre diferentes Redes de Acesso Sem Fio (RAS); (b) Relação temporal entre os eventos no decorrer do *handover* entre duas Redes de Acesso Sem Fio.

A Internet, contudo, não foi projetada para suportar tal mobilidade. Ao unificar a informação de identificação (o nó) e de localização (a rede) no campo de endereçamento, o protocolo IP possui um problema que é de propósito semântico. Uma mudança de rede envolve a mudança de endereço IP, pois o nó móvel é endereçado com um novo endereço IP fornecido pela rede visitada. Observando a operação da camada de Transporte, uma conexão TCP se baseia na relação entre o endereço IP e porta da origem com o endereço IP e porta do destino. A mudança de endereço IP em um dos nós resulta na quebra da conexão estabelecida, o que leva a falha de colapso na aplicação, interrompendo a sua execução.

A segunda perspectiva do problema considera a relação temporal entre os eventos encadeados com a mobilidade. Gerenciando a mobilidade nos nós-fim através do mecanismo de Sessão, a sucessão de eventos pode ser definida conforme a linha do tempo ilustrada na Figura 1.1(b), onde:

t_d : é o tempo de detecção de ruptura entre o início do evento de *handover*, em que o nó móvel é deslocado em direção à nova rede, e o evento de perda de conectividade com a rede anterior (*link down*).

t_{ld} : é o tempo de ruptura do enlace até a retomada de conectividade (*link up*), o que envolve a aquisição de um endereço IP na rede visitada.

¹Ver definição de “*Handover L3*” na Seção Glossário.

t_r : é o tempo de retomada da sessão com o par remoto a partir da nova localização. Isso envolve o restabelecimento da conexão TCP, autenticação entre os nós e sincronização da transmissão com o reenvio de dados perdidos durante o *handover*.

t_{down} : é o tempo total de desconexão com o par remoto, o que inclui a soma dos tempos dos eventos dos itens anteriores.

1.2 Objetivo

Com o propósito de prover *handovers* transparentes ao usuário em trânsito por redes de acesso na borda da Internet, a pesquisa desenvolvida foi direcionada ao projeto, implementação e avaliação de estratégias de Gerenciamento de Mobilidade que tentam minimizar os tempos dos eventos em t_d , t_{ld} , t_r e, conseqüentemente, t_{down} . No contexto da ampla variedade de soluções de mobilidade existentes, o mecanismo de Sessão tem o objetivo de prover suporte eficiente, incorporando novas estratégias e reunindo as melhores técnicas de protocolos de mobilidade fim-a-fim bem conhecidos em uma única solução leve e totalmente implementada no *espaço de usuário* do nó móvel.

1.3 Hipótese

Aplicações distribuídas podem ser projetadas e implementadas para serem cientes de falhas de comunicação [29]. Existem mecanismos de detecção disponíveis na API de Sockets para que uma aplicação perceba a indisponibilidade de um enlace. Assim como a autenticação entre nós em uma comunicação fim-a-fim, poderia ficar à cargo da aplicação tratar eventos como o estouro de temporizadores, restabelecimento de conexão quebradas, eventuais perdas de dados em transmissões não confirmadas, bem como prover serviços de garantia de identificação e localização dos nós móveis envolvidos em uma comunicação.

1.4 Motivações

As Redes de Próxima Geração (*Next Generation Networks* - NGN) são sistemas compostos por redes sem fio de tecnologia heterogênea de acesso, os quais são baseados na infraestrutura IP. Esses sistemas poderão disponibilizar em suas redes periféricas diferentes tecnologias de comunicação sem fio, como os padrões IEEE 802.11 a/b/g/n em redes locais WLAN; IEEE 802.16 d/e e IEEE 802.20 em redes metropolitanas WMAN; as redes celulares 3G; e ainda as redes *ad-hoc* móveis ou veiculares, MANETs e VANETs, respectivamente. Nesse contexto, um dos desafios de pesquisa para os sistemas NGN é o projeto de técnicas inteligentes de Gerenciamento de Mobilidade que utilizam as tecnologias baseadas em IP existentes para permitir mobilidade global em redes de acesso heterogêneo [5].

Com a oferta de múltiplos acessos na borda da Internet, buscar por melhores conexões disponíveis na localização do usuário e, com isso, possibilitar acesso *Always Best Connected* (ABC), é um desafio [21]. Nesse cenário, *handovers* de nível 3 podem ser frequentes durante a busca por melhores conexões.

Recentemente, esforços vêm sendo concentrados no suporte ao NEMO (*Network Mobility*) em Redes de Comunicação Veicular (*Vehicular Communication Networks* - VCNs) [11]. Nessas redes, aplicações baseadas na Internet, como aplicações *infotainment*², são executadas em dispositivos de alta mobilidade, onde, por exemplo, o contato entre veículos em movimento é de curta duração (entre 10 e 40 segundos) [11]. Em cenários altamente dinâmicos, como VCNs, protocolos de Gerenciamento de Mobilidade requerem modificações para operar de forma eficiente na pilha de protocolos veicular, principalmente em termos de otimização de rotas e latência de *handovers* [31].

Nesse contexto, diversos protocolos de Gerenciamento de Mobilidade foram propostos ao longo da última década. Soluções de mobilidade baseadas na camada de Rede, como o protocolo *Mobile IP* [45][46][24], utilizam infraestrutura de rede adicional para encaminhar datagramas para a localização atual do nó móvel. Protocolos de mobilidade fim-a-fim projetados na camada de Transporte são implementados em espaço de *kernel* do nó móvel, como TCP-Migrate [56], HIP [42], e, recentemente, MPTCP [17]. Nesses casos, há a necessidade de modificações em protocolos legados da pilha TCP/IP e, conseqüentemente, implica-se em certas dificuldades de implantação e manutenção da solução.

Por outro lado, soluções de mobilidade de espaço de usuário, como ROCKS [69], SIP [54] e TIPS [28], são implementadas como componentes de software. Além da capacidade de operar de forma transparente à pilha de protocolos TCP/IP, o uso de uma infraestrutura adicional de rede não é necessário, uma vez que nessas soluções a mobilidade é tratada diretamente nos nós-fim. Essas características facilitam as etapas de implantação e manutenção da solução. Entretanto, enquanto soluções de espaço de *kernel* apresentam custos elevados de implantação e manutenção, soluções de espaço de usuário podem apresentar sobrecarga na comunicação devido à cópia de dados entre diferentes espaços em memória. De qualquer forma, independente da estratégia adotada, há de se esperar uma degradação de desempenho ao incluir a operação de um protocolo de mobilidade na pilha TCP/IP.

No âmbito das diversas aplicações e de cenários de comunicação mais variados, observa-se então a necessidade de soluções de mobilidade que sejam eficientes, nesse caso, que possibilitem tempos de desconexão t_{down} mínimos durante rupturas, e, ao mesmo tempo que sejam de fácil implantação, provendo operação transparente de forma não intrusiva aos protocolos legados da pilha TCP/IP.

²Neologismo que combina as palavras *information* e *entertainment*.

1.5 Organização do texto

O restante desta tese está organizado em outros cinco capítulos. O capítulo seguinte apresenta uma revisão da literatura, onde são descritos em maiores detalhes os protocolos de Gerenciamento de Mobilidade que foram fundamentais para o projeto e implementação das Sessões propostas. A apresentação das soluções é dividida em duas partes: protocolos de mobilidade baseados em infraestrutura de rede e protocolos de mobilidade fim-a-fim. O Capítulo discute a escolha de camada de operação, estratégias de implementação e princípios de projeto para soluções de mobilidade fim-a-fim.

O Capítulo 3 traz os resultados preliminares e uma análise de desempenho que compara TIPS [28] [32] e o protocolo MIPv6 [24] a partir de experimentos iniciais realizados em um *testbed IPv6*. O Capítulo também apresenta a definição de requisitos essenciais para gerenciadores de mobilidade fim-a-fim, os quais foram utilizados como princípios de projeto no desenvolvimento das Sessões.

O Capítulo 4 traz em detalhes o funcionamento das Sessões de Comunicação tolerantes a rupturas. São apresentados aspectos fundamentais, como: a abstração de Sessões, a confiabilidade na comunicação fim-a-fim mesmo sob evento de rupturas na transmissão, gerenciamento de localização dos nós móveis, segurança nas etapas de (re) abertura de sessões, gerenciamento de gatilhos sob a ocorrência de eventos detectados com a mobilidade do dispositivo, e aspectos técnicos de implementação.

O Capítulo 5 apresenta resultados obtidos a partir de experimentos conduzidos em ambientes reais e emulados. Os parâmetros de desempenho utilizados na avaliação foram a latência dos serviços de sessão e a sobrecarga das sessões na comunicação fim-a-fim. Uma comparação do desempenho da abertura de sessão é feita com protocolos legados da pilha TCP/IP, como TCP, SSL e SSH.

Por fim, o Capítulo 6 apresenta as conclusões da pesquisa realizada. O Capítulo evidencia as contribuições do mecanismo de sessão proposto através de uma avaliação qualitativa dos principais protocolos de mobilidade. Para tanto, os requisitos definidos no Capítulo 3 são utilizados como parâmetros de comparação entre as soluções. A produção científica gerada e uma perspectiva para trabalhos que merecem atenção futura também são apresentadas.

Mobilidade na Internet

Mobilidade IP é um problema cuja solução não possui lugar bem definido na pilha de protocolos TCP/IP. Ao longo dos últimos anos diversas soluções foram propostas na literatura para operar em diferentes camadas da pilha. Desde o trabalho descrito em [14], a escolha da melhor camada de operação é uma questão ainda em aberto. Ao passo que novas soluções de mobilidade são propostas para operar em camadas superiores, como Transporte, Sessão e Aplicação, otimizações e extensões em protocolos de mobilidade de camada de Rede são amplamente pesquisadas.

Neste Capítulo, os protocolos de Gerenciamento de Mobilidade que influenciaram o projeto e implementação das Sessões são organizados conforme a Figura 2.1. Os aspectos que foram considerados aqui pertinentes definem uma simples classificação quanto:

- à *Implementação*: soluções implementadas em espaço de usuário, ou nas camadas de Sessão ou Aplicação, possibilitam maior facilidade de implantação, uma vez que são componentes de software em sistemas finais. Por outro lado, soluções implementadas em espaço de *kernel*, as quais são também soluções de camadas de Rede ou Transporte, requerem modificações na implementação no nível do Sistema Operacional. Ao necessitar de adaptações para compatibilizar a solução às diferentes versões de *kernel* implica-se também em dificuldades de manutenção.
- à *Pilha de Protocolos TCP/IP*: protocolos de mobilidade IP podem ser implementados para operar na camada de Rede, Transporte, Sessão, Aplicação ou, ainda, em multicamadas em um modelo *cross-layer*. Segundo [14], camadas superiores são apontadas como locais mais apropriados para o tratamento da Mobilidade. Devido ao fato de se

requerer pouca ou nenhuma infraestrutura adicional de rede, o autor aponta a camada de Transporte como a camada ideal. Por outro lado, protocolos legados de transporte, como TCP, o qual é implementado em espaço de *kernel*, precisam ser modificados, o que implica nas dificuldades de implantação nos sistemas finais descritas acima.

- ao *Gerenciamento*: a mobilidade pode ser gerenciada de forma fim-a-fim ou com o auxílio de infraestrutura de rede. Na primeira categoria, os próprios sistemas finais são responsáveis por gerenciar a mobilidade e, com isso, minimiza-se a necessidade de uma infraestrutura adicional de rede. As soluções na segunda categoria, geralmente são implementadas no nível da camada de Rede e dependem de uma infraestrutura específica de suporte. O suporte de uma infraestrutura de rede, contudo, minimiza a complexidade dos nós-fim.

Gerenciamento	Implementação	Pilha TCP/IP	Protocolos
fim-a-fim	Espaço de Usuário	Aplicação	TIPS
		*Sessão	ROCKS
baseado em infraestrutura de rede	Espaço de Kernel	Transporte	TCP-Migrate HIP
		Rede	MIPv4 MIPv6
		Enlace	
		Física	

* Camada abstrata prevista no Modelo de Referência OSI comumente implementada no espaço de usuário, no nível da camada de Aplicação na Pilha de Protocolos TCP/IP.

Figura 2.1: Relação entre as soluções de mobilidade pertinentes a este trabalho classificadas por: gerenciamento, implementação e camada na pilha de protocolos.

A Seção seguinte apresenta os protocolos baseados em infraestrutura, como MIPv4 [46] e MIPv6 [46][24]. Em seguida, a Seção 2.2 descreve em maiores detalhes os principais protocolos de mobilidade fim-a-fim, como HIP [42], TCP-Migrate [56], ROCKS [69] e TIPS [28][27][26].

2.1 Protocolos de mobilidade baseados em infraestrutura de rede

Uma estratégia para gerenciar mobilidade é utilizar *proxy* de redirecionamento. Na camada de Rede, por exemplo, Agentes de Mobilidade são responsáveis por intermediar o fluxo de

datagramas transmitidos do nó correspondente ao nó móvel. Isso permite operação transparente às camadas superiores, como Transporte e Aplicação, além de minimizar a complexidade dos nós-fim. Dessa forma, pacotes perdidos durante *handovers* são detectados e recuperados normalmente pelo TCP.

Por outro lado, aumenta-se a complexidade na borda das redes de acesso, exigindo a infraestrutura adicional de um *proxy* para a entrega de datagramas aos nós móveis visitantes e gerenciamento de localização. Registros de localização e redirecionamento de datagramas para a localização corrente do nó móvel são gerenciados pelos Agentes de Origem (*Home Agent*) no protocolo *Mobile IPv4* (MIPv4) [45][46].

Em redes IPv6 o suporte à mobilidade na camada de Rede é realizado pelo MIPv6 [24]. O trabalho recente descrito em [7] apresenta uma revisão ampla e detalhada do protocolo e suas extensões, como FMIPv6 (*Fast Handover for MIPv6*) [33], HMIPv6 (*Hierarchical MIPv6*) [58] e PMIPv6 (*Proxy MIPv6*) [19].

A seguir, maiores detalhes da operação dos protocolos MIPv4 e MIPv6 são apresentados.

2.1.1 MIPv4

A comunicação entre dois nós-fim, nó móvel MN e nó correspondente CN, através do MIPv4 é ilustrada na Figura 2.2:

1. Datagramas destinados ao nó MN são direcionados à sua Rede de Origem (*Home Network*) através de roteamento IP convencional. O nó CN assume o endereço IP do Agente de Origem HA (*Home Agent*) como IP de destino para atingir MN. Na prática, CN não é ciente da mobilidade do par remoto MN.
2. Os datagramas recebidos são redirecionados para a atual Rede Visitada (*Foreign Network*), onde o nó MN se encontra. Para tanto, um túnel IP sobre IP é criado entre os Agentes HA e FA.
3. Na rede visitada, os datagramas são desencapsulados pelo agente FA e entregues ao nó MN.
4. Para a comunicação no sentido MN-CN, o agente FA atua como um *gateway* padrão e encaminha os datagramas do nó MN ao nó CN através de roteamento convencional.

O nó MN utiliza o seu HA (*Home Address*), que é o endereço IP do agente HA, como endereço IP de Origem para todos os datagramas que envia. Quando o nó CN recebe um datagrama de MN, CN irá assumir o endereço HA como Endereço de Destino de MN. Isso faz com que os datagramas destinados ao nó MN sejam roteados para a sua rede de origem.

Ao ser movido para uma nova rede visitada, o agente FA dessa rede atribui um endereço IP ao nó MN, chamado CoA (*Care-of-Address*). O nó MN então informa a sua atual localização ao agente HA, registrando seu novo CoA. Os registros de localização são realizados através da

Figura 2.2: Comunicação entre MN (*Mobile Node*) e CN (*Correspondent Node*) através dos Agentes de Mobilidade HA (*Home Agent*) e FA (*Foreign Agent*) no protocolo MIPv4 [45].

troca de mensagens do tipo *Registration Request* e *Registration Reply* transportadas via protocolo UDP entre o nó MN e o agente HA. A cada novo registro de localização que chega ao agente HA, um novo túnel IP sobre IP deve ser estabelecido entre os agentes HA e o novo FA.

Embora o protocolo MIPv4 permita a mobilidade transparente às camadas superiores do nó móvel, o uso de um *proxy* introduz ineficiências no protocolo ao redirecionar datagramas para a rede visitada. Os datagramas no sentido CN-MN irão sempre percorrer o caminho não ótimo em uma estratégia de roteamento triangular, gerando impacto negativo na latência de transmissão e vazão fim-a-fim. Além disso, os gerenciamentos de mobilidade e localização são centralizados no agente HA. Se o agente HA falhar, todos seus nós MNs ficarão incomunicáveis. A principal desvantagem é que o uso de agentes para gerenciar a mobilidade no nível de controle de datagramas, ao invés do controle de conexões, impõe a necessidade de uma infraestrutura adicional de rede, elevando os custos da implantação [14].

2.1.2 MIPv6

Na versão do protocolo MIP para redes IPv6 [24] não há a presença do agente FA na rede visitada. Para registrar a localização, o nó MN utiliza o procedimento de *binding* para informar seu endereço temporário CoA ao agente HA. Esse registro também pode ser feito com o nó CN. Nesse caso, com o nó CN ciente da localização do nó MN, o roteamento triangular é evitado.

A operação de *binding* prevê a troca das seguintes mensagens de controle:

- *Binding Update*: o nó MN registra seu CoA com o agente HA e/ou com o nó CN.
- *Binding Acknowledgement*: o agente HA e/ou o nó CN confirmam o recebimento de mensagens *Binding Update* com mensagens de *Binding Acknowledgement* enviadas ao nó MN.

Quando o nó MN é movido para uma nova rede visitada, ele recebe mensagens RA (*ICMP Router Advertisement*) propagadas em difusão pelo roteador local (no caso do MIPv4, essas mensagens são propagadas pelo agente FA). Essa mensagem contém o prefixo da rede IPv6 e, com ele, o próprio nó MN configura seu CoA. O endereço CoA autoconfigurado é a concatenação do prefixo da rede com o endereço MAC do nó MN.

Se o nó CN implementar MIPv6, o nó MN poderá fazer registro de localização também com CN. Uma vez que CN é ciente da mobilidade de MN, o mecanismo de otimização de rotas pode ser utilizado para que MN e CN se comuniquem, evitando o redirecionamento do tráfego para a rede de origem de MN. Nesse caso, o nó CN, antes de enviar um datagrama ao nó móvel, busca localmente o registro entre o *Home Address* e o endereço temporário CoA do nó móvel. Se existir um registro, os datagramas são destinados para o endereço CoA de MN ao invés do endereço HA. O mecanismo de otimização de rotas evita o roteamento triangular e, consequentemente, o agente HA não é mais um gargalo na comunicação fim-a-fim, como ilustra a Figura 2.3.

Figura 2.3: Otimização de Rotas no MIPv6.

Por outro lado, se o nó CN não executar o protocolo MIPv6, todo o tráfego destinado ao nó MN é roteado pelo agente HA, desta vez em um túnel bidirecional IPv6 sobre IPv6, como ilustra a Figura 2.4.

Figura 2.4: Tunelamento Bidirecional no MIPv6.

De todo modo, no protocolo MIPv6 não há a presença do agente remoto FA na rede visitada. Isso minimiza o custo de infraestrutura nas redes de acesso e, conseqüentemente, o custo de implantação. Além disso, com um espaço amplo de 128 bits do endereçamento IPv6, a escassez de endereços IPv4 não é mais um limitador. Se o nó CN implementar MIPv6, pode-se utilizar otimização de rotas; caso contrário, a mobilidade é tratada pelo agente HA na rede de origem.

Em contrapartida, o protocolo MIPv6 requer a implantação de uma rede IPv6 nas bordas de acesso móvel. Tendo em vista que grande parte dos serviços e infraestrutura da Internet é baseada em IPv4, a adoção ao protocolo IPv6 vem sendo realizada lentamente.

2.2 Protocolos de mobilidade fim-a-fim

Mecanismos de mobilidade fim-a-fim podem também ser apoiados por infraestrutura de rede. Uma abordagem semelhante ao MIPv4 é utilizar *proxies*, desta vez, para redirecionar segmentos de Transporte [38]. Se, por um lado, a latência da transmissão fim-a-fim aumenta com o uso do *proxy* intermediando a comunicação entre os nós-fim, por outro, períodos de desconexão durante *handovers* podem ser minimizados com *bufferização* no próprio *proxy* de segmentos destinados ao nó móvel.

Quando a mobilidade é tratada pelos próprios nós-fim, o nó deve estar ciente da mobilidade e localização de seu par. Isso evita a necessidade de *proxy*, mas o tratamento local pode gerar impacto na latência de *handovers*. Para retomar a transmissão, os nós devem recompor a comunicação fim-a-fim nas novas localizações, o que requer registro de localização, principalmente se o nó for um servidor móvel. Após a recomposição, os nós são responsáveis por sincronizar a transmissão e reenviar dados perdidos durante a desconexão.

No trabalho descrito em [55], o desempenho de protocolos de mobilidade fim-a-fim pode ser verificado. Os autores apresentam uma avaliação de desempenho de protocolos baseados em TCP através de modelos analíticos.

Segundo Eddy [14], cada camada na Pilha de Protocolos TCP/IP permite uma operação distinta:

- *camada de Transporte*: o suporte à mobilidade na camada de Transporte é independente do conceito de rede de origem (*home network*) ou infraestrutura adicional de rede. Facilidades como resolução de nomes via DNS e configuração dinâmica de endereço IP via DHCP são partes da infraestrutura da Internet, de modo que não representam infraestrutura adicional para mecanismos de mobilidade fim-a-fim. A otimização de rotas é uma tarefa conduzida normalmente pela camada de Rede, uma vez que a comunicação ocorre entre os nós de forma fim-a-fim. Ao detectar a mobilidade do nó, gerenciadores de mobilidade de camada de Transporte podem interromper as transmissões em andamento para minimizar a perda de segmentos durante *handovers*.

- *camada de Sessão*: a camada de Sessão possui as mesmas vantagens de soluções projetadas na camada de Transporte. Ainda, a implementação de mecanismos na camada de Sessão pode fornecer o tratamento da mobilidade não intrusivo às camadas adjacentes, de Transporte e Aplicação. Dessa forma, alterações em protocolos legados, como TCP e UDP, são evitadas. Ao detectar a mobilidade do nó, gerenciadores de mobilidade na camada de Sessão podem estabelecer uma nova conexão na camada de Transporte para retomar a transmissão quebrada com o *handover*. Por outro lado, o tempo de sincronização e restabelecimento de sessão entre os nós são sobrecargas que podem estender o período de desconexão.
- *camada de Aplicação*: soluções no nível da Aplicação são independentes de arquitetura de processamento, sistemas operacionais e infraestrutura. Extensões de suporte à mobilidade em APIs, como a API de Sockets, podem ser projetadas para operar de forma transparente à lógica das aplicações [26]. Para prover transparência às aplicações, chamadas de sistemas podem ser interceptadas com o uso de bibliotecas dinâmicas que possuem funções equivalentes com suporte à mobilidade.

A principal vantagem é que nas camadas de Transporte, Sessão ou Aplicação, protocolos requerem pouca ou nenhuma infraestrutura de rede. As Seções seguintes descrevem os protocolos de mobilidade fim-a-fim, os quais foram determinantes no projeto e implementação das Sessões propostas.

2.2.1 HIP

O protocolo HIP (*Host Identity Protocol*) [42] introduz um espaço de nome HI (*Host Identity*) para identificação do nó que é desassociado do endereço IP. Uma camada HIP é inserida entre a camada de Rede e Transporte do nó móvel, como ilustra a Figura 2.5. Um par de chaves pública-privada gerado pelo próprio nó provê a identificação HI. Para manter compatibilidade com a camada de Rede, a partir do HI é gerada a *tag* HIT (*Host Identity Tag*), que consiste de um *hash* de 128 bits sobre a chave pública do nó. O comprimento de 128 bits se torna compatível com o espaço de endereçamento IPv6. Uma identificação de escopo local, chamado LSI (*Local Scope Identity*), de 32 bits pode ser gerada a partir dos últimos 4 bytes da *tag* HIT, tornando o LSI compatível com o espaço de endereçamento IPv4.

Endereços IPs são utilizados como informação de localização do nó móvel, os quais mudam conforme a mobilidade, enquanto a HIT é única para o nó e se mantém inalterada. A partir da separação das informações de localização e de identificação, o protocolo HIP provê suporte à *multihoming*, permitindo que múltiplos endereços IP sejam vinculados a uma única *tag* HIT.

Quando o nó é movido para uma rede visitada, o novo endereço IP obtido nessa rede deve ser registrado pelo nó em um servidor RVS (*Rendezvous*). Esse servidor é responsável por

Figura 2.5: Camada de Identificação HIP na Pilha de Protocolos TCP/IP [6].

gerenciar a localização do nó móvel armazenada sob sua respectiva HIT, resolver consultas de localização baseadas na HIT e auxiliar o início da comunicação entre os nós.

Para que um nó móvel HIP esteja acessível na Internet, pode-se combinar o uso de DNS e RVS [42]. Um servidor DNS irá armazenar o registro do nome FQDN¹ do nó móvel. Esse registro pode armazenar o endereço IP corrente e a HIT do nó móvel, bem como o endereço IP do servidor RVS responsável. Entretanto, sistemas DNS não seriam apropriados para manter atualizado o endereço IP correspondente toda vez que o nó móvel for deslocado por diferentes redes [6]. Nesse caso, o registro de nome em DNS se mantém inalterado, pois irá conter informações estáticas, como o endereço do servidor RVS, que é um nó fixo, e a *tag* HIT do nó móvel consultado. Para tanto, o nó móvel deve inicialmente registrar no servidor DNS de seu domínio o servidor RSV que irá resolver as consultas HIT.

Se um nó I (*Initiator*) deseja iniciar comunicação com um nó R (*Responder*), o nó I irá primeiro resolver o nome do FQDN do nó R via DNS. Com a consulta DNS, I poderá obter a *tag* HIT de R e o servidor RVS correspondente, como ilustra a Figura 2.6. O nó I então envia um pacote I1 contendo a HIT de R ao servidor RVS. O servidor RVS irá atuar como um servidor de encontro, encaminhando o pacote I1 ao nó R na sua atual localização. Ao receber o pacote I1, o nó R então responde com um pacote R1 diretamente ao nó I e o *handshake* é continuado com a troca de pacotes I2 e R2.

Figura 2.6: Estabelecimento de comunicação HIP entre os nós I (*Initiator*) e R (*Responder*).

¹Fully Qualified Domain Name.

Para prevenir ataques MITM (*man-in-the-middle*) e DoS (*Denial-of-Service*) durante o estabelecimento da comunicação o protocolo HIP executa um *handshake* de 4 vias com a troca de pacotes I1, R1, I2 e R2. A Tabela 2.1 apresenta a descrição de cada pacote.

Tabela 2.1: Pacotes trocados durante o *handshake* HIP [20].

Pacote	Descrição
I1:{HIT(i), HIT(r)}	Para iniciar uma comunicação HIP, o nó I deve enviar o pacote I1, o qual contém as <i>tags</i> HIT de ambos os nós I e R. A <i>tag</i> HIT e o endereço de R podem ser obtidos a partir de um serviço DNS e/ou RVS.
R1:{HIT(r), HIT(i), puzzle, DH(r), K(r), sig}	Quando o nó R recebe o pacote I1, ele responde com um pacote R1, o qual contém, respectivamente: as <i>tags</i> HIT de R e I, um desafio, a chave <i>Diffie-Hellman</i> de R, a chave pública de R, e a assinatura digital sobre o pacote.
I2:{HIT(i), HIT(r), solution, DH(i), K(i), sig}	Ao receber o pacote R1, o nó I verifica a assinatura com a chave pública de R. Em caso de sucesso, I responde com o pacote I2, o qual contém, respectivamente: as <i>tags</i> HIT de I e R, a solução para o desafio, a chave <i>Diffie-Hellman</i> de I, a chave pública de I e uma assinatura digital sobre o pacote.
R2:{HIT(r), HIT(i), sig}	Da mesma forma que I, o nó R verifica a assinatura do pacote I2 com a chave pública de R. Em caso de sucesso, o nó R envia a mensagem R2 para finalizar o <i>handshake</i> , a qual contém as <i>tags</i> HIT de R e I, bem como a assinatura digital sobre o pacote.

O desafio é a tarefa de encontrar um valor que produz zeros quando passado pela função *hash* SHA-1. O número de zeros determina a dificuldade, podendo ser custoso para o nó I encontrar o valor correto quando o número de zeros aumenta. A verificação do desafio pelo nó R é realizada em uma única operação *hash*, uma vez que o par conhece a resposta. Isso permite que o nó R ajuste dinamicamente a dificuldade do desafio conforme a carga de pacotes I1 recebidos. Dessa forma, o nó R se protege contra ataques DoS, aumentando a dificuldade do desafio [20].

Após o *handshake* os nós estabelecem uma associação segura SA (*Security Association*) com IPsec através de tunelamento IP. Os datagramas IP então são transmitidos no tunelamento IPsec e encapsulados em pacotes ESP (*Encapsulating Security Payload*). A chave simétrica utilizada na criptografia é compartilhada com os pacotes R2 e I2, os quais realizam a troca de chave *Diffie-Hellman* durante o *handshake*. A tarefa da camada HIP é mapear pacotes ESP que chegam para a *tag* HIT correspondente. Isso é feito de acordo com o valor do campo SPI (*Security Parameters Index*) contido no pacote ESP, o qual é um valor arbitrário de 32 bits que, junto com o endereço IP de destino, identifica unicamente uma associação SA no receptor. Para os pacotes ESP que saem do nó a camada HIP seleciona o endereço de origem e a interface de saída também conforme o valor SPI.

Assumindo I como parceiro móvel na comunicação, após sua mobilidade, I deve informar ao nó R a sua nova localização, como ilustra a Figura 2.7. Quando I é endereçado na nova rede visitada, I então envia pacotes do tipo UPDATE especificando um novo endereço IP no parâmetro LOCATOR. R então irá confirmar o recebimento com pacote UPDATE ACK. Nesse pacote de confirmação, R também verifica a disponibilidade de I na nova localização com o parâmetro

ECHO_REQUEST. O nó I então responde ao ACK ECHO_REQUEST com um pacote de ACK ECHO_REPLY.

Figura 2.7: Atualização de Localização após mobilidade do nó I.

Os nós I e R possuem um par de índices SPI entre eles para identificar a associação SA em cada lado da comunicação. Para reconstruir o tunelamento na nova localização de I, a primeira mensagem de UPDATE inclui o parâmetro ESP_INFO, onde I informa a R seu antigo e novo valor SPI, além de um número de sequência SEQ. A segunda mensagem de UPDATE, R irá confirmar o pacote recebido e informar a I seu antigo e novo valor SPI no campo ESP_INFO. Uma vez que a camada HIP mapeia a *tag* HIT de acordo com o valor SPI contido no pacote ESP e, após a mobilidade, os antigos e novos valores de SPI são conhecidos pelas partes, evita-se gerar uma nova chave simétrica para a sessão ESP.

Opcionalmente, ao invés de recuperar a chave gerada anteriormente em uma sessão ESP, uma nova chave pode ser negociada na nova localização. Ao invés de informar antigos e novos valores de SPI no campo ESP_INFO, a primeira e a segunda mensagem de UPDATE incluem os parâmetros de *Diffie-Hellman* para calcular a nova chave simétrica para ser usada na nova sessão ESP.

O espaço de nomes para identificação desassociada da localização e a comunicação segura são as principais contribuições do protocolo HIP. Nesse esquema, a identificação do nó é independente de seu endereço IP, podendo ainda suportar *multihoming*. A utilização do protocolo HIP, contudo, requer modificação na pilha de protocolos ao introduzir a camada HIP, o que implica em certas dificuldades de implantação nos nós-fim.

Uma vez que a mobilidade é tratada de forma fim-a-fim com mensagens de UPDATE entre os nós, é fundamental que o nó móvel consiga atingir o par remoto após sua mobilidade. Isso implica que R seja executado em um nó fixo e acessível para que I consiga atingi-lo. Portanto, a mobilidade de ambas as partes I e R (*double jump*²) ou outros cenários onde os dois nós perdem a conectividade direta não são suportados [20].

²Ver definição na Seção Glossário.

2.2.2 TCP-Migrate

TCP-Migrate [56] é uma arquitetura fim-a-fim para permitir suporte à mobilidade em conexões TCP. Uma nova opção, chamada *Migrate TCP Option*, é utilizada para possibilitar migrações de conexões TCP enquanto o nó é deslocado entre diferentes redes de acesso na Internet. As conexões são unicamente identificadas através de um *token*, transportado como opção do cabeçalho TCP. Percebe-se então que o *token* possui a mesma finalidade das *tags* HIT do protocolo HIP.

Um *token* T entre os nós i e j numa conexão TCP é dado por $T = \text{SHA1}(N_i, N_j, K)$. O *token* T corresponde aos 64 bits mais significativos do *hash* SHA-1 sobre a concatenação dos números de sequências iniciais N_i e N_j da conexão TCP estabelecida entre i e j , além de uma chave secreta K compartilhada entre eles. A chave secreta K é negociada via protocolo ECDH (*Elliptic Curve Diffie-Hellman*) e computada depois da recepção do segmento SYN/ACK pelos nós i e j . Um *token* então é gerado no estabelecimento da primeira conexão e enviado como opção do protocolo TCP.

No passo 1 da Figura 2.8 é ilustrado o estabelecimento de conexão TCP e, no passo (5), o restabelecimento da conexão depois da mobilidade. No passo 1, o nó móvel envia como opções do segmento SYN a sua chave pública k_m e o *timestamp* T_m , os quais são utilizados pelo nó Fixo para computar localmente a chave secreta K . O mesmo é feito pelo nó Fixo no passo 2, com o envio de sua chave pública k_f e *timestamp* T_f no segmento SYN/ACK. A fase de estabelecimento de conexão permite que os nós verifiquem a compatibilidade com TCP-Migrate entre eles e habilitem a opção *Migrate*. Se um dos pares, por exemplo, o nó fixo, não possuir suporte ao TCP-Migrate, a conexão e a transmissão ocorrem normalmente com o TCP convencional.

Em ambos os nós, a arquitetura TCP-Migrate inclui um novo estado na conexão TCP, chamado MIGRATE_WAIT. Em uma conexão em andamento, cujo estado é ESTABLISHED, a conexão muda para o estado MIGRATE_WAIT ao receber um segmento RST (*Connection Reset*), o qual é enviado pelo par remoto ao ser migrado para uma nova rede. Ao contrário de uma conexão TCP comum, no qual o recebimento do segmento RST finaliza a conexão, o protocolo TCP-Migrate mantém o *status* da transmissão sobre conexão, preservando o número de sequência e os *buffers* de envio e recebimento da conexão TCP. A conexão sai do estado MIGRATE_WAIT ao enviar um segmento SYN *Migrate* ou receber um SYN/ACK.

Após a mobilidade, como mostra o passo 5, o cliente (Móvel) na nova rede visitada retoma a comunicação através do restabelecimento da conexão via *three-way handshake*. Entretanto, o segmento de controle SYN enviado contém a opção *Migrate* com o *token* T que identifica a sessão, a qual é conhecida pelos pares, além de uma requisição R . Essa requisição é computada pelo cliente por $R = \text{SHA1}(N_i, N_j, K, S, I)$, onde N_i e N_j são os números de sequências iniciais, K é a chave secreta, S é o segmento SYN *Migrate* e I é o número de sequência da requisição. Da mesma forma, no destino o servidor computa R , uma vez que os parâmetros

Figura 2.8: Migração de Conexões TCP [56].

são conhecidos por um par confiável na conexão. Para continuar ou não a migração, o servidor então compara sua requisição computada com a requisição recebida.

T e R embutidos no segmento SYN permitem que o servidor Fixo identifique que o estabelecimento da conexão TCP pertence a uma sessão T existente. Uma vez que os números de sequências e os *buffers* são preservados, segmentos eventualmente perdidos durante a mobilidade são retransmitidos normalmente pelo TCP na nova localização do nó móvel. O servidor então responde ao SYN *Migrate* (passo 6) com um segmento SYN/ACK. O número de sequência corresponde ao último byte transmitido e o ACK confirma os bytes recebidos corretamente antes da mobilidade. Finalmente, o cliente Móvel finaliza o *handshake* com um segmento ACK confirmando a sequência de bytes, a qual foi preservada durante a migração.

Na arquitetura TCP-Migrate sistemas DNS são utilizados para gerenciar a localização dos nós móveis servidores, os quais possuem nome FQDN inalterados independentemente da mudança de localização. Ao receber um novo endereço IP em uma rede visitada, o nó móvel atualiza o registro A em seu servidor DNS Autorizado através de *Secure DNS Dynamic Update* [13] [67]. Para evitar que registros sejam mantidos em *cache* em sistemas na borda da Internet, o registro A do nó móvel armazenado no servidor DNS Autorizado possui o valor TTL (*Time to Live*) igual a zero. Isso faz com que o nó requisitante sempre consulte o servidor DNS Autorizado apropriado para resolver a localização atualizada do nó móvel.

Nesse caso, uma consulta DNS que não está em *cache* iria percorrer a hierarquia DNS, sendo direcionada inicialmente para o servidor DNS raiz. O servidor raiz através do registro NS (*Name*

Server) do domínio consultado iria indicar o servidor TLD (*Top Level Domain*) responsável pelo domínio. Esse último então iria indicar o servidor Autorizado responsável pelo registro A do nó móvel, o qual iria resolver a consulta do nome pelo endereço IP correspondente. Entretanto, registros NS possuem TTL superior a zero, o que permite mantê-los em *cache* nos sistemas da borda, pelo menos por horas. Nesse caso, a consulta DNS para um nó móvel alvo poderia ser direcionada ao servidor Autorizado responsável, evitando passar pelo servidor raiz e TLD na hierarquia.

Se por um lado a arquitetura TCP-Migrate trata a mobilidade somente nos nós-fim de forma transparente às camadas de Rede e Aplicação, além de não adicionar infraestrutura de rede para esse tratamento, por outro, ela modifica a implementação do protocolo TCP, o que implica em certas dificuldades na implantação. Uma vez que o protocolo TCP é implementado em espaço de *kernel*, nos nós-fim envolvidos, haverá necessidade de recompilação de *kernel* com as modificações sobre o TCP.

A mobilidade do servidor somente é possível se o nó móvel for endereçado na nova rede visitada com um endereço IP roteável na Internet. Caso o nó for migrado para uma rede privada, mesmo com o uso de DNS, o servidor estará escondido por trás de roteadores NAT, impossibilitando a reconexão do cliente.

Há algumas implicações quanto à segurança. Após o envio de um segmento SYN *Migrate*, o *token T* será conhecido por qualquer nó no novo caminho fim-a-fim. Ataques de negação de serviços podem ser disparados com o envio de segmentos SYN *Migrate* falsos, mas com *tokens* válidos. As requisições *R*, por outro lado, permitem que o servidor autentique o cliente, o que inibe parcialmente ataques de sequestro de sessão iniciados por clientes falsos. Entretanto, a autenticação é unilateral. Isso torna o cliente susceptível à servidores falsos, além de ataques de MITM (*man-in-the-middle*).

2.2.3 ROCKS

A solução ROCKS (*Reliable Sockets*) [69] é implementada através de uma biblioteca que opera entre a Aplicação e o *kernel* em ambos nós-fim. Essa biblioteca exporta a API de Sockets para o código da aplicação para gerenciar mobilidade. A Figura 2.9 ilustra os estados de um socket ROCKS durante uma conexão TCP.

Figura 2.9: Estados de um socket ROCKS em uma conexão TCP [69].

Para estabelecer uma conexão TCP, a aplicação utiliza normalmente as primitivas da API de Sockets. Entretanto, as chamadas, ao invés de serem tratadas pelo *kernel*, são tratadas inicialmente pela biblioteca ROCKS. No estabelecimento de conexão são executados os seguintes passos:

1. Teste de Interoperabilidade: para verificar se o servidor é compatível com socket ROCKS. Após o estabelecimento da conexão TCP, o cliente ROCKS envia uma mensagem EDP (*Enhancement Detection Protocol*). Caso o servidor não seja compatível, a biblioteca ROCKS no cliente reverte o socket para operação padrão.
2. Conexão de dados: a conexão de dados é a conexão TCP que, após a conexão ROCKS ser estabelecida, é usada para transmissão de dados da aplicação.
3. Inicialização: um identificador para a conexão é criado com base nos endereços dos pares e em um *timestamp*; uma chave é negociada via *Diffie-Hellman* para autenticação posterior; os nós informam um ao outro o comprimento de seus *buffers* do socket localizados no espaço de *kernel*³.
4. Controle do socket: um canal de controle UDP dedicado é criado entre os pares para a sinalização, onde há troca de mensagens para notificação de falhas sobre a conexão de dados.

Após esses passos, o socket ROCKS muda para o estado CONNECTED. Uma vez conectado os pares se comunicam enviando e recebendo dados como em um socket convencional. Todo dado enviado é copiado para um *buffer* redundante na biblioteca ROCKS. O comprimento desse *buffer* corresponde à soma dos comprimentos dos *buffers* de envio e recebimento do socket. Os pares contabilizam a quantidade de dados enviados e recebidos.

Falhas de conexões são detectadas com *heartbeat probes* periodicamente enviados sobre o canal de controle UDP para o par remoto. Esse mecanismo permite a detecção de indisponibilidade do par na ordem de segundos. Com a ausência de respostas a um número de *probes* sucessivos, o estado do socket é alterado para SUSPENDED.

Automaticamente, o cliente tenta estabelecer uma nova conexão de dados (TCP) com o servidor. Uma vez conectados, os pares se autenticam mutuamente através de um protocolo de desafio-resposta baseado na chave secreta negociada durante a fase de inicialização. Um novo canal UDP para controle também é estabelecido. Os dados eventualmente perdidos durante desconexão são recuperados do *buffer* redundante e retransmitidos. A quantidade de dados perdidos é conhecida pelos pares, pois os mesmos trocam a quantidade de bytes enviados e recebidos pelo novo canal de controle.

³Com a primitiva `getsockopt()` da API de Socket é possível obter parâmetros do descritor de socket na Aplicação.

O restabelecimento da conexão de dados, contudo, falha quando: 1) o servidor mudar de localização durante o período de desconexão; 2) houver um *firewall* na rede de destino bloqueando a tentativa de conexão; 3) ou servidor estiver localizado em uma rede privada, sendo ofuscado por um roteador NAT. Após a reconexão de dados, um socket SUSPENDED somente é retomado após autenticação mútua baseada em desafio-resposta e chave secreta. Entretanto, a fase de inicialização, onde a chave secreta é negociada via *Diffie-Hellman*, está sujeita a ataques MITM. Além disso, o uso de um canal específico para controle implica em sobrecarga de sinalização.

2.2.4 TIPS

Transparent IP Sockets (TIPS) [28][27][26] inicialmente foi uma solução baseada em uma adaptação da API de Sockets para o tratamento reativo da mobilidade IPv4 conduzido na própria aplicação. Primitivas de comunicação da API de Sockets, adaptadas em TIPS por meio de funções *wrapper*, proviam suporte transparente à lógica da aplicação tratando os erros gerados sobre o descritor de socket após a mobilidade.

Os serviços providos consistiam em [26]:

- i) Identificação Independente de Localização: um mecanismo de identificação única do nó móvel com significado para camada de Aplicação e desassociada da camada de Rede.
- ii) Registro de Localização Escalável: utilização de um mecanismo escalável de armazenamento de registro, consulta e atualização de localização dos terminais móveis através de uma DHT.
- iii) Baixo custo de implantação: nos nós-fim, requer pouca modificação na sintaxe do código de Aplicações baseadas na API de Sockets, contudo, mantendo a lógica da aplicação.
- iv) Desconexão sem perda de dados: assim como ROCKS, um *buffer* no espaço de usuário é utilizado para copiar os dados enviados sobre o socket, permitindo recuperação e retransmissão de eventuais dados perdidos durante a mobilidade do nó.

A Figura 2.10 ilustra o modelo de comunicação em TIPS. Um servidor de registro é necessário para que os pares se registrem e consultem registros de localização. Os registros são documentos XML simples que possuem dois campos principais: *host_name*, que contém o nome do nó; e *current_location*, que é a sua localização (endereço IP) corrente. Os registros são inseridos na DHT sob o identificador de nó. Há uma apresentação inicial, conduzida em duas vias, entre cliente e servidor, onde ambos trocam seus identificadores e sequência de envio e recebimento.

Embora fosse capaz de gerenciar a mobilidade para ambos os pares cliente e servidor, TIPS possuía algumas limitações, como:

Figura 2.10: Comunicação entre Cliente e Servidor com TIPS [28].

- a) o tratamento reativo e sob demanda, somente sendo realizado após *handovers*, quando a aplicação chamava uma função *wrapper* de envio ou recebimento;
- b) múltiplas conexões da aplicação, por exemplo, de um servidor multi-cliente, não eram suportadas;
- c) a implantação levemente intrusiva, havendo necessidade de recompilação da aplicação;
- d) operável somente em redes IPv4;
- e) ausência de autenticação entre os pares, sendo susceptível a ataques *man-in-the-middle* e sequestro de sessão durante as reconexões;

A partir da experiência obtida no desenvolvimento de TIPS, foi então proposto o mecanismo de Sessão Tolerante a Ruptura descrito nesta tese.

Resultados Preliminares

Este Capítulo apresenta os resultados preliminares obtidos, os quais levaram a propor o mecanismo de Sessão. Esses resultados são discutidos em dois contextos: o de uma análise de desempenho entre TIPS [32] e o protocolo MIPv6 [24] a partir de experimentos iniciais; e o de uma análise de requisitos para gerenciadores de mobilidade fim-a-fim.

3.1 Experimentos iniciais

A partir de [28], evoluções na implementação de TIPS foram feitas para: 1) eliminar a necessidade de recompilação da aplicação alvo, tornando TIPS uma camada isolada e transparente entre a camada de Aplicação e Transporte; 2) e operar em *dual stack* IPv4 e IPv6. Essas evoluções e as técnicas utilizadas para o tratamento de mobilidade no nível da aplicação foram descritas em [32].

O objetivo deste experimento foi comparar o desempenho de soluções implementadas em camadas superiores e inferiores: TIPS, uma solução de mobilidade fim-a-fim implementada no nível da Aplicação; e MIPv6, considerada a solução de mobilidade geral da camada IP. Para ambas as soluções, *handovers* de nível 3 foram analisados em um *testbed* experimental implementado em uma topologia de rede IPv6.

No experimento, o nó móvel MN foi preparado para ser movido entre as redes sem fio do *testbed IPv6*, ilustrado na Figura 3.1 e descrito em maiores detalhes na seção seguinte. Para tanto, um *script* foi executado no nó MN para forçar a sua mudança de associação com o APs (*Access Points*), evitando a mobilidade física. O nó MN era associado temporariamente em um

AP e, em seguida, associado a outro, realizando saltos periódicos entre as redes na sequência D, C e A. Uma conexão TCP foi estabelecida entre o cliente no MN e o servidor fixo no nó fixo CN, de modo que os *handovers* de MN ocorriam em meio à conexão TCP estabelecida. Tendo em vista que TIPS é uma solução de camada superior, ela somente foi implantada nos nós-fim MN e CN. Além disso, como o servidor residiu em um nó fixo, logo, registros de localização não foram necessários.

Para analisar o tempo que cada solução de mobilidade levou para restabelecer a transmissão entre os nós após os saltos de MN, foi utilizada a ferramenta *Iperf*¹. Além de medir o tempo de desconexão, essa ferramenta também foi utilizada para gerar *streams* de dados em mensagens de 1024 KB a cada 1 segundo no sentido do nó MN ao nó CN sobre a conexão TCP entre eles. Foi observada uma amostra de aproximadamente 100 *handovers* para cada solução de mobilidade.

3.1.1 Configuração do *testbed*

A Figura 3.1 ilustra a topologia do *testbed*. Para prover Gerenciamento de Mobilidade IP foi utilizado o protocolo MIPv6² e implantada a sua infraestrutura: nó correspondente CN e o agente da rede de origem HA, dois roteadores IPv6 (AR1 e AR2) e um nó móvel MN.

Figura 3.1: Topologia do *Testbed*[32].

Para possibilitar otimização de rotas, o protocolo MIPv6 foi implantado e executado nos nós MN, HA e CN. Nesse caso, o protocolo MIPv6 requer que o nó MN faça registro de localização também com CN [24]. Ao enviar um datagrama IPv6 ao MN, o nó CN verifica se possui em *cache* o atual endereço IPv6 de MN, o endereço CoA. Se o CoA for encontrado, datagramas podem ser roteados ao nó MN sem passar por sua Rede de Origem, evitando o roteamento triangular introduzido na versão MIPv4 [46].

¹<http://iperf.sourceforge.net/>

²<http://www.mobile-ipv6.org/>

O enlace sem fio é provido por três Pontos de Acesso (AP1, AP2, e AP3) compatíveis com o padrão IEEE 802.11 b/g/n. Eles são conectados via cabo IEEE 802.3u (*Fast Ethernet*) e segmentados nas redes lógicas A, C e D, respectivamente. O roteamento das redes D e C é conduzido pelo roteador AR1, enquanto o roteamento da rede A é feito por AR2. Ambos os roteadores possuem rotas para todas as redes na topologia, de modo que o nó MN é capaz de atingir todos os outros nós do *testbed*. Ambos os roteadores propagam em difusão mensagens do tipo *ICMPv6 Router Advertisement* (RA) em intervalos de 30 a 70 milissegundos.

Como descrito na Seção 2.1.2, RAs são utilizados no mecanismo de autoconfiguração de endereçamento *stateless* do nó móvel. Tão logo o nó MN é movido para uma rede visitada, ele recebe RAs do roteador dessa rede. Baseado no prefixo IPv6 da rede contido na mensagem RA recebida e no endereço MAC do MN, o próprio nó MN então gera seu endereço CoA IPv6. O nó MN valida o endereço gerado utilizando os protocolos NDP (*Neighbor Discovery Protocol* [44]) e *IPv6 Stateless Address Autoconfiguration* [65]. Mensagens NS (*Neighbor Solicitation*) são enviadas em difusão pelo nó MN, o qual espera por uma mensagem de reconhecimento NA (*Neighbor Acknowledgement*) durante um tempo predefinido. Se o MN não receber nenhuma NA dentro do tempo esperado, assume-se que seu CoA é único e ele pode utilizá-lo como endereço válido na rede visitada.

3.1.2 Latência de *handovers*

Ambas as soluções TIPS e MIPv6 utilizaram a facilidade provida pelo mecanismo de autoconfiguração de endereçamento *stateless*. Portanto, o tempo utilizado na geração e validação dos respectivos CoAs, referenciado aqui como t_{CoA} , é acrescido à latência de *handover*.

No caso de TIPS, a latência de *handover* reflete o impacto na camada de Aplicação causado por *handovers* no nível da camada de Rede. De acordo com a implementação de TIPS utilizada, a latência de *handover* é expressa por:

$$L_{TIPS} = t_{CoA} + t_{det} + t_{con} + t_{rec}, \quad (3.1)$$

onde: t_{det} é o tempo de detecção de quebra de conexão, que nesse caso foi reativa; t_{con} é o tempo de estabelecer uma nova conexão TCP entre o cliente em MN e o servidor fixo em CN; t_{rec} é o tempo de retransmissão dos dados perdidos no *handover*.

A latência de *handover* do protocolo MIPv6 é expressa por:

$$L_{MIPv6} = t_{CoA} + t_{BU} + t_{BA} + t_{L4}, \quad (3.2)$$

onde: t_{BU} (otimização de rotas) é o tempo do nó MN propagar as mensagens de *Binding Update* (BU) para o agente HA e para o nó CN; t_{BA} é o tempo de espera pelos respectivos *Binding Acknowledgments* (BA) enviados em resposta aos BUs pelos nós HA e CN; e t_{L4} é o

tempo da camada de Transporte (nesse caso, o TCP) reenviar eventuais segmentos que foram perdidos durante o *handover*.

A Tabela 3.1 apresenta os resultados sumarizados para as latências observadas de cada solução. Os tempos em L_{TIPS} são menores que os apresentados em L_{MIPv6} . Considerando que quanto menor a latência, menor é o tempo de desconexão entre os pares durante a mobilidade, isso indica que no experimento realizado TIPS apresentou melhor desempenho no tratamento da mobilidade que o protocolo MIPv6.

Tabela 3.1: Resultados sumarizados das latências de *handover* (em segundos) de TIPS e MIPv6 sobre conexões TCP [32].

Tempos	Mín.	Máx.	DP	Média	Mediana	LI	LS
L_{TIPS}	1.0	3.5	0.35	1.09	1.0	0.76	2.13
L_{MIPv6}	1.0	29.5	7.51	10.07	13.5	0.62	30.29

Legenda: Desvio Padrão (DP), Limite Inferior (LI), Limite Superior (LS).

Os tempos menores são devidos à estratégia adotada em TIPS para ambos os nós MN e CN, a qual foi favorecida pelo próprio comportamento da aplicação utilizada nos testes. Como o tratamento é reativo, a quebra de conexão causada por *handovers* de MN é detectada na camada de Aplicação de ambos MN e CN com o erro gerado quando a aplicação tenta utilizar o descritor de socket para leitura ou escrita. O descritor, contudo, se torna corrompido tão logo o nó MN autoconfigurar um novo endereço IPv6 na nova rede visitada. Como o cliente em MN gera fluxos contínuos TCP para o servidor em CN, os descritores de sockets são utilizados incessantemente (para escrita no cliente e leitura no servidor) permitindo rápida detecção do erro. Quando o servidor detecta que houve quebra de conexão, ele imediatamente espera por uma nova conexão do cliente móvel. O cliente, por sua vez, após detectar a quebra de conexão e, a partir de novo descritor de socket, tenta estabelecer uma nova conexão TCP com o servidor.

Diferentemente do protocolo MIPv6, atualizações de localização mal sucedidas não influenciam a latência de *handover* de nós móveis em TIPS. Especialmente no caso da mobilidade de nós clientes, a atualização de localização é uma operação desnecessária. Evitando operações de registro e busca de localização e provendo rápida sincronização entre cliente móvel e o servidor fixo, foi possível obter baixas latências de *handovers*. No experimento realizado, essa estratégia funcionou bem para o envio de *streams* de dados em rajadas de 1024 KB do nó móvel MN para o nó fixo CN.

Os histogramas dos tempos observados são apresentados nas Figuras 3.2(a) e 3.2(b). A maior parte dos tempos em L_{TIPS} está concentrada em 1.0 s (mais de 80 ocorrências), e uma pequena parte entre os tempos 1.5, 2.0, 2.5 e 3.5 segundos. Por outro lado, em L_{MIPv6} os tempos estão concentrados nos intervalos (2.0, 4.0] e (12.0, 14.0], com aproximadamente 26 e 30 ocorrências, respectivamente. De acordo com a amostra para L_{MIPv6} , observa-se a distribuição

dos tempos com mediana de 13.5 s e latência média de 10.07 s. Os limites inferior e superior do intervalo de confiança foram de 0.62 s e 30.29 s, respectivamente.

Figura 3.2: Histograma dos tempos de: (a) L_{TIPS} e (b) L_{MIPv6} [32].

Os tempos no intervalo $(0.0, 5.0]$ em L_{MIPv6} são devidos aos *handovers* no qual o nó MN foi movido para sua Rede de Origem HN (nesse caso, a rede A, onde está localizado o agente HA). *Handovers* com destino à rede *home* não requerem ser completados com mensagens de BA, pois o nó móvel reutiliza o endereço recebido quando foi iniciado na sua rede de origem. Por outro lado, longos tempos em L_{MIPv6} são devidos às tentativas mal sucedidas de registro de localização no agente HA e no CN.

Uma vez que o protocolo MIP é implementado na camada de Rede, o protocolo TCP considera *handovers* como se fossem um problema de congestionamento na rede. Com as tentativas mal sucedidas de registro de localização de MN, os segmentos de confirmação do TCP receptor em CN são encaminhados para a antiga localização de MN. Dependendo do fluxo TCP e da latência do registro de localização de MN com os nós CN e HA, o tempo t_{L4} pode impactar a latência final de *handover* L .

Além disso, a posição do nó MN nas redes significa a passagem por 1 ou 2 roteadores para atingir CN. Isso pôde ter influência nos resultados do MIPv6, pois as mensagens de *binding update* do nó MN com CN passaram por rotas distintas, com um salto ou dois saltos até o destino. A latência da operação de registro, contudo, não foi observada.

3.1.3 Análise de desempenho

A Figura 3.3 apresenta um gráfico que compara a eficiência de ambos os mecanismos TIPS (sem sessão) e MIPv6. Foi considerada uma latência de $N = 500$ milissegundos como referên-

cia para uma latência ideal de *handover*. Dada a complexidade do Gerenciamento de Mobilidade e sua otimização, meio segundo de *handovers* seria uma meta, cujo tempo de desconexão seria empiricamente transparente ao usuário.

O desempenho é expresso por [32]:

$$P = N/L, \quad (3.3)$$

onde L é a amostra dos tempos observados ordenada ascendentemente.

Assume-se então que um desempenho desejado seria obtido quando $L \leq N$ e $P \geq 1$. De acordo com P , o gráfico da Figura 3.3 mostra com as curvas a tendência para a melhor latência observada.

Figura 3.3: Análise de Desempenho [32]. O desempenho é dado por $P = N/L$, onde N é a latência esperada perto de um valor ideal (assume-se $N = 500$ milissegundos), e L são as latências ordenadas ascendentemente.

Com a amostra dos tempos observados, o mecanismo TIPS sem sessão foi capaz de prover melhor experiência para o usuário. Aproximadamente 90% dos tempos em L_{TIPS} possuem maior p , i.e $L = 1.0$, enquanto o desempenho do MIPv6 está inferior a 2.0 em 70% dos tempos em L_{MIPv6} . Para atingir a latência ideal de 500 milissegundos, vale ressaltar que ambos os mecanismos precisam dobrar o seu melhor desempenho ($P = 0.5$)

3.1.4 Motivação para um gerenciamento otimizado

Assumindo como motivação obter o desempenho ideal $P \geq 1$ para $N = 500$, esforços foram concentrados na otimização do mecanismo TIPS a partir dos resultados apresentados no trabalho em [32]. Partiu-se da premissa de que informações contextuais podem ser utilizadas para implementar mecanismos proativos de detecção de *handover* para antecipar o tratamento

da mobilidade e, conseqüentemente, minimizar o tempo de desconexão do nó móvel durante a mobilidade. C

om informações precisas sobre o *status* do *link* na camada L2, melhores *handovers* podem ser realizados. Nesse sentido, o padrão IEEE 802.21 [2] mostrou-se uma iniciativa promissora. Os desdobramentos dessa iniciativa então resultaram no mecanismo de Sessões proposto.

3.2 Definição dos requisitos para o gerenciamento de mobilidade fim-a-fim

De acordo com as principais soluções existentes de mobilidade fim-a-fim, um mecanismo para gerenciar a mobilidade em camadas superiores requer requisitos, como:

- **Identificadores:** a comunicação fim-a-fim deve ser orientada a identificadores de *host* ou de sessões. Quando o nó móvel for movido para uma nova rede somente sua localização (endereço IP) é alterada. Os identificadores nesse caso podem ser utilizados como referência para retomar conexões quebradas de uma sessão em andamento. A identificação desassociada de endereços IP foi proposta inicialmente no trabalho descrito em [42] e a ideia vem sendo utilizada em outras abordagens, como [56], [69], [28], [8] e [16].
- **Tipo de Mobilidade:** os cenários de comunicação móvel mais comuns na Internet são os de *Single Jumps* (SJ), onde o cliente é executado em nó móvel e o servidor em um nó fixo acessível. O cenário mais complexo é aquele em quem ambos nós são móveis, chamado de *Double Jumps* (DJ). Esse cenário de mobilidade requer gerenciamento de localização, uma vez que os clientes devem saber a atual localização de servidores móveis para retomar transmissões perdidas. Para efetiva comunicação fim-a-fim quando ambos nós são endereçados em redes privadas na borda da Internet, há a necessidade também de se prover mecanismos para travessia NAT.
- **Gerenciamento de Localização:** registros de localização são necessários principalmente quando nós servidores são movidos entre diferentes redes de acesso. Agentes de mobilidade na camada de Rede (*Home Agent* (HA) do protocolo MIP [46] [24]), servidores *redesvous* na camada de Transporte (servidor RVS do protocolo HIP [42]), atualizações dinâmicas de DNS (DDNS [67]) e servidores de registro (*registrar server* do protocolo SIP [54]), têm sido empregados no Gerenciamento de Localização. Dessa forma, quando um nó é deslocado para uma nova rede ele então registra a sua atual localização em uma entidade de registro, permitindo que outros nós o consultem para restabelecer a comunicação em sua nova localização.
- **Tolerantes a atrasos e rupturas:** se uma conexão for quebrada, uma nova deve ser estabelecida para preservar a sessão ativa entre as aplicações executadas em nós móveis. Ses-

sões de comunicações resilientes e tolerantes a desconexões estão implícitas em soluções como [56], [69] e [40]. Entretanto, atrasos também devem ser identificados e tratados por gerenciadores de mobilidade. Por exemplo, se as conexões TCP de uma aplicação forem configuradas com temporizadores do *keepalive* com valores baixos, gerenciadores podem não conseguir tratar a mobilidade na demanda aplicação. Nesse caso, o estouro de temporizadores prematuros leva a quebra da lógica da aplicação no par remoto em meio ao tratamento de mobilidade no nó móvel.

- **Baixa sobrecarga de Sinalização:** sinalização contínua e canais de controle (*out-band*) para coordenar a mobilidade do nó são abordagens eficazes, porém, pouco eficiente no uso dos recursos do dispositivo, que são limitados. O consumo da energia da bateria aumenta com o envio de dados de controle durante a sinalização, se mal dimensionada. Nesse caso, a quantidade de dados de controle deve ser minimizada de modo que os recursos do dispositivo possam ser melhor utilizados na transmissão efetiva de mensagens das aplicações. Sinalização *in-band* [34], por exemplo, utiliza a mesma conexão de dados da aplicação para enviar dados de controle ao par remoto.
- **Confiabilidade:** uma vez que segmentos transmitidos durante *handovers* são perdidos, o nó móvel de alguma forma deve recuperar o *status* da comunicação fim-a-fim, retransmitindo esses segmentos perdidos. Um *buffer* redundante pode ser utilizado para armazenar temporariamente as mensagens enviadas pela aplicação. Após as rupturas de conexões, segmentos perdidos podem ser recuperados a partir desse *buffer* auxiliar [28]. *Buffers* redundantes, chamados *In-flight buffers*, foram propostos no trabalho descrito em [69] como forma de prover confiabilidade na transmissão de mensagens de aplicações em nós móveis.
- **Segurança:** falsos servidores e clientes oportunistas devem ser identificados e ter suas tentativas de reconexões negadas. Mecanismos de autenticação baseados nos identificadores de *hosts* combinados com assinaturas e certificados digitais, por exemplo, poderiam prevenir ataques, provendo à aplicação a capacidade de rejeitar reconexões de nós desconhecidos. Segurança é um requisito presente nos trabalhos descritos em [42] e [34].
- **Implantação:** soluções que demandam maiores custos durante as fases de implantação e manutenção são aquelas dependentes de infraestrutura ou agentes de mobilidade [46] [24] e/ou aquelas implementadas em espaço de *kernel* do nó, o que implicam em adaptações de código do SO e sua recompilação ou modificações em protocolos legados da pilha TCP/IP [42]. Tais dependências e implicações já não existem em soluções de camadas superiores implementadas no nível de Sessão ou Aplicação. Essas soluções são componentes de software que podem ser projetados para operar de forma transparente e independentemente do SO e não demandam infraestrutura, pois a mobilidade é tratada no nós-fim.

- **Transparência:** o tratamento da mobilidade deve ser transparente às aplicações. Dessa forma, o custo da implantação de soluções de mobilidade é também minimizado, pois não há necessidade de recompilação de código no nó móvel. Bibliotecas dinâmicas podem ser implementadas para interceptar chamadas de sistemas ao *kernel*, e sobrepô-las com novas primitivas que embutem o tratamento da mobilidade de forma transparente à aplicação [69].
- **Desempenho:** para se prover transições transparente (*seamless*) os protocolos de mobilidade devem ser eficientes, provendo baixa sobrecarga sobre a comunicação e baixa latência de desconexão durante as rupturas. Esse desempenho é uma das principais motivações para o desenvolvimento de soluções de espaço de *kernel*. Por outro lado, soluções desenvolvidas em camadas superiores podem utilizar abordagens *cross-layer* para detecção de mobilidade no sentido de antecipar o tratamento de ruptura e atingir a eficiência desejada.
- **Deteção de Mobilidade:** mecanismos de detecção de ruptura podem utilizar informações da camada de Enlace (*e.g.* condições do enlace sem fio, como aquelas especificadas pelo padrão IEEE 802.21 [2]) para otimizar os precedimento de *handover*, provendo baixas latências de desconexão. Esse mecanismos de detecção pode ser projetados para operar de forma: (U-MN) unilateral, somente no nó móvel; (N) unilateral com notificações de mobilidade ao nó correspondente e/ou ao gerenciador de localização; (B) bilateral, onde ambos nós envolvidos na comunicação são capazes de detectar rupturas causadas pelo próprio nó ou pelo par correspondente; (R) reativa, onde a falha gerada pela mobilidade do nó identifica a ruptura de comunicação; e (A) assíncrono, operando nó como um sistema ou subsistema de monitoramento da comunicação fim-a-fim e/ou da condição do enlace.

Esses requisitos desejados foram utilizados como princípios de projeto para nortear o desenvolvimento do mecanismo de sessão proposto, descrito em detalhes no Capítulo seguinte.

Sessões de Comunicações Tolerantes a Rupturas

Este Capítulo apresenta detalhes de todo o funcionamento do mecanismo de Sessão proposto. A abstração de Sessões de Comunicação é introduzida ressaltando os componentes de uma arquitetura fim-a-fim responsável por prover tratamento de Mobilidade à camada de Aplicação. Os serviços providos nessa arquitetura, bem como os elementos que compõem uma sessão em cada lado da comunicação são discutidos em detalhes.

Tendo em vista que dados da aplicação podem ser perdidos durante rupturas, um mecanismo de *bufferização* no envio é utilizado para prover confiabilidade no nível da Aplicação. Neste Capítulo são detalhados: o funcionamento do envio e recebimento de mensagens da Aplicação; como é quantificada a perda de dados durante as rupturas; como ocorre a cópia em *buffer* e a recuperação dos dados perdidos.

Assumindo que a identificação do nó móvel deve ser desassociada de endereços IP, o Gerenciamento de Localização é um serviço fundamental para Protocolos de Gerenciamento de Mobilidade. As Sessões utilizam uma DHT para armazenar e consultar registros de localização dos nós móveis ao longo da duração de uma sessão. Este Capítulo apresenta detalhes: da interface BambooDHT, a qual é a implementação da DHT utilizada; do formato do registro de localização; do funcionamento das consultas e atualizações de localização; e de como o Gerenciamento de Localização poderia auxiliar a comunicação entre nós ofuscados por roteadores NAT.

Após o evento de ruptura, uma nova conexão TCP é estabelecida para retomar e sincronizar a comunicação entre os pares em uma sessão. Entretanto, os estabelecimentos de novas conexões estão susceptíveis à ataques do tipo *Man-In-The-Middle* durante a (re) abertura de uma sessão

em andamento. Para prover um mecanismo seguro de (re) abertura de sessão, logo após o estabelecimento da nova conexão, os pares executam um *handshake* e uma autenticação mútua baseada em desafio-resposta. Este Capítulo então apresenta detalhes da geração do desafio e computação da resposta para autenticação, das operações necessárias para a abertura de sessões fechadas e reabertura de sessões suspensas.

O mecanismo de sessão é alimentado com informação da camada de Enlace, como as condições do *link* sem fio. Um mecanismo de Gerenciamento de Gatilhos, o qual observa as condições do enlace e, sob a ocorrência de eventos relacionados à mobilidade do nó, dispara gatilhos que fazem com que outros componentes da arquitetura tomem ações. Neste Capítulo são apresentados detalhes de como as sessões são sincronizadas com o auxílio das informações de estado do enlace e da comunicação fim-a-fim, bem como a forma com que ocorre a detecção de mudança de estado de enlace e detecção de par indisponível.

O mecanismo de sessão é implementado através de uma camada de Socket de mobilidade, chamada de *Transparent IP Sockets* em nossos trabalhos anteriores. Por fim, este Capítulo apresenta detalhes pertinentes sobre a implementação, como: vetor de sessões utilizados para prover múltiplas sessões na aplicação, o encapsulamento da mensagem de controle durante o *handshake* e a dependência entre as bibliotecas que implementam os serviços de sessão.

4.1 Abstração de sessões de comunicação

Entre duas partes, ou pares, em uma comunicação fim-a-fim, uma sessão de comunicação prevê serviços de [62]:

- i) *Controle na Transmissão fim-a-fim*, determinando a ordem da transmissão entre as partes;
- ii) *Gerenciamento de Tokens*, evitando que as duas partes realizem a mesma operação crítica, ao mesmo tempo;
- iii) *Sincronização*, permitindo que as partes retomem, através de pontos de verificação, a sessão a partir de onde foi interrompida depois de uma eventual falha durante as transmissões.

As Sessões Tolerantes a Rupturas são baseadas no conceito de *estado de sessão*. A informação de estado é fundamental no Controle da Transmissão fim-a-fim para evitar que a aplicação envie mensagens quando os pares não estão conectados. Esse conceito foi adaptado a partir do diagrama de estados do socket proposto no trabalho descrito em [69].

O comportamento de uma sessão é definido de acordo com um diagrama de três estados, como ilustra a Figura 4.1: CLOSED (*C*), OPEN (*O*), and SUSPENDED (*S*).

A aplicação no nó móvel inicia uma sessão no estado CLOSED. Para abrir a sessão é necessário: *i*) primeiro estabelecer uma conexão TCP com o par remoto; *ii*) executar uma apresentação (*session handshake*) entre os nós; *iii*) sincronizar o *status* da transmissão dos dados

Figura 4.1: Sessões Tolerantes a Rupturas [29].

entre os nós. Quando a sessão atinge o estado OPEN, a aplicação então é liberada para enviar e receber mensagens sobre um descritor de socket conectado.

Um mecanismo de monitoramento assíncrono à execução da aplicação observa as condições do enlace e da sessão. Havendo a detecção de ruptura no enlace, o monitor altera o estado de todas as sessões OPEN para SUSPENDED. Posteriormente, quando o enlace (ou um novo enlace) estiver pronto para o uso, o monitor aciona o mecanismo de reabertura de sessão suspensa retomando de forma consistente o estado OPEN novamente. Após um número de tentativas mal sucedidas de reabertura de uma sessão, o monitor altera o estado da sessão suspensa para CLOSED.

4.1.1 Uma camada de Socket de mobilidade

Diferentemente das principais arquiteturas baseadas em sessões que foram propostas na literatura, como [69] e [57], o mecanismo descrito nesta tese provê um conjunto básico de funções de comunicação estendidas da API de Sockets com serviços transparentes para [29]:

- i) **Localização de nós móveis:** uma DHT é utilizada para recuperar registros de localização armazenados sob um identificador único de nó;
- ii) **Detecção de ruptura e restabelecimento do enlace:** um Monitor observa as condições do enlace e, sob o evento de ruptura ou restabelecimento, notifica as sessões com um subconjunto de eventos especificados pelo padrão IEEE 802.21 [2].
- iii) **Suspensão de sessão:** um Monitor bloqueia as transmissões da aplicação quando o enlace e a sessão não estiverem preparados para uso.
- iv) **(Re) Abertura de sessão automática e segura:** através da combinação de autenticação mútua de desafio-resposta e PKI.
- v) **Sincronização:** sobre uma nova conexão TCP estabelecida na (re) abertura, os nós realizam um *handshake* para trocar informações correntes da sessão e retransmitir dados perdidos na ruptura, os quais podem ser recuperados de forma consistente a partir de um *buffer* auxiliar da sessão que mantém cópias de mensagens enviadas com sucesso.

Esses serviços de sessão são implementados através de componentes em uma arquitetura, como ilustra a Figura 4.2, que reside em ambos cliente e servidor. Essa arquitetura é responsável por prover suporte a mobilidade à Aplicação através de uma camada que opera de forma transparente sobre a interface de Sockets.

Figura 4.2: Arquitetura: Uma camada de Socket de mobilidade.

4.1.2 Elementos de uma Sessão

Uma sessão é uma coleção de elementos que armazenam informações sobre os pares ao longo da duração de uma comunicação fim-a-fim. Sua definição é dada por:

$$S = \{sstate, sevent, smutex, mh, sock, timers, buf, si_L, si_R, cert_L, cert_R, K_{priv_L}, K_{pub_R}\} \quad (4.1)$$

onde:

sstate: estado corrente da sessão, podendo assumir os valores CLOSED, OPEN ou SUSPENDED.

sevent: evento em curso, podendo assumir os valores ON_RECV, ON_SEND, ON_OPEN ou ON_REOPEN.

smutex: semáforo *Posix* utilizado para sincronizar a transmissão da aplicação de acordo com *sstate*.

mh: *Mobility Handler*, monitor responsável por iniciar mecanismos de recuperação da sessão após rupturas;

sock: parâmetros do descritor de socket utilizado na conexão entre os nós.

timers: temporizadores utilizados nos procedimentos de conexão, envio e recebimento de mensagens.

buf: *buffer* que provê redundância das últimas mensagens enviadas pela aplicação para permitir a recuperação de dados perdidos nas rupturas;

si_L, si_R : subconjuntos de Informações da Sessão (*Session Information*) pertinentes à comunicação entre o nó e o par remoto, respectivamente, que são negociados entre eles durante (re) abertura.

K_{pub_L}, K_{pub_R} : chaves públicas RSA do nó e par remoto, respectivamente.

$cert_L, cert_R$: certificados X509 do nó e par remoto, respectivamente.

4.1.3 Informação de Sessão

Os nós participantes em um sessão são individualmente controlados por um par de Informação de Sessão Local (si_L) e Remoto (si_R). Uma Informação de Sessão consiste de uma tupla de 5 elementos:

$$si = \{h_{id}, s_{id}, seq_S, seq_R, f\}, \quad (4.2)$$

onde: h_{id} é o identificador único do nó; s_{id} é o índice da sessão; f são *flags* de controle; e seq_S e seq_R contabilizam a quantidade de bytes enviados e recebidos, respectivamente, ao longo de uma sessão.

Durante a fase (re) abertura de sessão um par conhecerá o outro através de uma apresentação em 4 vias (*4-way session handshake*) em que ambos trocam suas Informações de Sessão Local si_L . Esse procedimento é apresentado em maiores detalhes na Seção 4.4. As Seções seguintes descrevem cada um dos elementos de uma Informação de Sessão.

Identificador Único de Nó

A finalidade do identificador h_{id} é permitir que um nó móvel seja unicamente identificado no amplo domínio da Internet através de um mecanismo independente da localização do nó, *i.e.* desassociado do endereço IP. Com o mesmo propósito, a solução TCP-Migrate [56] utiliza *Tokens* de sessão e o protocolo HIP [42] utiliza identificadores HI (*Host Identity*).

De forma semelhante ao HIP, o identificador h_{id} é computado com *hash* SHA-1 sobre a Chave Pública K_{pub_L} pertencente ao nó:

$$h_{id} = SHA1(K_{pub_L}). \quad (4.3)$$

HIP é compatível com a camada IP ao adaptar o identificador HI ao endereço IP. Os 32 ou 128 bits mais significativos do identificador do nó são utilizados para endereçar o nó com IPv4 ou IPv6, respectivamente. No caso do identificador proposto h_{id} , os 160 bits gerados pelo *hash* SHA-1 são utilizados unicamente para identificar o nó e, assim, separar a identificação da localização na camada de Rede e ser independente de versões do protocolo IP.

Ao basear a identificação do nó em uma Chave Pública única divulgada em um certificado X509 assinado por uma CA (*Certificate Authority*) comum entre os nós, além de permitir autenticação do par remoto com a verificação de assinatura, a criação de identificadores ilegítimos é inibida. Outro aspecto importante é o uso de um espaço de nomes amplo que permite gerar 2^{160} identificadores únicos com a função de espalhamento SHA-1.

Índice de Sessão

Um vetor, denominado `_session[]`, é utilizado para acomodar as sessões em andamento em uma aplicação. O objetivo do Índice de Sessão é permitir o acesso direto a uma dada sessão existente nesse vetor. Nesse contexto, o descritor de socket criado pela aplicação, cujo valor é um inteiro positivo quando válido, se mostrou bastante oportuno para atuar como índice de acesso à sessão no vetor. Os índices permitem manipular as sessões em operações de complexidade constante $O(1)$, evitando buscas pelo vetor nas fases de abertura, transmissão de mensagens ou reabertura. Isso, sobretudo, ajuda a minimizar a sobrecarga das Sessões na transmissão entre os pares.

O valor de um descritor de arquivo alocado pelo Sistema Operacional pode variar. Os descritores podem ser reutilizados se um descritor é fechado e outro arquivo ou socket é aberto. Sem o fechamento de descritores, a alocação ocorreria de forma incremental, iniciando em 0 e limitado¹ em 1023.

Uma vez que os valores dos descritores usados nas pontas de uma conexão variam e podem ser assíncronos, identificar uma sessão em ambos os pares com apenas um índice geraria inconsistência no acesso ao vetor de cada nó na sessão. Dessa forma, o Índice de Sessão s_{id} é definido pela composição dos índices de acesso local e remoto, respectivamente:

$$s_{id} = \{i_L, i_R\}. \quad (4.4)$$

¹Tipicamente em sistemas Linux há um limite de 1024 arquivos abertos para um processo de usuário, porém configurável.

Particularmente o índice remoto i_R permite que um servidor localize uma sessão suspensa quando o cliente solicitar sua reabertura. Na fase de reabertura de sessão suspensa, após uma nova conexão TCP ser estabelecida entre os nós, o cliente enviará ao servidor sua Informação de Sessão Local si_L , a qual será recebida e entendida pelo servidor como a Informação de Sessão Remota si_R . Então, o valor do índice i_R enviado pelo par remoto será o valor correspondente de i_L localmente no nó. Assim, o mapeamento de índices local e remoto em um par será $si_L.s_{id}.i_L = si_R.s_{id}.i_R$.

Da mesma forma, o servidor envia a sua Informação de Sessão Local si_L ao cliente após autenticá-lo. O cliente então irá acessar a sessão suspensa e validar a Informação de Sessão Remota si_R fornecida pelo servidor. Detalhes do procedimento de autenticação são apresentados na Seção 4.4.

Flags de Controle

As *flags* em f consistem de um conjunto de elementos lógicos de controle binário organizados em uma estrutura de dados de 1 byte de comprimento. O conjunto é definido por:

$$f = \{mh, srv, clt\}, \quad (4.5)$$

onde: *mh* (*Mobile Host*) determina se a aplicação é executada em um nó móvel ou não; *srv* (*Server*) determina se o nó na sessão é servidor ou não; e *clt* (*Client*) determina se o nó na sessão é cliente ou não.

As *flags* são utilizadas pelos pares para determinar se é ou não necessário realizar consultas ou registros de localização ao longo da sessão, como ilustra a Figura 4.3. Um registro de localização, por exemplo, somente é conduzido pelo nó se suas *flags* locais em $si_L.mh$ e $si_L.srv$ estiverem habilitadas. Por outro lado, o nó somente irá consultar a localização do par remoto em um HLR (*Home Location Register*) se as *flags* $si_R.mh$ and $si_R.srv$ estiverem habilitadas. Com o uso das *flags* a latência de reabertura de sessões suspensas é minimizada, uma vez que operações de registros e consultas podem ser evitadas.

A *flag mh* é configurada no momento da criação das sessões, a partir de um arquivo de configuração carregado no início da execução da aplicação.

Na reabertura da sessão, a sincronização entre os pares é determinada pelas *flags clt* e *srv*. Cada par saberá qual a função que desempenha na sessão e irá se preparar para a reabertura. O cliente deve ser proativo e tentar se conectar com a localização corrente do servidor. O servidor, por sua vez, deve estar pronto para aceitar as tentativas de reconexão do cliente. Essas duas *flags* são configuradas em tempo de execução, quando primitivas típicas do cliente ou do servidor são chamadas pela aplicação. Por exemplo, a *flag clt* é habilitada no cliente quando a

Figura 4.3: Uso dos *flags* de controle durante a reabertura de uma sessão suspensa.

aplicação chama a função `connect()`. As primitivas `listen()` e `accept()` caracterizam a operação de um servidor e, quando são chamadas pela aplicação, habilitam a *flag srv*.

Considerando as aplicações P2P, que possuem função equivalente, as *flags* específicas poderiam auxiliar o restabelecimento da comunicação entre as aplicações. Nesse caso, os pares poderiam negociar a função de cliente ou servidor para estabelecer a sessão, ou permitir reconexões bidirecionais como forma de otimizar o estabelecimento de uma sessão. O suporte a essas aplicações, contudo, ainda não é provido pelas Sessões.

Sequências de Envio e Recebimento de dados

O *status* de transmissão em uma sessão é controlado em ambos os pares por dois *offsets*: sequência de envio seq_S e sequência de recebimento seq_R . Ambas as sequências, na prática, são variáveis do tipo inteiro sem sinal, de 8 bytes, que contabilizam a quantidade de bytes enviados e recebidos, respectivamente, ao longo de uma sessão. Ao criar a sessão, as sequências são zeradas, se uma mensagem de n_s bytes é enviada com sucesso pela aplicação, então n_s bytes são somados à seq_S . Da mesma forma, se a aplicação recebe n_r bytes sobre a sessão, n_r bytes são somados à seq_R .

Com 8 bytes é possível atribuir valores inteiros sem sinal entre 0 e 16 EB², o que é suficiente para contabilizar os bytes transmitidos em uma sessão durante a execução da grande maioria das aplicações existentes. Com um dispositivo móvel transmitindo a uma taxa constante e fim-a-fim de 10 Mbps, por exemplo, uma sessão levaria mais de 55784 anos para transmitir 16 EB em mensagens ao par correspondente. Dessa forma, o transbordamento (*integer overflow*) de somas cumulativas nas variáveis de sequência é considerado um fato improvável durante um tempo de sessão (humanamente factível) de execução em um dispositivo móvel.

²16 EB (Exabytes) correspondem a 16.384 Petabytes, ou 17.179.869.184 GB.

Conhecer o *status* da transmissão é uma importante informação para os pares determinarem a quantidade de dados perdidos na ruptura. Dessa forma, as sequências *seq_S* e *seq_R* operam como pontos de verificação (*checkpoints*) permitindo aos pares retomarem sessões suspensas de forma confiável. Esses dados perdidos são recuperados a partir do *buffer* da sessão utilizado para armazenar cópias de mensagens enviadas pela aplicação. Detalhes são apresentados na Seção 4.2.

4.2 Confiabilidade no nível da Aplicação por meio de *bufferização* no envio

Uma ruptura, seja ela causada pelo nó ou pelo par remoto, leva à quebra da conexão e corrompe os descritores de socket abertos em ambos lados da comunicação. Com o descritor corrompido, o acesso aos *buffers* do socket no espaço do *kernel* é inviabilizado. Dessa forma, segmentos ainda não confirmados e não enviados que estão armazenados no *buffer* de envio do socket são perdidos.

Para prover confiabilidade na transmissão após rupturas, as mensagens enviadas pela aplicação são salvas e recuperadas a partir de um *buffer* de sessão. Esta Seção apresenta detalhes das operações em *buffer*.

4.2.1 Envio e Recebimento de mensagens na Aplicação

O envio e recebimento de mensagens é, na prática, um procedimento de cópia entre o *buffer* da aplicação e o *buffer* do socket no espaço do *kernel*, como ilustra a Figura 4.4. No envio, os *out_len* bytes da mensagem *buf_out* são escritos no *buffer* de envio. Esses bytes são envelopados pelo TCP em segmentos de comprimento limitado ao MSS, os quais são passados à camada de Rede conforme o Controle de Congestionamento e de Fluxo do TCP. No receptor, o TCP irá desencapsular os segmentos e armazená-los no *buffer* de recebimento do socket. Cabe à aplicação no destino então consumir os *in_len* bytes a partir do *buffer* de recebimento, os quais serão copiados para o *buffer* da aplicação *buf_in*.

As primitivas da API de Sockets utilizadas para transmissão de mensagens, por operação padrão, são bloqueantes. Nas primitivas de envio, como *send()* e *write()*, o bloqueio ocorre enquanto o *buffer* de envio estiver cheio. Nesse caso, se o comprimento da mensagem for maior que o espaço livre no *buffer*, *i.e.* $out_len > SO_SNDBUF - SND.WND$, o bloqueio ocorre enquanto houver bytes remanescentes da mensagem a serem escritos no *buffer* de envio. O *kernel* não irá retornar da função de envio até que o último byte do *buffer* da aplicação seja copiado no *buffer* de envio do socket [60].

Nas primitivas de recebimento, como *recv()* e *read()*, o bloqueio ocorre enquanto o *buffer* de recebimento estiver vazio e não houver dados para a aplicação consumir; nesse caso, *RCV.USER*

Figura 4.4: Buffers de envio e recebimento do Socket.

Tabela 4.1: Notação utilizada na descrição dos *buffers* de envio e recebimento do Socket (adaptações a partir da terminologia apresentada nos trabalhos [49] e [10]).

Símbolo	Descrição
send()	Primitiva de envio de mensagens da API de Sockets.
recv()	Primitiva de recebimento de mensagens da API de Sockets.
sockfd	Descritor de socket utilizado para transmissão em cada aplicação.
buff_out	Buffer da aplicação utilizado no envio de mensagens.
buff_in	Buffer da aplicação utilizado no recebimento de mensagens.
out_len	Comprimento da mensagem armazenada em buff_out.
in_len	Comprimento da mensagem a ser armazenada em buf_in.
flags	Flags que especificam o tipo de transmissão.
ns	Quantidade de bytes escritos no buffer de envio, em caso de sucesso.
nr	Quantidade de bytes lidos do buffer de recebimento, em caso de sucesso.
>>>	Espaço utilizado no buffer.
...	Espaço livre no buffer.
SO_SNDBUF	Comprimento do buffer de envio do socket, que pode ser manipulado pela aplicação.
SO_RCVBUF	Comprimento do buffer de recebimento do socket, que pode ser manipulado pela aplicação.
SND.UNA	Número de sequência de um segmento TCP enviado, mas ainda não confirmado pelo destino.
SND.NXT	Número de sequência do próximo segmento TCP a ser enviado, o qual o receptor está pronto para receber.
SND.WND	Comprimento da Janela de Transmissão. Contém os segmentos em trânsito no intervalo dos não confirmados pelo destino em [SND.UNA, SND.NXT] e dos que estão prontos para o envio em [SND.NXT, SND.UNA + SND.WND].
SND.USER	Segmentos inseridos no buffer de envio, mas que ainda não estão prontos para o envio, onde $SND.USER \leq SO_SNDBUF - SND.WND$.
RCV.NXT	Número de sequência do próximo segmento que o receptor está pronto para receber.
RCV.WND	Comprimento da janela de recebimento que é notificada ao emissor.
RCV.USER	Segmentos recebidos e confirmados pelo destino, mas que ainda não foram consumidos pela aplicação. Se $RCV.WND = 0$, então $RCV.USER = SO_RCVBUF$.

= 0. A primitiva de recebimento retornará com os bytes disponíveis no *buffer* de recebimento,

mesmo se o comprimento da mensagem que a aplicação espera receber for maior que a quantidade de bytes disponíveis para leitura no *buffer*, *i.e.* $in_len > RCV.USER$.

Cabe ressaltar que a quantidade máxima a ser lida do *buffer* de recebimento com uma única operação `recv()` é de 64 KB. Esse limite é devido ao tamanho máximo de `RCV.WND`, que no cabeçalho do TCP pode assumir valores dentro de um espaço de 16 bits de comprimento. Se o emissor enviar mensagens de comprimento superior a 64 KB, serão necessárias leituras sucessivas de blocos das mensagens com `recv()` no receptor para atingir `out_len` bytes enviados pelo emissor.

Estão previstas duas situações de retorno das chamadas: em caso de sucesso ou fracasso na execução. Casos de fracasso, onde o valor de retorno é -1 para n_s ou n_r e ocorrem principalmente quando: parâmetros inválidos são passados para as primitivas; o descritor de socket não está conectado ao par remoto; a conexão é reiniciada pelo par remoto; há estouro de temporizador do TCP sobre a conexão; e recursos (memória) são insuficientes para executar operação. A execução bem sucedida deve retornar um valor não negativo indicando a quantidade de bytes lidos ou escritos nos *buffers* do socket a partir do descritor `sockfd`.

Os descritores de sockets, contudo, podem ser configurados para transmissões não bloqueantes. Nesse caso, o retorno das funções é imediato e varia conforme o estado dos *buffers*, que podem estar cheios no momento da escrita e/ou vazios em leituras. A aplicação então deve tratar retornos de execução mal sucedida das funções em operação não bloqueante.

4.2.2 Perda de dados durante rupturas

O retorno bem sucedido, no entanto, apenas indica à aplicação que a quantidade de bytes da mensagem foi inserida ou consumida a partir dos *buffers* de um socket conectado. No envio, a aplicação assume que a mensagem foi transmitida ao par remoto, quando na realidade os bytes da mensagem é que foram inseridos com sucesso no *buffer* de envio do socket. Uma ruptura causada pelo nó móvel faz com que o emissor falhe durante o envio. O descritor de socket se torna corrompido e o acesso ao *buffer* de envio a partir do espaço de usuário é impossibilitado. Na ausência de tratamento, tentativas sucessivas de utilizar o descritor corrompido leva a falha de colapso (*crash failure*) interrompendo a execução da aplicação.

A operação bloqueante da primitiva de envio força a mensagem a ser escrita por completo pela aplicação. Se o emissor faz escrita de mensagens de comprimento superior à capacidade de seu *buffer* de envio, nesse caso $out_len > SO_SNDBUF$, o processo ficará dormindo enquanto o *kernel* insere partes dessa mensagem no *buffer* de envio, à medida que `SND.WND` avança, até que toda a mensagem tenha sido inserida. Se houver ruptura durante a transmissão, parte dessa mensagem chegará ao *buffer* de recebimento do socket no destino e estará disponível em `RCV.USER`. A aplicação no destino irá consumir os bytes disponíveis em `RCV.USER` e, na leitura seguinte, entrará em bloqueio, pois seu *buffer* de recebimento estará vazio. Além de comprometer a consistência do envio, isso levará o receptor a entrar em situação de impasse ao

permanecer bloqueado aguardando a continuação do recebimento da mensagem de um emissor que se tornou indisponível com a quebra de conexão.

Nesse cenário, o receptor irá detectar a quebra após um período de ociosidade na transmissão que excede o temporizador da conexão (*timed out*), o qual é determinado pelo mecanismo de *keepalive* do TCP. Até ocorrer *timed out*, a conexão é considerada aberta³ pelo receptor. Em sistemas Linux, o protocolo TCP inicia o envio de *probes* de *keepalive* após 7200 segundos de ociosidade na conexão, com intervalos de 75 segundos entre os envios, e somente considera a conexão terminada após 9 *probes* sem resposta do destino [63]. Em outras palavras, o par remoto, que causou a ruptura, somente é considerado indisponível pelo nó após aproximadamente 2 horas e 11 minutos depois do último segmento transmitido com sucesso entre eles. Para uma rápida detecção de ruptura, contudo, os parâmetros do mecanismo de *keepalive* devem ser reduzidos ao mínimo possível pela aplicação [32]. Caso contrário, as primitivas de recebimento podem entrar em situação de impasse no receptor.

No emissor, durante a transmissão pode haver acúmulo no *buffer* de envio dos bytes em trânsito na janela de transmissão SND.WND e dos bytes escritos e não prontos para o envio em SND.USR. Como o receptor pode consumir dados do *buffer* de recebimento enquanto RCV.USR > 0, uma ruptura então poderá levar a perda de bytes:

$$loss = SND.WND + SND.USR. \quad (4.6)$$

O pior caso de *loss* será quando $SND.WND + SND.USR = SO_SNDBUF$, ou seja, se no momento da ruptura, o *buffer* de envio estiver cheio.

Transmissões originadas em dispositivos móveis, as quais estão sujeitas a atrasos e rupturas, são passíveis da perda *loss*, não havendo garantias de que os bytes no *buffer* de envio do emissor chegarão ao *buffer* de recebimento no destino, tampouco a aplicação no destino consumirá as mensagens armazenadas em seu *buffer* de recebimento. Nesse contexto, sem o suporte de um mecanismo de gerenciamento de mobilidade, o princípio de *confiabilidade* na camada de Transporte provido pelo protocolo TCP é falho para comunicação entre processos que são executados em dispositivos móveis.

4.2.3 Bufferização no envio

Com o objetivo de prover confiabilidade no nível da Aplicação, cada sessão existente no vetor `_session[ ]` possui um *buffer* auxiliar (*buf*) capaz de armazenar uma cópia dos últimos *loss* bytes de mensagens submetidas às primitivas de envio. Dessa forma, após a reabertura da sessão, os bytes perdidos durante a ruptura podem ser recuperados de *buff* e retransmitidos, caso houver.

³Esse problema é conhecido na Literatura como *TCP half-open connection*.

O *buffer* da sessão é dimensionado para atender ao pior caso de perda de dados em uma ruptura. Dessa forma, seu comprimento é determinado pelo tamanho máximo dos *buffers* do socket que, por sua vez, pode ser manipulado⁴ individualmente pela aplicação com as opções `SO_RCVBUF` e `SO_SNDBUF` sobre o descritor. As opções do socket podem ser manipuladas pela aplicação através da primitiva `setsockopt()`⁵. Em sistemas Linux, nas versões de *kernel* superiores a 2.4, o tamanho máximo manipulado via `SO_RCVBUF` e `SO_SNDBUF` está limitado a 131071 bytes (~ 128 KB).

Tendo $loss \leq 131071$ bytes, o comprimento do *buffer* da sessão `BUFF_SIZE` então será:

$$BUFF_SIZE \leq 128 \text{ KB.} \quad (4.7)$$

A cópia da mensagem da aplicação no *buffer* da sessão é encapsulada pela primitiva de envio `m_send()`, como indicam as linhas 22 e 24 na Figura 4.5. A primitiva `m_send()` é a implementação confiável e tolerante a ruptura da primitiva original `send()` da API de Sockets, que é chamada via ponteiro `__send` (linha 10). A cópia em *buffer* somente ocorre se (linha 20) o retorno da primitiva de envio for bem sucedido e o evento corrente da sessão não for `ON_RETX`. Esse evento indica retransmissão de dados perdidos sobre uma sessão em fase de reabertura. Nesse caso, a mensagem da aplicação `buff` contém os *loss* bytes perdidos que foram recuperados do *buffer* da sessão e estão sendo retransmitidos ao destino pelo mecanismo que reabre sessões suspensas. Uma vez que os `len` bytes da mensagem ainda persistem no *buffer* da sessão, sua cópia é desnecessária.

Se o total de bytes em `n` for maior que a capacidade do *buffer* da sessão (linha 21), somente os últimos `BUFF_SIZE` bytes da mensagem da aplicação são copiados. Caso contrário, os `n` bytes escritos no *buffer* de envio do socket são integralmente copiados para o *buffer* da sessão. Após a cópia, os `n` bytes então são acrescentados à sequência local de envio `siL.seqs` da sessão.

A cópia somente dos últimos bytes da mensagem é devido à escrita circular para inserção em *buffer*. Se o comprimento da mensagem for maior que a capacidade do *buffer*, partes do início da mensagem serão sobrepostas nos ciclos de escrita. Nesse caso, restará no *buffer* parte

⁴O dimensionamento dos *buffers* do socket pode ser calculado através do produto $bandwidth \times delay$ fim-a-fim [61]. Por exemplo, supondo que o enlace de menor capacidade entre o nó e o par remoto seja de 5 Mbps, e o RTT entre eles seja de 50 milissegundos, o produto entre a vazão e o atraso será de 32768 bytes. Se os *buffers* na conexão entre esses pares forem menores que o produto de 32 KB, a janela de transmissão no emissor não irá se expandir até atingir a capacidade do canal fim-a-fim, o que irá limitar o desempenho em algum momento. Por outro lado, *buffers* de comprimento superior a 32 KB nessa conexão, além de haver desperdício de memória local, não melhorariam o desempenho. Mesmo havendo mais espaço para os pares lerem e escreverem nos *buffers* locais, a taxa de transmissão máxima fim-a-fim é determinada pelo mínimo entre a janela de transmissão na origem e a janela de recepção no destino. A janela de recepção, contudo, é limitada ao tamanho máximo de 64 KB devido ao comprimento de 16 bits do campo `rwnd` do cabeçalho do protocolo TCP, no qual é responsável por transportar o valor da janela atual. Para enlaces de alta velocidade, a janela de recebimento pode ser redimensionada para valores superiores a 64 KB [23]. De todo modo, a janela de transmissão não será maior que o produto entre a vazão e o atraso fim-a-fim ou que a janela de recepção do par remoto.

⁵<http://www.kernel.org/doc/man-pages/online/pages/man2/setsockopt.2.html>

```

1  #include "tips.h"
2  ssize_t m_send(int sid, const void *buf, size_t len, int flags) {
3      ssize_t n;
4      if(!_m_init) m_init(_tips_conf_file);
5  in:
6      mt_waitlinkstate(LINK_UP);
7      mt_waitsessionstate(sid, OPEN);
8      _session[sid]->sevent = ON_SEND;
9      _session[sid]->req_len = len;
10     n = __send(_session[sid]->sock.fd, buf, len, flags);
11     if (n == FAILURE) {
12         if (errno == EWOULDBLOCK) goto in;
13         if (mt_failure(errno)) {
14             mt_suspend(sid);
15             mt_reopen(sid);
16             goto in;
17         }
18         return FAILURE;
19     }
20     if ((n > 0) && (_session[sid]->sevent != ON_RETX)) {
21         if (n > BUFF_SIZE)
22             buff_write(_session[sid]->buff, buf + n - BUFF_SIZE, BUFF_SIZE);
23         else
24             buff_write(_session[sid]->buff, buf, n);
25         _session[sid]->siL.seqS += n;
26     }
27     return n;
28 }

```

Figura 4.5: Envio confiável e tolerante a ruptura na Aplicação através da primitiva `m_send()`.

final da mensagem, que corresponde ao comprimento do buffer. Para evitar ciclos de escrita de sobreposição, nesse caso, grava-se somente o fim de mensagem.

Figura 4.6: Escrita no *buffer* da sessão.

A primitiva `buff_write()` copia no *buffer* da sessão os bytes da mensagem da aplicação a partir de um *offset* de escrita, `offset_w`, como ilustra a Figura 4.6. A operação de escrita é circular, então `offset_w` é atualizado conforme $(\text{offset_w} + n) \bmod \text{BUFF_SIZE}$.

4.2.4 Recuperação e reenvio dos dados perdidos

Na fase de reabertura da sessão suspensão, os pares autenticam um ao outro e trocam suas Informações de Sessão Local si_L (detalhes do procedimento são apresentados na Seção 4.4). Através do valor corrente da sequência de recebimento do par remoto, o par emissor é capaz de determinar *loss* na camada de Aplicação. Do ponto de vista da sessão, os bytes perdidos na ruptura correspondem ao intervalo entre o total de bytes que o emissor assume ter envi-

ado ($si_L.seq_S$) e o total de bytes que o receptor efetivamente recebeu no nível da aplicação ($si_L.seq_R$). No nó móvel a perda então é expressa por:

$$usrloss = si_L.seq_S - si_R.seq_R. \quad (4.8)$$

Se $usrloss > 0$, os bytes perdidos são recuperados e reenviados na fase de reabertura da sessão através da primitiva `mt_recoverylostdata()`, como ilustra a Figura 4.7. Os $usrloss$ bytes são recuperados do *buffer* da sessão através da primitiva `buff_read()` e armazenados em `ldata`, como ilustra a Figura 4.8. Para isso, o *offset* de leitura é posicionado em $si_R.seq_R \bmod \text{BUFF_SIZE}$. O *offset* de escrita `offset_w` não é ajustado, pois ele está posicionado em $si_L.seq_S \bmod \text{BUFF_SIZE}$ a partir do último envio bem sucedido.

```

1  #include "monitor.h"
2  size_t mt_recoverylostdata(int sid) {
3      void *ldata = NULL, *ptr = NULL;
4      size_t usrloss, n, ns=0;
5      n = usrloss = _session[sid]->siL.seqS - _session[sid]->siR.seqR;
6      if (usrloss == 0) return 0;
7      if (!(ldata = calloc(1, usrloss))) return FAILURE;
8      buff_read(_session[sid]->buff, _session[sid]->siR.seqR, ldata);
9      ptr = ldata;
10     _session[sid]->sevent = ON_RETX;
11     while (usrloss > 0) {
12         ptr += ns;
13         if ((ns = m_send(sid, ptr, usrloss, 0)) == FAILURE)
14             return FAILURE;
15         usrloss -= ns;
16     }
17     _session[sid]->sevent = ON_REOPEN;
18     free(ldata);
19     return(n);
20 }

```

Figura 4.7: Recuperação e reenvio dos dados perdidos com a primitiva `mt_recoverylostdata()`.

Figura 4.8: Leitura a partir do *buffer* de Sessão.

Os dados recuperados em `ldata` são retransmitidos ao destino através da própria primitiva `m_send()`, como apresentado na linha 13 da Figura 4.7. Para evitar a inserção dos $usrloss$ bytes novamente no *buffer* da sessão, o evento corrente é alterado para `ON_RETX` (linha 10). A

cópia desses dados comprometeria a consistência da transmissão, pois, além de ainda estarem armazenados no *buffer* no momento da retransmissão, eles seriam contabilizados novamente em *si_L.seqs* na primitiva *m_send()*. Ao indicar o evento *ON_RETX*, as sequências de envio e recebimento se mantêm consistentes, mesmo havendo rupturas durante a própria retransmissão. Nessa situação, bastaria aplicar novamente o mesmo algoritmo para recalcular, recuperar e reenviar os dados perdidos que restaram.

4.3 Gerenciamento de Localização com o uso de DHTs

Pelo fato de os nós móveis estarem sujeitos às mudanças de localização, o endereço IP se torna inapropriado para identificar o nó. Dessa forma, baseado em um identificador único e assumindo o endereço IP como informação de localização e não de identificação, é essencial que clientes conheçam a localização corrente de seus servidores para, então, estabelecer conexão e abrir a sessão. De forma similar aos sistemas de telefonia móvel, HLRs (*Home Location Registers*) são necessários para gerenciar a localização de nós móveis.

Enquanto servidores DNS (*Domain Name System*) proveem resolução de nomes em endereços estáticos de nós fixos na Internet, HLRs permitem serviços de busca e registro de localização de nós móveis. Agentes HA (*Home Agents*) do IP Móvel [46][25] e servidores RVS (*Redezvous Servers*) do protocolo HIP [42], por exemplo, são HLRs em protocolos de Gerenciamento de Mobilidade na Internet.

As sessões propostas utilizam uma DHT de propósito geral para buscar e armazenar registros de localização dos nós participantes em uma sessão. Para tanto, é utilizada a implementação *open source* fornecida pela BambooDHT⁶, a qual provê um armazenamento persistente e escalável, além de uma interface simples para acesso à DHT.

O uso de DHTs para registro de localização de nós móveis foi originalmente proposto no trabalho descrito em [28]. Utilizando um Identificador Único de Nó, h_{id} , como *hash* para chave de busca, a DHT provê uma distribuição balanceada dos registros de localização entre os vários servidores DHT. Recentemente, o uso de DHTs para essa finalidade tem sido explorado por outros protocolos de Gerenciamento de Mobilidade. No trabalho descrito em [3] é descrita uma interface preliminar da utilização da DHT em conjunto com o protocolo HIP [42] para resolução do nome do nó pelo identificador correspondente, e a resolução da identificação para o endereço corrente.

As sessões propostas aprimoram o uso da DHT através da resolução do identificador do nó h_{id} e um registro mais sofisticado que os utilizados anteriormente em [28]. O registro é capaz de reunir, além das informações de localização, como o par IP-porta utilizado pela aplicação e o mapeamento NAT correspondente no roteador mais externo, também um certificado X509. Esse certificado é utilizado na autenticação entre os nós durante o estabelecimento da sessão. A dis-

⁶<http://bamboo-dht.org/>

tribuição de certificados é facilitada ao embuti-los em registros disponíveis para ampla consulta a partir de uma entidade de registro comum e conhecida pelos nós envolvidos na comunicação.

4.3.1 A interface BambooDHT

Qualquer dado de até 1024 bytes de comprimento poder ser armazenado, consultado e removido da DHT em função de uma chave de busca k . A chave k é um *hash* de 160 bits gerado pelo nó requisitante com a função de espalhamento SHA-1. O acesso à DHT é provido por meio de uma interface genérica com operações de *put*, *get* e *remove*, como mostra a Tabela 4.2. Essas operações podem ser executadas pelo nó requisitante através de RPC (*Remote Procedure Calls*) sobre TCP. Além de ser de uso simples e estar disponível na maioria das linguagens de programação, RPCs permitem a comunicação entre os nós distribuídos da DHT por trás de roteadores NATs e *firewalls* [52].

Tabela 4.2: Interface genérica *put/get/remove* provida pela BambooDHT [52].

Procedimento	Funcionalidade
$put(k, v, H(s), t)$	Insere uma entrada (k, v) na tabela por um TTL t , a qual pode ser removida com o segredo s .
$get(k)$ retorna $\{(v, H(s), t)\}$	Obtém todas as entradas armazenadas sob uma chave k .
$remove(k, H(v), s, t)$	Remove uma entrada (k, v) com o segredo s , o qual foi inserida. t deve ser maior que TTL restante a partir da inserção.

Uma operação *put* armazena e identifica unicamente na DHT uma tripla $(k, v, H(s))$, onde: k é o *hash* de 160 bits da chave de busca; v é o valor de até 1024 bytes a ser armazenado; $H(s)$ corresponde ao *hash* SHA-1 de 160 bits gerados sobre um valor secreto s arbitrário de até 40 bytes de comprimento. Múltiplos *puts* podem ser armazenados sob uma mesma chave k . Isso fará com que todas as triplas submetidas sejam armazenadas na DHT e encadeadas conforme a ordem de envio das requisições. O armazenamento de uma tripla expira em um tempo t de TTL. Dessa forma, um *put* com a mesma chave, valor e segredo de um *put* anterior irá prolongar o TTL dessa tripla existente na DHT [52].

Uma operação *get(k)* retorna o valor v , o *hash* do segredo e o TTL restante de todas as entradas associadas a uma chave k . Para facilitar a manipulação de múltiplas entradas sob k , uma interface de iteração provida pela BambooDHT permite que o nó requisitante percorra os valores recebidos.

Para remover uma entrada (k, v) , o nó requisitante deve fornecer à DHT o *hash* do valor a ser removido com $H(v)$ e revelar o segredo s , cujo *hash* é conhecido pela DHT com $H(s)$ desde a inserção da tripla. Uma vez revelado, o segredo s não deve ser reutilizado em subseqüentes inserções [52]. Isso evita que nós requisitantes maliciosos personifiquem requisições de remoção. O TTL t passado como argumento da remoção deve ser maior que o valor do

TTL restante da entrada a ser removida. Caso contrário, algoritmos de replicação da DHT irão recuperar a entrada removida depois que o TTL passado na remoção expirar.

Para atualizar o valor em uma tripla $(k, v, H(s))$ armazenada, o nó requisitante deve remover a entrada correspondente na DHT e realizar um novo *put* com o valor desejado. Essa operação é custosa, principalmente se a aplicação no nó requisitante for *stateless*. Nesse caso, uma atualização requer a combinação das três operações: um *get* para obter o valor a ser removido e então gerar seu *hash* com $H(v)$, necessário na operação de *remove*; o próprio *remove* sobre o valor desejado; e, posteriormente, um *put* para inserir a tripla com o novo valor.

4.3.2 Registro de Localização

A Figura 4.9 ilustra um registro armazenado sob uma chave de busca h_{id} utilizada pelas Sessões. Um registro é um documento XML que possui: o endereço interno (Ai) adquirido na atual rede visitada e a porta alocada pela aplicação (Pi); o endereço externo (Ae) e a porta externa (Pe) obtida do mapeamento NAT do roteador mais à frente do nó móvel, se houver; e um certificado X509 (X509Data) do nó no formato PEM (*Privacy Enhanced Mail*). Para determinar quão recente um registro armazenado é, o registro possui o atributo *ts* que indica o tempo no formato *Unix Timestamp* do momento do armazenamento na DHT.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<TIPSreg ts="1311834743">
  <Ai>192.168.0.10</Ai>
  <Pi>12345</Pi>
  <Ae>198.51.100.10</Ae>
  <Pe>60944</Pe>
  <X509Data>
    -----BEGIN CERTIFICATE-----
    MIIB6jCCAZQCAQIwDQYJKoZIhvcNAQEFBQAwDELMAkGA1UEBhMCQ1IxCzAJBgNV
    BAgTAK1HMRawDgYDVQQHEwdJdGFqdWJhMRIwEAYDVQQKEw1DQSBVbmlmZWkxEzAR
    BgNVBAsTCkNBL0RNQy9JQ0UxHTAbBgNVBAMTFENldGhmaWNhdGUgQXV0aG9yaXR5
    MB4XDTEyMDcyNTE4MTAyN1oXDTEyMDcyMjE4MTAyN1owGyxCzAJBgNVBAYTAkJS
    MQswCQYDVQQLIEwJNRzEQMA4GA1UEBxMHSXRhanViYTEPMA0GA1UEChMGVW5pZmVp
    MRAwDgYDVQQLEwdETUMvSUNFMRUwEwYDVQQDEwxCcnVubyBLaW11cmExIzAhBgkq
    hkiG9w0BCQEWFGtpbXVyYUB1bmlmZWkuZWRLMjYwDQYJKoZIhvcNAQEBBQAD
    SwAwSAJBAMRdhJOGJblPtulylmiu/57KhryIHSdtN+BhKFucRjHyYlW8QNM4yveL
    8qqU7a/uyHG+7T/FDSR5TCNZptIr8TUCAwEAATANBgkqhkiG9w0BAQUFAANBAJsS
    Kp0HkztGn/WSRaWe6edID7zdIO75qo8e+AdviLM2YWEAX5SgzrTBmKHV2u04a+J
    kMOWJHIPORbGJQurFBI=
    -----END CERTIFICATE-----
  </X509Data>
</TIPSreg>
```

Figura 4.9: Registro de Localização [29].

O registro ilustrado na Figura 4.9 possui 975 bytes de comprimento. O certificado embutido é assinado por uma CA (*Certificate Authority*), contém uma Chave Pública RSA de 512 bits e informações do dono do certificado e da CA. Um certificado com uma chave pública maior que 512 bits e/ou informações extensas do dono e da CA assinante poderia inviabilizar o registro na DHT, se ultrapassar o limite de 1024 bytes na operação de *put*.

4.3.3 Consulta e atualização de registros

A Figura 4.10 ilustra a busca e o registro de localização em um cenário onde ambos os nós em uma sessão, cliente e servidor, são executados em nós móveis localizados em redes de acesso na Internet. O servidor insere seu registro na DHT, enquanto o cliente tenta buscar a localização corrente do servidor.

Figura 4.10: Gerenciamento de Localização em redes IPv4 [29].

Os clientes da DHT enviam requisições a um *gateway* de acesso à tabela, chamado DHT GW, que é responsável por processar RPCs de *put* e *get* recebidas. No caso da inserção, ao receber um *put*, o DHT GW extrai as informações de IP e Porta de destino da requisição RPC recebida e as insere entre as *tags* do par externo Ae e Pe. Dessa forma, se tal nó estiver ofuscado por roteadores NAT, os valores do par Ae:Pe irão corresponder ao mapeamento NAT do roteador mais externo a partir da rede do nó requisitante. Caso contrário, se o nó requisitante for migrado para uma rede que lhe atribua um endereço globalmente acessível na Internet, os valores do par externo Ae:Pe serão iguais aos do par interno Ai:Pi na inserção do registro desse nó.

Para manipular o registro no momento que antecede sua inserção na DHT, a implementação original da BambooDHT, escrita na linguagem Java, teve de ser adaptada no nó DHT GW. A adaptação consiste na inclusão de uma Classe simplificada para analisar o documento XML de registro e atualizar dois campos desse documento com informações da comunicação obtidas da própria requisição RPC recebida. A manipulação do documento XML é feita via interface DOM (*Document Object Model*) e é realizada durante o tratamento da RPC *put* pelo

nó DHT GW antes da efetiva inserção distribuída na tabela. Dessa forma, ocorre somente o pré-processamento da requisição, mantendo a forma de inserção na tabela inalterada.

No lado do nó requisitante, as operações de *put* e *get* foram adaptadas em duas primitivas básicas da API das Sessões para inserção e busca de registros, respectivamente: $m_putreg()$, $m_getreg()$. Como ilustra a Figura 4.10, a inserção com a primitiva $m_putreg()$ contempla a tripla $(k, v, H(s))$ com o uso dos parâmetros h_{id} (Identificador Único do Nó Móvel requisitante), reg_s (o registro de localização) e K_{sec} (*hash* sobre uma chave secreta), respectivamente. Para os pares em uma sessão é importante ter na DHT o registro de localização mais recente do nó móvel. Dessa forma, a primitiva $m_putreg()$ opera através de atualização e não realiza inserções cumulativas.

A cada *i*ésima atualização de registro, uma chave secreta K_{sec} é computada pelo nó requisitante da seguinte forma:

$$K_{sec} = SHA1(S_i), \quad (4.9)$$

onde S_i é um segredo computado a partir da concatenação de três valores:

$$S_i = SHA1(\{T_i \parallel rand(T_i) \parallel K_{priv_L}\}), \quad (4.10)$$

onde: T_i é o tempo corrente no formato *Unix Timestamp* do nó requisitante; $rand(T_i)$ resulta em um valor inteiro pseudo-aleatório, cuja semente geradora é inicializada com o próprio *timestamp* T_i ; K_{priv_L} é uma chave privada RSA no formato PEM pertencente ao nó requisitante. Ambas a Chave Privada K_{priv_L} e a chave secreta S_i são também utilizadas em outras situações, como na autenticação mútua entre os nós em uma sessão.

O primeiro registro é automático e ocorre quando a sessão é criada, independentemente se o nó é móvel ou fixo. Os registros possuem TTL de 3600 segundos na DHT. Se o servidor for fixo e se a sessão durar mais que esse TTL, o registro na DHT irá expirar. Por outro lado, após a abertura de sessão fechada, o cliente mantém o certificado do servidor para autenticação futura, não sendo mais necessárias consultas à DHT em reaberturas de sessão. Detalhes são descritos na Seção 4.4.

4.3.4 Detectando a presença de roteadores NAT

Após obterem acesso ao enlace local, nós móveis são endereçados dinamicamente pela rede visitada, geralmente via DHCP (*Dynamic Host Configuration Protocol* [12]). Endereços de um espaço de endereçamento privado⁷ [51] são atribuídos aos nós móveis e, posteriormente, reutilizados pela rede na configuração de novos nós visitantes. A comunicação com nós externos na Internet é realizada através de suporte de tradução de endereços NAT (*Network Address Translation*) [15]. Datagramas originados por nós na rede interna têm seus endereços de origem, que

⁷Três blocos de endereçamento privado IPv4 definidos pela AINA (*Internet Assigned Numbers Authority*) são comumente utilizados para endereçar redes privadas: 10.0.0.0/8, 172.16.0.0/12 e 192.168.0.0/16.

são privados, traduzidos no endereço do roteador NAT, que é público, único e acessível. Pacotes no caminho de retorno passam pela tradução inversa no roteador para atingir nós internos. Para isso, roteadores NAT mantêm em memória uma tabela contendo o mapeamento entre os pares endereço-porta interna e endereço-porta externa.

A tradução NAT, no entanto, pode ser baseada em filtragem de pacotes dependente de endereço e porta (*Address and Port-Dependent Filtering* [9]). Para receber pacotes de um nó externo específico, é necessário que primeiro o nó interno envie pacotes endereçados ao IP e à Porta desse nó externo para que seja criado o mapeamento no roteador NAT. Essa filtragem não é um problema para clientes na rede interna, como Navegadores Web, que se conectam com servidores HTTP externos. A conexão simplesmente parte de um nó interno para um nó externo, cujo endereço IP é público. Por outro lado, para aplicações P2P (*Peer-to-Peer*) e servidores que residem em nós móveis, essa filtragem inviabiliza a comunicação fim-a-fim entre nós escondidos por roteadores NAT. Esse problema é conhecido na literatura como *travessia NAT*.

De forma similar ao STUN (*Session Traversal Utilities for NAT* [53]), as atualizações e consultas de registros de localização na DHT permitem que o nó móvel descubra qual o IP e Porta estão alocados a ele no roteador NAT mais externo. Para descobrir a presença de um roteador NAT à frente, logo após a um *put* de um registro, o próprio nó pode obter o registro recém armazenado com um *get* e comparar o par interno Ai:Pi com o externo Ae:Pe. Ainda, com o disparo de sequências de um *put* seguido de um *get*, o nó móvel pode determinar a variação da porta externa Pe alocada pelo roteador NAT.

A especificação de endereços e portas do nó em parte interna e externa no registro de localização, por si só, não é uma solução para travessia NAT. Entretanto, assim como STUN, tais informações são uma ferramenta útil na implementação de técnicas e mecanismos que permitam a travessia.

A técnica de *Hole Punching* [59], por exemplo, tenta prover comunicação fim-a-fim entre aplicações P2P, mesmo que os nós estejam localizados em redes privadas. Ela consiste na tentativa de um nó criar um “furo” no roteador NAT do par oposto de tal maneira que permita o estabelecimento da conexão direta entre eles. Se dois nós, como A e B ilustrados na Figura 4.11, ambos atrás de roteadores NAT, desejam se comunicar fim-a-fim através de uma conexão TCP, então ambos devem primeiro conhecer o mapeamento NAT um do outro. Nesse momento é bastante conveniente o uso dos registros de localização disponíveis para consulta na DHT.

É factível assumir que A e B possam iniciar uma conexão TCP simultânea, enviando um ao outro segmentos SYN endereçados ao IP e Porta do mapeamento NAT do par oposto. Supondo que A tenha iniciativa e envie primeiro um segmento SYN destinado ao mapeamento NAT_B (evento #1 na figura), A irá fazer com que seu roteador NAT_A crie um mapeamento correspondente para esse envio. Ao receber o segmento SYN não esperado de A, o roteador NAT_B

Figura 4.11: Uso de registros de Localização no suporte à travessia NAT com a técnica *TCP Hole Punching*[59] entre os pares A e B em uma aplicação P2P.

(evento #2 na figura) irá filtrar o segmento descartando-o⁸ ou rejeitando-o⁹ através do envio de um segmento de resposta TCP RST (*TCP Connection Reset*) ou uma mensagem ICMP de destino inalcançável (*ICMP Destination Unreachable*), dependendo da implementação do NAT. Entretanto, quando o segmento SYN de B chegar ao roteador NAT_A (evento #3 na figura), NAT_A permitirá que o segmento passe para a rede interna e atinja o nó A. Isso é possível, pois os endereços de origem e destino do segmento SYN de B correspondem a uma conexão TCP que o roteador NAT_A acredita estar em andamento [59] - devido ao mapeamento criado no momento em que A enviou seu SYN a B (evento #1).

Dependendo da implementação NAT, no entanto, a técnica pode não funcionar adequadamente. O roteador deve permitir o recebimento de um segmento SYN do par oposto sobre o mesmo mapeamento que foi utilizado para enviar o segmento SYN a esse nó. Estimativas indicam que 13.6% dos roteadores NATs não permitem essa abordagem [18]. Por outro lado, grande parte dos roteadores NAT, cerca de 70% [18], operam com filtros independentes de endereço e porta (*Endpoint-Independent Filtering* [9]), *i.e.* utilizam o mesmo mapeamento de pacotes subsequentes enviados por um mesmo IP e porta interna para qualquer IP e porta externa. Nesse caso, o roteador NAT_B não iria enviar mensagens ICMP ou segmentos TCP RST em resposta ao inesperado SYN enviado por A, mas o roteador NAT_A iria permitir a entrada do SYN de B, pois já existiria um mapeamento válido criado quando A enviou seu SYN. Dessa forma, as tentativas de conexão TCP simultânea da técnica de *TCP Hole Punching* funcionariam na grande maioria de roteadores NAT [18][59].

⁸O descarte irá produzir *SYN Timeout* em A.

⁹Ao receber um segmento TCP RST ou uma mensagem ICMP, A irá interromper sua tentativa de conexão com B.

4.4 Handshaking e autenticação mútua de desafio-resposta

Para (re) abrir uma sessão os nós se autenticam mutuamente através de um mecanismo de desafio-resposta, o qual é baseado na RFC 6287 [43], como ilustra a Figura 4.12. Manter a compatibilidade com uma especificação que já está válida previne falhas de segurança, além da necessidade de provar a eficácia do mecanismo de autenticação e negociação de chaves.

A autenticação de sessão consiste de um *handshake* realizado em 4 vias, onde:

- 1: Cliente gera e envia seu desafio Q_c .
- 2: Servidor computa a resposta R_s ao desafio Q_c recebido, gera o seu desafio Q_s e envia ambos ao cliente.
- 3: Cliente verifica a resposta do servidor R_s e, então, computa e envia a sua resposta R_c .
- 4: Servidor verifica a resposta do cliente R_c e, então, finaliza a autenticação com uma mensagem *OK*.

Figura 4.12: Autenticação Mútua de Desafio-Resposta, RFC 6287 [43].

4.4.1 Gerando o Desafio

Um desafio Q é gerado conforme a Figura 4.13. O nó desafiante concatena em um valor V sua Informação de Sessão Local si_L e o tempo corrente T no formato *Unix Timestamp*. Um valor *hash* V_h é gerado sobre V com o algoritmo SHA-1. Com sua chave privada K_{priv_L} o nó desafiante utiliza o Algoritmo de Assinatura RSA para gerar a assinatura *sig* sobre V_h . No nó desafiado, o tempo T atua como um *nonce*¹⁰ embutido na mensagem para prevenir ataques do tipo *replay* disparados por falsos desafiantes na (re) abertura de sessão.

¹⁰Number Once Used.

Figura 4.13: Geração do Desafio Q pelo par desafiante [31].

A assinatura sig do desafiante é verificada pelo nó desafiado como ilustra a Figura 4.14. O *timestamp* T e a Informação de Sessão si_L são desencapsulados e concatenados no valor V pelo desafiado, o qual é resumido em um *hash* V_h novamente. O *hash* V_h e a assinatura sig são verificados com a Chave Pública K_{pub_R} do desafiante através do Algoritmo de Verificação RSA. Em caso de sucesso, o par desafiado computa sua resposta R ao desafio Q recebido. Caso contrário, a (re) abertura de sessão é interrompida.

Figura 4.14: Verificação da assinatura do desafiante pelo desafiado [31].

Com a função $m_getreg()$, o desafiado utiliza o Identificador de Nó h_{id} (a partir da Informação de Sessão si_L embutida no desafio Q recebido) para obter o registro de localização do desafiante na DHT. A partir do certificado X509 contido no registro do desafiante, o desafiado extrai a Chave Pública K_{pub_R} para verificar a assinatura em Q . Dessa forma, ambos os nós devem se registrar na DHT logo ao inicializarem as sessões localmente.

Os certificados podem ser assinados por uma CA comum entre os pares. Nesse caso, o próprio nó DHT GW poderia atuar como uma CA. Caso a assinatura do certificado não seja verificável com a Chave Pública da CA, a (re) abertura da sessão é também abortada. Assim,

ao inserir o certificado como parte do registro de localização disponível para consulta na DHT, o custo da distribuição dos certificados é reduzido.

4.4.2 Computando a Resposta

Antes de responder a um desafio Q_R recebido do par remoto, o nó também gera localmente seu desafio Q_L da mesma forma, como ilustra a Figura 4.13. A resposta R é um valor V criptografado com a Chave Pública do par remoto desafiante K_{pub_R} , como ilustra a Figura 4.15. V corresponde a concatenação do desafio Q_R do par remoto, com o desafio Q_L gerado pelo nó e com um segredo S compartilhado entre os nós.

Figura 4.15: Computação da Resposta R [31].

O segredo S é um campo opcional gerado pelo cliente e seguramente compartilhado com o servidor. O segredo corresponde ao *hash* da concatenação do tempo T_0 , com um valor aleatório V_r e com a Chave Privada do nó local K_{priv_L} , como ilustra a Figura 4.16. T_0 corresponde ao tempo do primeiro desafio Q_0 computado, que é utilizado como semente randômica para gerar V_r . Após receber a primeira resposta R_c do cliente, o servidor preserva o segredo S recebido para autenticar as reaberturas de sessão posteriores.

Figura 4.16: Geração do Segredo S no cliente [31].

Uma vez que os dados sigilosos são criptografados com a chave pública do destino, a atuação de um impostor no caminho entre os nós verdadeiros em uma sessão é impedida. Além disso, a chave pública do destino é extraída de um certificado assinado por uma CA conhecida entre os nós. Ainda, o fato de utilizar a chave privada do nó na geração do segredo também não compromete a segurança. O segredo se torna irreversível ao ser derivado de uma função *hash*, que é unidirecional.

Como prevê a etapa 2 da Figura 4.12, o servidor envia sua resposta R_s e seu desafio Q_s ao cliente após autenticá-lo. A resposta R_s , contudo, não contém o segredo S , mesmo ele sendo conhecido pelo servidor em futuras reaberturas. O segredo é sempre criado e enviado pelo cliente de modo que a tarefa do servidor é verificar a equidade de um segredo recebido com um segredo conhecido.

A resposta R é verificada como ilustra a Figura 4.17. O nó usa sua Chave Privada K_{priv_L} para descriptografar a resposta R recebida em um valor V' , o qual é comparado com o valor V correto que é conhecido por ele. Se a resposta vier de um par confiável, além de sua assinatura ser válida, V' será igual a V e, então, o processo de (re) abertura da sessão será continuado. Caso contrário, se o nó não conseguir descriptografar R ou V' descriptografado for diferente de V conhecido, a (re) abertura da sessão é abortada.

Figura 4.17: Verificação da Resposta R [31].

Após verificar a resposta R_c (etapa 3 ilustrada na Figura 4.12) o servidor envia uma mensagem de OK para finalizar a autenticação mútua. Essa mensagem é simplesmente um desafio Q_s , gerado conforme os passos da Figura 4.13. Contudo, uma nova assinatura é computada, pois T corresponde ao tempo corrente da geração da mensagem OK . No cliente, a assinatura da mensagem OK é então verificada.

4.4.3 Abertura de Sessões Fechadas

O uso dos serviços de localização e o procedimento de autenticação mútua entre os nós na fase de abertura de sessão fechada são ilustrados na Figura 4.18. Uma sessão é inicializada com o estado CLOSED e, antes de seu estabelecimento com o par remoto, ambos os pares se registram na DHT. Com o registro de localização disponível, o nó pode obter o certificado e extrair a Chave Pública do par remoto, os quais são necessários para autenticação.

Figura 4.18: Abertura de Sessão Fechada [31].

Após o registro, o cliente tenta estabelecer uma conexão TCP com o servidor. Uma vez estabelecida a conexão, o cliente gera o desafio Q_c e envia-o ao servidor. Com o campo $Q_c.si.h_id$, o servidor consulta o registro de localização do cliente a partir do *gateway* DHT GW comum entre os nós. Com o registro do cliente reg_c , o servidor desencapsula o certificado do campo X509Data do registro, armazena-o em $cert_R$, extrai a Chave Pública e a armazena em K_{pub_R} da sessão. Com a chave K_{pub_R} o servidor verifica a assinatura do cliente contida no desafio R_c , como descrito anteriormente. Caso a verificação seja bem sucedida, o servidor gera seu desafio Q_s , computa a resposta R_s e envia ambos para o cliente. Da mesma forma, com base no Identificador do Nó contido no desafio Q_s recebido, o cliente obtém o registro de localização reg_s do servidor, armazena o certificado em $cert_R$, extrai a Chave Pública K_{pub_R} e, então, verifica a assinatura do servidor em Q_s .

Ao passar pelas etapas da autenticação, o nó salva localmente a sessão i_L (onde i_L corresponde ao valor do descritor de socket conectado) em $_{session}[i_L]$. Então, são preservados os elementos de PKI $cert_R$ e K_{pub_R} extraídos dos registros obtidos, bem como a Informação de

Sessão embutida no desafio recebido como Informação de Sessão Remota si_R . O estado da sessão é alterado para OPEN e, com isso, cliente e servidor podem transmitir mensagens da aplicação sobre o descritor de socket conectado em uma sessão aberta entre eles.

As operações de consulta de registro na DHT são custosas e aumentam significativamente a latência da abertura da sessão. Questões de desempenho durante as fases de abertura e reabertura são discutidas no Capítulo 5.

4.4.4 Reabertura de Sessões Suspensas

Conexões TCP são sensíveis a *timeouts*, rupturas no enlace e, consequentemente, desconexões causadas pela mobilidade do nó. A detecção de rupturas é conduzida por um Monitor de Mobilidade no mecanismo de sessão desenvolvido, o qual observa o estado da sessão e as condições do enlace - detalhes do procedimento são apresentados na Seção 4.5. Uma vez que a ruptura é detectada, o gerenciador de mobilidade no nó móvel altera o estado do enlace de LINK_UP para LINK_DOWN, bem como o estado das sessões em andamento de OPEN para SUSPENDED. Posteriormente, quando o estado de enlace estiver em condições apropriadas para o uso no nó móvel, *i.e.* LINK_UP, o gerenciador inicia o mecanismo de reabertura de sessão. Em um nó fixo, cujo estado de enlace não é susceptível à rupturas, o gerenciador de mobilidade inicia a reabertura da sessão suspensa tão logo detectar quebra de conexão.

A Figura 4.19 ilustra um cenário em que ambos cliente e servidor estão em execução em nós móveis. Se o servidor é executado em um nó fixo, o cliente não precisa obter o registro do servidor durante a reabertura. Se não há necessidade de consultas do cliente, então não há necessidade de o registro de localização ser realizado pelo servidor antes da reabertura de sessão. As operações de consulta e registro de localização, ilustradas na área cinza da Figura 4.19, são determinadas de acordo com as *flags* de controle do par remoto, as quais são conhecidas pelo nó em si_R . O cliente, por exemplo, no momento da reabertura está ciente da mobilidade do servidor através do valor da *flag* em $si_R.f.mh$.

Caso o servidor esteja em execução em um nó móvel, sob falha na tentativa de conexão, o cliente pode obter a localização corrente do servidor consultando a DHT. Então, o cliente tenta reconectar ao servidor de acordo com a localização Ae e Pe obtida do registro do servidor. Isso permite que o cliente trate a *condição de disputa*, eventualmente ocorrida quando o servidor não estiver pronto para aceitar novas conexões ou não atualizar a tempo sua localização na DHT. Nesses casos, uma falha de conexão é retornada pela primitiva `m_connect()` e uma nova consulta de registro de localização é realizada pelo cliente.

Após a falha de conexão, o cliente aguarda 1 segundo para consultar a DHT e então tentar uma nova reconexão. A espera de 1 segundo pelo cliente, em um caso de disputa, dá condições para que o servidor atualize o seu registro na DHT. Experimentos apontam que 61% das atualizações na DHT podem ser realizadas em tempos inferiores a 0,5 segundo (Ver Seção 5.1.3). A persistência do cliente em restabelecer a conexão é determinada em arquivo de configuração

Figura 4.19: Reabertura de Sessão Suspensa [31].

das sessões no nó, onde é definido o número máximo de tentativas de conexão. Esse número é arbitrário, podendo o usuário definir o valor.

Se a consulta de localização não for necessária, a fase de reabertura de sessão suspensa é menos custosa que a abertura de sessão fechada. Nesse caso, ambos os pares já possuem os elementos de PKI, os quais foram obtidos de consultas realizadas na fase de abertura.

Durante a autenticação dos nós, os índices são usados para validar a existência de uma dada sessão i no vetor $_session[]$. O índice enviado pelo par remoto i_R corresponde ao índice i_L que o nó utiliza para acessar a sessão, e vice-versa. Se i_R existir, o nó compara o identificador de nó do par remoto (obtido a partir da mensagem de desafio recebida $Q.si.h_{id}$) com o identificador h_{id} conhecido na sessão $_session[Q.si.s_{id}.i_R]$. Além da autenticação, a validação da sessão através da comparação de identificadores é utilizada para dar ou não continuidade na reabertura de sessão.

Com o auxílio das sequências de envio e recebimento, seq_S e seq_R , ambos os pares ficam cientes da quantidade de bytes transmitidos pela camada de Aplicação até o momento de ruptura [32]. Os bytes de dados perdidos durante a ruptura do nó móvel, calculados conforme a

Equação 4.8, são recuperados do *buffer* da sessão e retransmitidos de forma consistente ao par remoto. Finalmente, depois da autenticação mútua e sincronização da transmissão entre cliente e servidor, o Monitor atualiza o estado da sessão para OPEN novamente.

4.5 Gerenciamento de gatilhos de mobilidade

Gatilhos de eventos relacionados às transmissões do dispositivo são uma importante ferramenta para minimizar o Tempo Médio de Reparo (*Mean Time To Repair* - MTTR) de Protocolos de Gerenciamento de Mobilidade e, conseqüentemente, reduzir a latência de *handover*. Nesta Seção é apresentado o mecanismo de Gerenciamento de Gatilhos de Mobilidade utilizado pelas Sessões. Ressalta-se o suporte para suspender e retomar sessões com aspectos técnicos de implementação em sistemas Linux.

O mecanismo de gatilho é auxiliado por dois subsistemas projetados para detectar eventos relacionados à mobilidade do nó: Detecção de Mudança de Estado de Enlace (*Link State Change Detection* - LSCD) e Detecção de Par Indisponível (*Dead Peer Detection* - DPD). O primeiro é responsável por detectar rupturas e o restabelecimento do enlace sem fio e da conectividade do nó móvel de acordo com o subconjunto de eventos especificados pelo padrão IEEE 802.21 [2]. O segundo subsistema estende o mecanismo de *Keepalive* do protocolo TCP para determinar a disponibilidade do par remoto em momentos de rupturas e atrasos ao longo da duração da sessão.

4.5.1 Sincronização baseada em Estado de Enlace e Estado de Sessão

A condição do enlace é uma importante informação de contexto que pode ser usada para otimizar o gerenciamento de mobilidade. Baseada no estado de enlace, uma sessão pode ser *suspensa* e *retomada* quando o enlace estiver apropriado para o uso. Nesse sentido, o subconjunto de eventos LINK_UP e LINK_DOWN especificados pelo padrão IEEE 802.21 [2] é utilizado para indicar mudanças de estados na camada de Enlace.

As Sessões garantem que uma Aplicação envie e receba mensagens somente quando o estado de enlace for LINK_UP e o estado da sessão for OPEN [29]. Para tanto, um mecanismo de monitoramento assíncrono à execução da aplicação observa as condições do enlace do nó móvel e da sessão estabelecida entre os nós.

Ao detectar ruptura, o monitor altera o estado de enlace de LINK_UP para LINK_DOWN e suspende a transmissão sobre todas as sessões OPEN da aplicação. A transmissão na aplicação permanece em bloqueio enquanto a sessão estiver em estado SUSPENDED. Quando o enlace estiver apropriado para o uso, o monitor altera o estado de enlace de LINK_DOWN para LINK_UP e inicia o mecanismo de reabertura de sessão suspensas. Esse mecanismo é

responsável por restabelecer uma nova conexão TCP com o par remoto mantendo a lógica cliente-servidor, executar a autenticação mútua e sincronizar a sessão com o mesmo *status* da transmissão do momento da detecção de ruptura. Com a mudança de estado de SUSPENDED para OPEN, as sessões bloqueadas são liberadas e as transmissões da aplicação são continuadas de forma consistente.

A condição do enlace é indicada por uma variável de estado, *lstate*, que é única e associada a um semáforo *Posix*, *lmutex*, que controla o uso do enlace. Similarmente, para cada sessão em andamento, uma variável *sstate* indica o estado corrente da sessão e um semáforo *Posix*, *smutex*, controla o uso da sessão.

A Figura 4.20 ilustra a execução concorrente de duas sessões, *i* e *j*. *Threads* em operação sobre o vetor de sessões `_session[]` são sincronizadas primeiramente pelo estado de enlace através da função `mt_waitlinkstate()`. Com o evento de ruptura, o monitor mantém o semáforo do enlace bloqueado enquanto o estado for diferente do esperado, que é `LINK_UP`. Isso torna a função `mt_waitlinkstate()` uma barreira, evitando operações de leitura e escrita quando o enlace não tiver condições de uso.

Figura 4.20: Sincronização por Estado de Enlace e Estado de Sessão.

As *threads* bloqueadas pelo estado de enlace são liberadas quando o monitor detecta conectividade, *i.e.* a disponibilidade de uma nova rede sem fio em que o nó móvel se associou e obteve um endereço IP. O monitor então altera o estado para `LINK_UP` e desbloqueia o semáforo do enlace *lmutex*. Em seguida, o monitor inicia o mecanismo de reabertura de sessão suspensa. O semáforo *smutex* da *i*ésima sessão é mantido em bloqueio até que o estado da sessão seja `OPEN` novamente. Nesse caso, o bloqueio do fluxo da aplicação ocorre na função `mt_waitsessionstate()`.

Ambas as funções `mt_waitlinkstate()` e `mt_waitsessionstate()` são inseridas nas primitivas tolerantes a rupturas de envio e recebimento de mensagens, como apresentam as linhas 6 e 7 da Figura 4.5. Dessa forma, o bloqueio com semáforos *Posix* ocorre independentemente da configuração do descritor de socket, mesmo ele operando de forma não bloqueante. Enquanto as *threads* de usuário que invocam as primitivas de envio e recebimento das Sessões permanecem

bloqueadas aguardando a condição de desbloqueio, o monitor observa a atividade do enlace e da sessão no par remoto. Essa tarefa de monitoramento é conduzida pelas funções assíncronas, respectivamente, como ilustra a Figura 4.21: LSCD (*Link State Change Detection*) e DPD (*Dead Peer Detection*).

Figura 4.21: Monitoramento e Sincronização.

4.5.2 Detecção de Mudança de Estado de Enlace - LSCD

Após a ruptura o descritor de socket se torna inutilizado. As primitivas de envio retornam o erro EPIPE¹¹ com a tentativa de envio sobre a conexão quebrada. Em operação padrão, a aplicação é notificada pelo Sistema Operacional com o sinal *Posix* SIGPIPE. Se a aplicação não tratar esse sinal, a tentativa mal sucedida de escrita leva a falha de colapso interrompendo sua execução. A mobilidade nesse caso seria detectada apenas com a tentativa de escrita sobre o descritor de socket corrompido. Para evitar a detecção condicionada à falha em operação de escrita, as sessões utilizam o mecanismo LSCD.

O mecanismo LSCD é implementado através de uma *thread* responsável por detectar a ruptura no enlace de rádio de acordo com eventos de rede disparados pelo *kernel*. Mudanças na tabela de roteamento no nó móvel são traduzidas para o formato padronizado dos eventos LINK_UP e LINK_DOWN definidos pelo padrão IEEE 802.21, como ilustra a Figura 4.22.

Assim como proposto no trabalho descrito em [47], LSCD utiliza *Rtnetlink Sockets*¹² para receber eventos internos do SO que são disparados pelo *kernel*. *Rtnetlink Sockets* é uma API baseada em Sockets disponível em sistemas Linux que permite que processos em nível de usuário se comuniquem com o *kernel* por meio de descritores de sockets ordinários [66]. Dessa

¹¹ Significa que o descritor de socket orientado a conexão foi encerrado inesperadamente pelo nó ou pelo par remoto.

¹²<http://www.kernel.org/doc/man-pages/online/pages/man7/rtnetlink.7.html>

Figura 4.22: Detecção de Mudança de Estado de Enlace no nó móvel.

forma, gatilhos de mobilidade implementados no nível da aplicação são aptos a recuperar informações de rede, como: parâmetros correntes da camada de Enlace, adição e remoção de rotas na tabela de roteamento, endereços IP das interfaces, informações sobre as redes vizinhas, etc.

Descritores de *rtnetlink socket* podem ser configurados para receber notificações do *kernel* sobre um conjunto desejado de eventos, o que evita o envio de requisição ao *kernel* e verificação periódica da condição do enlace. Isso permite que o mecanismo LSCD seja passivo, evitando *espera-ocupada* no monitoramento. O grupo de eventos desejados, chamado *rtnetlink multicast group messages*, para o LSCD é inicializado para que o *kernel* notifique somente eventos relacionados às alterações na tabela de roteamento do nó móvel. Como ilustra a Figura 4.22, LSCD filtra duas mensagens *rtnetlink* em particular, `RTM_DELADDR` e `RTM_NEWADDR`, as quais são disparadas pelo *kernel* ao remover ou adicionar endereços IPs na interface de rede padrão utilizada pela aplicação.

Quando o *kernel* dispara mensagens `RTM_DELADDR`, o LSCD atualiza o estado do enlace para `LINK_DOWN` com a função `mt_set_linkstate()` e imediatamente chama a função `suspend_all_sessions( )` para alterar o estado de todas as sessões `OPEN` para `SUSPENDED`. A mudança de estado gera a condição de bloqueio para as outras *threads* em execução concorrente que enviam e recebem mensagens sobre o vetor `_session[]`.

Ao receber a mensagem `RTM_ADDADDR`, assume-se que o nó móvel está associado e endereçado em nova rede visitada e o enlace está pronto para o uso novamente. Imediatamente o LSCD atualiza o estado do enlace para `LINK_UP` e chama a função `mt_reopen_all_sessions()` para reabrir todas as sessões suspensas. Para liberar *threads* bloqueadas pelo estado de sessão, cada *i*ésima sessão `SUSPENDED` é individualmente reaberta em uma *thread* específica com a função `mt_reopen(i)`.

4.5.3 Detecção de Par Indisponível - DPD

Uma vez que ambos cliente e servidor podem estar hospedados em nós móveis, então ambos podem originar rupturas em uma sessão OPEN entre eles. Se o servidor for o nó responsável por causar a ruptura, ao detectar o evento de mobilidade, o mecanismo LSCD no servidor inicia o processo de reabertura da sessão suspensa, aceitando conexões TCP novamente. No outro lado da comunicação, o cliente deve rapidamente detectar a ruptura causada pela mobilidade do servidor e, então, iniciar o mecanismo de recuperação para restabelecer a sessão suspensa.

Esse tratamento de reabertura da sessão é similar para a ruptura causada pelo cliente. Quando o mecanismo LSCD do cliente inicia o mecanismo de recuperação ao detectar a ruptura originada pelo próprio cliente, o servidor deve detectar a quebra de conexão e estar preparado para aceitar a reconexão do cliente.

Do ponto de vista do emissor e receptor na sessão, após a ruptura, o receptor irá consumir os bytes remanescentes no *buffer* de recebimento do socket. Na leitura seguinte, entrará em bloqueio, pois *buffer* estará vazio. O receptor irá detectar ruptura somente quando o TCP determinar *timeout* na conexão. Até esse evento, a conexão é considerada aberta pelo receptor. Isso leva o receptor a entrar em situação de impasse em leituras bloqueantes, pois estará aguardando mensagens sobre uma conexão quebrada.

Para evitar a condição disputa e impasse na fase de reabertura, um lado da comunicação deve rapidamente detectar a ruptura causada pelo lado oposto. A ausência de tal detecção provoca conexões entreabertas (*half-open*) no receptor. Nesse contexto, o mecanismo DPD é utilizado para determinar a disponibilidade do par remoto e inferir rupturas causadas por ele.

Ao invés de criar um mecanismo de *keepalive*, como IKE-DPD [22], DPD utiliza o próprio mecanismo de *keepalive* do TCP sobre o socket conectado da sessão para confirmar a disponibilidade do par remoto. Para possibilitar rápida detecção, os valores dos parâmetros do *keepalive* da conexão TCP, como TCP_KEEPCNT¹³, TCP_KEEPIIDLE¹⁴ e TCP_KEEPIINTL¹⁵, são reduzidos ao mínimo para cada sessão. Na camada de Aplicação esses parâmetros podem ser manipulados como opções do descritor de socket através da função `setsockopt()`, a qual está disponível na API de Sockets.

Assim como LSCD, o mecanismo DPD funciona de forma assíncrona à execução da aplicação através de uma *thread Posix*. Para cada sessão *i* OPEN, o DPD chama a primitiva `recv()` sobre o descritor de socket conectado para ler 1 byte dessa sessão, como ilustra a Figura 4.23. Como operação padrão, a primitiva `recv()` permanece em bloqueio enquanto o *buffer* de recebimento do socket estiver vazio ou ocorrer determinadas falhas na execução. Para evitar que o DPD fique bloqueado em uma única sessão, o temporizador de leitura do descritor de socket de cada sessão é configurado para permanecer em bloqueio no máximo por 1 segundo. Isso permite

¹³Numero de mensagens *probes* enviadas até a conexão ser considerada quebrada.

¹⁴Tempo ocioso, em segundos, antes do TCP iniciar o envio de *probes* ao par remoto.

¹⁵Intervalo, em segundos, entre o envio de *probes*.

Figura 4.23: Detecção de par indisponível.

que uma única *thread* DPD percorra o vetor `_session[]` a cada segundo e teste a disponibilidade de par remoto de todas as sessões OPEN.

Em sistemas Linux as *syscalls* utilizam uma variável inteira, chamada `errno`, para especificar o evento errôneo em caso de falha na execução da chamada. Após o estouro do temporizador de leitura, a primitiva `recv()` irá retornar com erro e a variável `errno` será configurada com o valor `EWOULDBLOCK` ou `EAGAIN`. Esse erro indica que o descritor está marcado como não bloqueante e não há dados para receber, pois o *buffer* de recebimento do socket está vazio. Se houver dados para leitura, a primitiva irá retornar sem erro, mas não consumirá o byte do *buffer* de recebimento. A *flag* `MSG_PEEK` permite que o byte seja tratado como não lido e permaneça no *buffer* para a próxima chamada de leitura sobre o descritor de socket, a qual será conduzida pela aplicação. Dessa forma, *threads* da aplicação poderão consumir mensagens de forma consistente independentemente do acesso ao *buffer* de recebimento realizado pelo mecanismo DPD.

Quando o *keepalive* do TCP detectar que o par remoto da *i*ésima sessão está indisponível devido ao *timeout* na transmissão, a chamada `recv()` sobre o descritor de socket irá falhar e a variável `errno` terá o valor `ETIMEDOUT`. Com a conexão de uma sessão expirada, o estado dessa sessão é alterado para `SUSPENDED` e o mecanismo de reabertura de sessão é iniciado através das funções `mt_suspend()` e `mt_reopen()`, respectivamente.

4.6 Implementação

Atualmente, dispositivos portáteis estão sendo fabricados para operarem sobre plataformas abertas baseadas em sistemas Linux, como *Maemo*¹⁶, *Android*¹⁷ e *Moblin*¹⁸. Esses sistemas implementam a especificação *Posix*¹⁹ [1] e, com isso, fornecem APIs na Linguagem C com um conjunto extenso de funções para comunicação entre processos, incluindo a API de Sockets

¹⁶<http://maemo.org/>

¹⁷<http://www.android.com/>

¹⁸<http://moblin.org/>

¹⁹IEEE 1003 *Portable Operating System Interface*

que incorpora BSD Sockets [36]. Nesse sentido, para validar e avaliar as sessões propostas, um protocolo de sessão de comunicações foi implementado utilizando a API *Posix* em sistemas Linux.

4.6.1 Incorporando as Sessões no mecanismo TIPS

As Sessões foram incorporadas ao mecanismo TIPS para que uma biblioteca de propósito geral provesse tolerância a rupturas através de um conjunto de funções básicas de comunicação estendidas da API de Sockets. Aproveita-se da experiência em TIPS principalmente a sua interface de funções *wrapper* e transparência. As novas funções são apresentadas na Tabela 4.3.

Tabela 4.3: Funções básicas de comunicação em TIPS.

Primitiva	Descrição
<code>m_socket()</code>	Criação e inicialização de uma sessão com índice i no vetor <code>_session[i]</code> , onde i é o valor do descritor de socket gerado pela função original da API de Sockets.
<code>m_connect()</code>	Estabelecimento de uma conexão TCP e execução em modo cliente da (re) abertura de sessão através do <i>handshake</i> e autenticação mútua em 4-vias.
<code>m_accept()</code>	Aceitação de conexão TCP e execução em modo servidor da (re) abertura de sessão através do <i>handshake</i> e autenticação mútua em 4-vias.
<code>m_write()</code> , <code>m_send()</code>	Envio de mensagens confiável, consistente e tolerante a ruptura.
<code>m_read()</code> , <code>m_recv()</code>	Recebimento de mensagens confiável, consistente e tolerante a ruptura.
<code>m_close()</code>	Fechamento de conexão TCP e encerramento de sessão aberta.
<code>m_getreg()</code>	Consulta de localização de nó a partir de seu identificador h_{id} e obtenção do registro de localização da DHT armazenado sob h_{id} .
<code>m_putreg()</code>	Armazena e/ou atualiza registro de localização na DHT sob o identificador h_{id} .

Nos trabalhos anteriores [28] [27] [30] [32], TIPS provia suporte reativo através do tratamento de erros gerados na aplicação após a mobilidade do dispositivo. As funções agora incorporam a abstração de sessões de comunicação tolerantes a rupturas, além de serem controladas pelo mecanismo de monitoramento assíncrono que observa as condições do enlace e da sessão no nó móvel. Isso permitiu um tratamento da ruptura e recomposição antecipada da comunicação fim-a-fim, o que não era provido em implementações anteriores de TIPS.

Como ilustrado na Figura 4.24, as Sessões podem ser utilizadas de duas maneiras:

- i) **levemente intrusivo**, na qual requer a inclusão da biblioteca `tips.h` no código da aplicação alvo. A biblioteca compartilhada (*Shared Library*) `libtips.so` contém compilação das funções apresentadas na Tabela 4.3. Assim, o programador inclui a biblioteca `tips.h` no código da sua aplicação C/C++ e utiliza as funções da API TIPS que são equivalentes às da API de Sockets. As funções em TIPS são *wrappers* das funções originais da API de Sockets, as quais preservam os mesmos parâmetros das respectivas funções definidas em `socket.h`. Assim, requer apenas a alteração das chamadas das primitivas na aplicação alvo,

uma vez que elas possuem o prefixo “m_”. Mesmo havendo a necessidade de recompilação, a lógica da aplicação é preservada.

- ii) **Transparente**, nesse caso, não há necessidade de modificação do código da aplicação alvo e, portanto, recompilação. A técnica utilizada para tanto consiste em filtrar *syscalls* disparadas pela aplicação, capturar as chamadas da API de Sockets e substituí-las pelas funções *wrapper* equivalentes em TIPS. A implementação das funções *wrappers* desse filtro é compilada na biblioteca dinâmica *libtips.ld.so*.

Figura 4.24: Operação TIPS: (a) *levemente intrusiva*, onde é necessário incluir a biblioteca *tips.h* e utilizar a funções da API; e (b) *Transparente*, no qual a biblioteca dinâmica *libtips.ld.so* é utilizada no filtro de captura de chamadas de sistema da aplicação, as quais são substituídas pelas equivalentes em TIPS.

A biblioteca dinâmica *libtips.ld.so* é carregada e anexada à aplicação alvo em tempo de execução com o carregador *ld.so*²⁰. Ao configurar a variável de ambiente *LD_PRELOAD* para apontar para uma nova biblioteca, o carregador redefine a biblioteca que será carregada antes das outras. Isso permite sobrescrever funções de outras bibliotecas, as quais não são as originais. Uma chamada de sistema disparada pela aplicação alvo então pode ser direcionada pelo *ld.so* primeiro para a biblioteca pré-carregada e posteriormente para as outras. Assim, pode-se criar um filtro capaz de capturar chamadas de funções da API de Sockets disparadas pela aplicação alvo e substituí-las pelas equivalentes na API TIPS a partir da biblioteca *libtips.ld.so*, dinamicamente carregada. Essa operação permite que o mecanismo de Sessão atue como *middleware*, sendo totalmente transparente às camadas adjacentes de Aplicação e Transporte.

A Figura 4.25 ilustra a operação transparente do protocolo. A *syscall* para a função *send()* da API de Sockets, cuja função é o envio confiável de mensagens pela aplicação sobre o descritor de socket *sockfd*, é interceptada e substituída por *m_send()* da API TIPS. O descritor *sockfd* é utilizado como índice local i_L para acesso à sessão no vetor *_session[]*. Após o tratamento da sessão, a mensagem da aplicação é efetivamente enviada pela função original da API de Sockets (apontada em *__send()*) sobre o descritor de socket conectado *sock.fd* associado à sessão i_L .

²⁰Carregador (*linker/loader*) de bibliotecas dinâmicas nativo em sistemas Linux.

Figura 4.25: `libtips.ld.so`: uma biblioteca dinamicamente carregada e anexada à aplicação alvo com *linker/loader* `ld.so`. As *syscalls* da API de Sockets (`socket.h`) são filtradas e substituídas pelas equivalentes na API em TIPS (`tips.h`), como indicam as setas pontilhadas.

4.6.2 Vetor de Sessões

Sockets são identificados e acessados pelo processo de usuário através de descritores de arquivos, os quais assumem valores inteiros positivos quando abertos. A primitiva `socket()` retorna, em caso de sucesso, um inteiro não negativo representando o menor índice do vetor de arquivos abertos que estiver disponível, podendo ter sido usado e liberado por um arquivo já fechado. Considerando a alocação de um descritor de socket por sessão, o próprio descritor, que é único enquanto aberto, então pode ser utilizado também como identificador local para essa sessão. Dessa forma, o uso de um vetor mostra-se uma estrutura de dados adequada para representar múltiplas sessões na aplicação. O próprio descritor de socket aberto, nesse caso, é o índice de acesso direto à sessão desejada evitando buscas no conjunto de sessões manipuladas pela aplicação.

O acesso ao descritor de socket de uma sessão i é dado por `_session[i]->sock.fd`. Após a detecção de ruptura, na etapa prévia à reabertura da sessão suspensa, o Monitor fecha o descritor corrompido em `_session[i]->sock.fd` e cria um novo descritor de socket no lugar. Um novo descritor de socket é necessário, pois o descritor anterior é corrompido com a ruptura de conexão e se torna inutilizável. Dessa forma, a estratégia empregada na reabertura da sessão suspensa é criar um novo descritor, conectá-lo em uma nova conexão TCP ao par remoto, autenticar e sincronizar os nós na sessão.

As sessões são declaradas no código fonte em `session.h` como ilustra a Figura 4.26(a). Os atributos da estrutura `session_t` são descritos como os elementos de sessão na Seção 4.1.2 do texto. O vetor de sessões `_session[]` (linha 14) é alocado dinamicamente em memória no momento de inicialização da biblioteca. Na prática, `_session[]` é um vetor de ponteiros alocado com comprimento `_s_max`. À medida que as sessões são criadas com a primitiva `m_socket()`,

é também alocada memória, conforme o tamanho da estrutura `session_t`, no vetor na posição correspondente ao valor do descritor de socket criado.

Figura 4.26: Vetor de sessões `_session[ ]`: (a) declaração da estrutura em `session.h`; (b) visualização do vetor de sessões com o uso de índices locais (descritores de socket).

O número máximo `_s_max` de sessões manipuláveis segue o valor especificado pelo usuário e/ou programador em arquivo de configuração `tips.conf`²¹. Tipicamente, em sistemas Linux, a quantidade máxima de descritores de arquivos que podem ser abertos pela aplicação é 1024. Considerando a alocação de um descritor de socket por sessão, a quantidade manipulável em `_s_max` seria efetivamente 1024. Esse limite de arquivos abertos, contudo, pode ser alterado através de ferramentas de configurações de parâmetros do *kernel* em tempo de execução do SO, como `sysctl`²² ou `ulimit`²³. Dessa forma, havendo configuração prévia dos limites para manipulação de arquivos, `_s_max` pode assumir valores superiores a 1024 conforme a definição do usuário.

Em operação padrão, os processos de usuário iniciam a execução com os 3 primeiros descritores (0, 1 e 2) abertos para *entrada*, *saída* e *saída de erro* padrão, os quais podem ser acessados na aplicação com o uso das constantes²⁴ `STDIN_FILENO`, `STDOUT_FILENO` e `STDERR_FILENO`, como ilustra a Figura 4.26-b. Assim, o primeiro socket criado pela aplicação seria identificado com o menor descritor disponível, nesse caso 3. A abertura de descritores padrão inutilizam as 3 primeiras posições em `_session[ ]`. Se, por outro lado, a aplicação fechar esses descritores, eles podem ser reutilizados na abertura de sockets e, com isso, permitindo a alocação das três posições iniciais do vetor de sessões.

Uma vez criada a sessão, ela será representada por um único índice durante a execução da aplicação que representa o primeiro descritor de socket aberto para a sessão. Se descritores são fechados ou abertos nas reaberturas, o novo descritor é atualizado no controle das sessões para

²¹Esse arquivo define parâmetros de configuração das sessões que são carregados no momento de inicialização do mecanismo.

²²Edição do arquivo `/etc/sysctl.conf`. atribuindo um novo valor ao parâmetro `fs.file-max`.

²³Permite configurar limites para arquivos, como tamanho e quantidade, que são manipulados com terminal interpretador de comandos `bash/shell`.

²⁴Constantes declaradas em `/usr/include/unistd.h`.

transmissão de dados. Assim, é mantido o índice inalterado e é atualizado o descritor de socket da sessão em caso de alteração.

4.6.3 Encapsulamento da mensagem de controle

O mecanismo de Sessão prevê um único formato de mensagem de controle: a Informação de Sessão. Essa mensagem possui 41 bytes de comprimento, como ilustra a Figura 4.27(b), e somente é transmitida para (re) abrir a sessão sobre uma conexão TCP recém estabelecida com o par remoto. As mensagens da aplicação, contudo, são transmitidas normalmente, sem nenhuma sobrecarga.

Figura 4.27: Informação de Sessão: 41 bytes trocados entre os pares durante a (re) abertura de sessão. (a) Declaração das estruturas em session.h; (b) Visualização do encapsulamento da Informação de Sessão na pilha de protocolos.

A Informação de Sessão é encapsulada em uma estrutura de dados, *si_t*, como apresenta a Figura 4.27-a. Os 41 bytes da estrutura são distribuídos da seguinte forma: 20 bytes de caracteres sem sinal para Identificação de Nó que serão preenchidos com o valor gerado com o algoritmo de dispersão *hash* SHA-1; 4 bytes para o Identificador de Sessão encapsulados na estrutura *sid_t*, sendo 2 bytes para cada índice; cada sequência é representada por um inteiro sem sinal de 8 bytes (*seq_t*); e as *flags* são encapsuladas em uma estrutura de dados (*flags_t*) de 1 byte de comprimento.

4.6.4 As bibliotecas e os serviços providos

Os serviços de sessão são implementados através de um subconjunto de APIs, como ilustra a Figura 4.28, as quais proveem:

1. *Gerenciamento de Sessão* (session.h): estabelecimento de conexões TCP, *handshake* e autenticação durante a (re) abertura de sessões;
2. *Segurança* (pki.h): manipulação de chaves e certificados, além da geração e verificação da criptografia e assinaturas digitais utilizadas pelas sessões durante a autenticação entre os nós.
3. *Gerenciamento de Localização* (location.h, opendht.h): consultas e atualizações de registros de localização na DHT.
4. *Monitoramento* (monitor.h): gerenciamento da execução concorrente dos monitores LSDC e DPD, bem como a interação deles com o mecanismo de Sessão.
5. *Confiabilidade no nível da aplicação* (buffer.h): provê primitivas de acesso exclusivo e *thread safe* para manipulação (criação, leitura, escrita e destruição) do *buffer* da sessão.
6. *Transparência* (tips.h): reimplementação das principais primitivas API de Sockets (descritas na Tabela 4.3) com o suporte das outras bibliotecas.

Figura 4.28: Dependência das bibliotecas do mecanismo de Sessão. Duas bibliotecas compartilhadas são geradas: libtips.so e libtips.ld.so.

Resultados obtidos

Para avaliar a eficiência do mecanismo de sessão, experimentos foram realizados e dois parâmetros foram analisados:

- **Latência:** das operações de consulta e registro de localização na DHT, abertura de sessões fechadas, detecção de ruptura de enlace e reabertura de sessões suspensas.
- **Sobrecarga:** das sessões na vazão fim-a-fim (*goodput*) durante a transmissão de mensagens entre dois nós envolvidos na comunicação.

5.1 Latência de operações

Para avaliar a latência das operações do mecanismo de sessão, experimentos foram realizados em uma rede de teste. Os resultados foram comparados com protocolos legados que utilizam apresentação entre os nós (*handshaking*) para estabelecer comunicação fim-a-fim, como TCP, SSL e SSH.

5.1.1 Cenário e equipamentos utilizados

A rede de teste consiste de uma rede local sem fio formada pelos nós Cliente, Servidor e Roteador. Um quarto nó, DHT GW, em uma rede remota na Internet, é responsável pelo gerenciamento de localização dos nós móveis. Através do roteador, ambos cliente e servidor na rede interna possuem rotas de acesso à Internet para inserir e consultar registros de localização na DHT.

Os equipamentos utilizados são de hardware convencional (*off-the-shelf*): ambos Cliente e Servidor foram executados em plataforma Linux 32 Bits, cuja versão de *Kernel* é 2.6-27-14, processador Intel(R) Core(TM)2 Duo CPU 2.26GHz, 4GB de RAM; roteador *wireless* IEEE 802.11 b/g; Nó DHT GW foi executado em Linux 64 bits e versão de *kernel* 2.6.20-15, 2GB de RAM, processador Intel(R) Core(TM)2 Quad CPU 2.40GHz.

5.1.2 Abertura de Sessão Fechada

O tempo gasto para abrir uma sessão fechada (CLOSED) é dado por:

$$t_o = t_{TCP} + t_{auth}, \quad (5.1)$$

onde: t_{TCP} e t_{auth} são os tempos gastos no estabelecimento de conexão TCP e na autenticação mútua entre os pares, respectivamente.

O tempo de autenticação mútua é dado por:

$$t_{auth} = t_{Q_c} + t_{R_s Q_s} + t_{R_c} + t_{OK}, \quad (5.2)$$

onde os tempos representam cada etapa do procedimento de 4 vias da autenticação mútua.

Tabela 5.1: Sumarização dos tempos (em milissegundos) da abertura de sessões fechadas.

Tempos	Mín.	Q1	Mediana	Média	Q3	Máx.
t_{TCP}	1.443	1.725	1.850	5.8680	2.088	210.70
t_{Q_c}	0.560	0.572	0.578	0.5826	0.586	1.3590
$t_{R_s Q_s}$	130.3	132.8	134.2	160.40	137.4	1055.0
t_{R_c}	0.173	0.184	0.188	0.1896	0.192	0.3130
t_{OK}	5.089	9.069	9.208	9.8750	9.802	130.00
t_{auth}	136.4	142.8	144.3	171.10	148.3	1064.0
t_o	138.4	144.7	146.3	177.00	150.3	1066.0

A Tabela 5.1 apresenta os tempos das Equações e sumarizados em seis parâmetros estatísticos: mínimo (Mín), primeiro quartil (Q1), mediana, média, terceiro quartil (Q3) e máximo. No nó Cliente, em 800 aberturas de sessões fechadas, cada operação foi medida isoladamente com o uso da primitiva `gettimeofday()`, a qual fornece precisão do tempo em microssegundos. Uma carga de trabalho nessa dimensão permitiu minimizar erros amostrais nos parâmetros estatísticos analisados.

Como observado, o tempo $t_{R_s Q_s}$ é dominante, sendo responsável em média por cerca de 90% do tempo total t_o , variando de centenas de milissegundos a 1.055 s. Esse tempo custoso é devido à sincronização e dependência entre os passos da autenticação, pois um processo emissor permanece em bloqueio esperando a resposta do receptor. Nesse caso, para receber a

mensagem R_sQ_s , o cliente espera o servidor obter o registro do cliente a partir da DHT, extrair a chave pública K_{pubR} a partir do certificado X509 desencapsulado do registro do cliente, verificar a assinatura na mensagem Q_c e, então, criptografar sua resposta R_s com a chave pública do cliente. Além da resposta R_s , o servidor deve criar também o desafio Q_s e enviar ambos concatenados em uma mensagem ao cliente. Ao receber R_sQ_s , o cliente também tem que consultar e obter o registro do servidor a partir da DHT para verificar a assinatura recebida em Q_s .

5.1.3 Gerenciamento de Localização

As altas latências em t_o são resultantes das operações de busca necessárias em ambos cliente e servidor durante a autenticação mútua. Com o objetivo de identificar o gargalo de desempenho foram avaliados os serviços de localização providos com o uso da DHT. A Figura 5.2 apresenta as probabilidades cumulativas para operações **Persistentes** (registros e buscas subsequentes são executadas sobre uma única conexão TCP com o nó DHT GW) e **Não-Persistentes** (uma nova conexão TCP com o nó DHT GW é estabelecida para cada operação de busca ou registro) para operações de *put* e *get* de registros de localização. Foram também conduzidos 800 testes para cada operação de registro $m_putreg()$ e de consulta $m_getreg()$.

Figura 5.1: Probabilidade Cumulativa de operações no Gerenciamento de Localização [31]: (a) Persistentes (P); e (b) Não-Persistente (NP).

De acordo com a amostra observada, a probabilidade de um *put* NP ser executado em menos de 0.5 s é cerca de 0.61, de 0.96 para *get* NP e de 0.86 para um *put* P. Há probabilidade 1 de um *get* P, *put* P, *get* NP e *put* NP ser menor 0.12 s, 1.75 s, 1.35 s e 2.97s respectivamente. Ao preservar a conexão, o nó DHT GW mantém o registro requisitado em memória para atender requisições subsequentes, enquanto requisições que chegam em novas conexões forçam o nó DHT GW recuperar o registro requisitado da DHT.

A alta latência na operação de registro, particularmente em requisições NP, é devida às interações necessárias com a DHT. Múltiplos *puts* sob uma mesma chave de busca *key* fazem com que a DHT armazene todos eles [52]. Assim, um *get* pode recuperar todos os registros armazenados sob *key*. Para atualizar um registro de localização é necessário que o nó requisitante obtenha o registro armazenado com um *get* e, com as informações do registro obtido, remova-o com um *remove* para, então, inserir o novo registro com um *put* sob a chave alvo *key*.

Os resultados apontam o registro e consulta de localização como o gargalo de desempenho no mecanismo de Sessão. É importante ressaltar que rupturas causadas pela mobilidade afetam todas as conexões em andamento do nó, incluindo as requisições que partem via RPC/TCP para acesso à DHT. Diferentemente de sessões das aplicações do usuário, as operações de registro e consulta de localização baseadas em *put-get* são um protocolo de requisição e resposta simples e sem estado. Nesse sentido, gerenciar o acesso à DHT com requisições NP é conveniente. Do contrário, o acesso via RCP/TCP também seria tratado com o mecanismo de reabertura de sessão e, com isso, aumentaria a latência na inserção e busca de registros de localização.

5.1.4 Reabertura de Sessão Suspensa

O tempo gasto para reabrir uma sessão suspensa é dado por:

$$t_r = t_d + t_{TCP} + t_{auth}, \quad (5.3)$$

onde t_d é o tempo de detecção de ruptura, t_{TCP} é o tempo para estabelecer uma nova conexão TCP com o par remoto e t_{auth} é o tempo do *handshake* e autenticação em 4-vias expresso na Equação 5.1.2.

Tabela 5.2: Sumarização dos tempos de resposta (em milissegundos) para a reabertura de sessão suspensa.

Tempos	Mín.	Q1	Mediana	Média	Q3	Máx.
t_d	0.014	2.905	3.176	3.3050	3.855	4.984
t_{TCP}	1.461	1.666	1.791	2.9710	2.224	47.05
t_{Q_c}	0.737	0.755	0.762	0.7689	0.778	1.344
$t_{R_s Q_s}$	4.999	6.734	7.056	8.2560	7.804	156.3
t_{R_c}	0.214	0.222	0.224	0.2408	0.276	0.352
t_{OK}	4.948	8.923	9.182	10.810	9.758	295.5
t_{auth}	12.91	18.75	19.60	23.050	21.45	339.7
t_r	15.49	21.94	23.12	26.350	24.90	343.8

Nesse experimento foi considerado um cenário de mobilidade em que o servidor reside em um nó fixo, enquanto o cliente é executado em nó móvel. Assim, inserções e buscas de

registros de localização são evitadas na fase de reabertura. Ambos cliente e servidor já possuem os certificados um do outro, obtidos das consultas realizadas na fase de abertura.

Assim como no experimento anterior, medidas de tempo foram coletadas no nó Cliente durante 800 testes de reabertura de sessões suspensas. A Tabela 5.1.4 apresenta os resultados sumarizados dos tempos expressos nas Equações 5.3 e 5.1.2.

Para causar rupturas em sessões abertas (OPEN) entre o Cliente e Servidor, um *script* periodicamente modificava o endereço IP da interface padrão do nó Cliente. Dessa forma, o tempo t_d representa o tempo gasto da notificação do *kernel* sobre o evento de remoção de endereço IP até o evento de adição de um novo endereço na tabela de roteamento do nó móvel. Como mostra a Tabela , o tempo t_d varia entre 14 μs a 5 ms.

Ao evitar o acesso à DHT na reabertura, os tempos em t_{auth} diminuem significativamente em relação à abertura. A autenticação mútua na reabertura varia entre 13 e 330 ms, enquanto na abertura o procedimento pode atingir 1 segundo. De acordo com a amostra observada o tempo máximo em t_{auth} corresponde a um valor atípico (*outlier*) na amostra afetado pelos tempos t_{RsQs} e t_{OK} .

O objetivo desse experimento foi avaliar somente as operações necessárias na reabertura de sessões e não o tempo de desconexão. Os resultados obtidos são provenientes dos melhores casos: não há necessidade de consultas na DHT; não há aquisição de endereço IP pelo nó móvel; não há condição de disputa entre os nós, de modo que o servidor esteve sempre preparado para receber reconexões do cliente.

Para se avaliar o tempo total de desconexão seria necessário observar outras operações, que foram desconsideradas em t_r . Em um cenário natural de mobilidade, por exemplo, há o impacto da latência de *handover* L2, em que há associação e, eventualmente, autenticação no nível de Enlace, além do tempo da aquisição de endereço IP fornecido pela rede visitada. A aquisição de endereçamento, contudo, não é parte da funcionalidade de mecanismos de Gerenciamento de Mobilidade. Assim como nos protocolos MIPv4 e MIPv6, o mecanismo de Sessão dependeria de suporte apropriado no nó móvel, como DHCP ou auto-configuração *stateless* de endereçamento.

5.1.5 Avaliação do *handshaking*

O desempenho das fases de abertura e reabertura de sessão foi comparado ao desempenho de abordagens de *handshaking* de protocolos bem conhecidos, como TCP, SSL e SSH. Foram conduzidos 800 testes de handshake para cada solução.

A Figura 5.2 mostra as probabilidades cumulativas e uma comparação entre as soluções. Foi comparado o desempenho da abertura da sessão com o *handshake* do protocolo SSH na Figura 5.2(a), uma vez que ambas as soluções têm operações onerosas durante o primeiro *handshake*.

O gargalo de desempenho é a utilização de requisições NP na abertura de sessão, no entanto, a operação é inevitável. É necessário obter o registro do par remoto, a fim de continuar o

(a) Handshake SSH e Abertura de Sessão

(b) Abertura e Reabertura de Sessão

(c) TCP, SSL e Reabertura de Sessão

Figura 5.2: Probabilidade Cumulativa de protocolos de apresentação (*handshake*) [31].

procedimento de autenticação. Por outro lado, a abertura da sessão SSH requer troca de chaves e negociação, além da interação com a autenticação em nível de usuário. Foi medido o tempo gasto na execução do *handshake* do protocolo SSH com o suporte da ferramenta *Expect*¹, que permite a construção de *scripts* para automatizar aplicações que requeiram dados de entrada a partir da entrada padrão, que é via teclado. Isso evitou introduzir manualmente login/senha. Analisando o tempo médio, o desempenho da sessão mostra um ganho de cerca de 92% em relação ao *Handshake* do protocolo SSH. A partir da Figura 5.2(a), observa-se que a abertura da sessão é realizada em até ~ 1 segundo, enquanto o *handshake* do protocolo SSH leva cerca de ~ 3 segundos.

¹<http://www.nist.gov/el/msid/expect.cf>

A segunda análise realizada comparou o desempenho da abertura com a própria reabertura de sessão. Para reabertura, onde o servidor residia em um nó fixo, registros e buscas de localização na DHT foram evitadas. A reabertura é bastante rápida (tempo médio de 0,02787 s) em comparação com a abertura (tempo médio de 0,1771 s). Na Figura 5.2(b) observa-se que mais de 90% da amostra dos tempos de reabertura levam menos de 0.02 s para serem concluídos, enquanto na abertura, o tempo mínimo de 0,1384 s com 0.9 de probabilidade de uma abertura ser menor que 0.2 s.

Foi utilizada a latência do *handshake* do protocolo TCP para balizar o desempenho do *handshake* do protocolo SSL e da reabertura de sessão. A partir da Figura 5.2(c), cerca de 99% de *handshakes* TCP levam menos de 0.01 s; mais de 90% são menores que 0.02 s para *handshakes* do SSL; e mais de 90% são inferiores a 0.03 s para a reabertura de sessão.

Considerando o tratamento das rupturas de conexões com uma camada de Sessão, que proporciona o restabelecimento seguro da comunicação, os tempos gastos nas fases de (re) abertura são razoáveis. Vale ressaltar que nos experimentos iniciais relatados no trabalho descrito em [32], a solução de mobilidade geral da camada IP apresentou tempos de resposta variando de 1 a 30.29 s, com média de 10.07 s. Isso permite assumir que tratar mobilidade no nível das sessões é vantajoso também do ponto de vista de desempenho.

As Sessões permitem a flexibilidade da configuração de temporizadores sobre uma conexão estabelecida. Isso é feito através de redução dos valores dos parâmetros do *Keepalive* do protocolo TCP para permitir rápida detecção em DPD. Tal flexibilidade de adaptação da aplicação móvel não é provida pelo protocolo MIP, o que pode impactar em falhas na aplicação se o tempo de desconexão for grande. Nesse caso, para uma aplicação que reduz os temporizadores de conexão, como nas Sessões, o protocolo MIP poderia ser ineficaz no provimento de mobilidade de acordo com a demanda dessa aplicação. Dessa forma, não se pode afirmar que o protocolo MIP provê tolerância a atrasos e rupturas como nas Sessões.

5.2 Sobrecarga do mecanismo

Para avaliar a sobrecarga do uso do mecanismo de Sessões em um ambiente controlado, livre de ruídos e interferências sobre a vazão das transmissões, foi utilizado o emulador de rede Netkit² [48]. Esse emulador permite reproduzir topologias de redes variadas, onde cada nó é uma máquina virtual *User Mode Linux*³ (UML). As máquinas virtuais são interconectadas com o uso de um *hub* virtual do Netkit que é executado na máquina real hospedeira. Quando em execução, uma máquina virtual é manipulada através de um terminal *bash*.

²<http://wiki.netkit.org>

³<http://user-mode-linux.sourceforge.net/>

5.2.1 Cenário emulado

Um experimento então foi realizado para avaliar a sobrecarga das sessões em uma rede virtualizada, ilustrada na Figura 5.3. Esse experimento consistiu na medição de vazão na camada de Aplicação (*Goodput*) entre a máquina cliente (emissor) e servidor (receptor). Há três redes de acesso representadas pelos Sistemas Autônomos AS1, AS2 e AS3, onde as máquinas dht, cliente e servidor estão localizadas, respectivamente. Essas redes são interconectadas pelos roteadores R1, R2 e R3, os quais executam roteamento BGP entre eles. Essa emulação de redes é possível, pois as máquinas virtuais no Netkit possuem o pacote de software de roteamento Quagga⁴, o qual provê a implementação dos protocolos OSPF, RIP e BGP.

Figura 5.3: Topologia da rede virtualizada no emulador Netkit.

Uma máquina virtual pode ser configurada para atuar como um dispositivo específico na rede, por exemplo, um roteador, um servidor (http, e-mail, dns, etc), desde que haja software para tanto. Dessa forma, a máquina dht executou a implementação modificada da BambooDHT e operou como nó DHT GW no experimento. Todas as máquinas virtuais possuíram configuração homogênea: Debian Linux, kernel 2.6.26-5, e 256 MB de memória RAM. A máquina hospedeira, a qual virtualizou a rede, possuía as seguintes configurações: Ubuntu Linux, kernel 2.6.38-13, processador Intel(R) Core(TM)2 Duo CPU 2.26GHz, 4GB de RAM.

Uma vez que o mecanismo de Sessões é implementado no espaço de usuário e somente é executado nos nós-fim, sua implantação na máquina virtual foi simples. As APIs utilizadas na implementação foram baixadas e instaladas no sistema de arquivo da máquina virtual a partir

⁴<http://www.nongnu.org/quagga/>

do repositório de pacotes da distribuição Debian. Após a instalação das dependências, então foi necessário apenas recompilar o mecanismo de sessão nas máquinas cliente e servidor.

5.2.2 Impacto na vazão fim-a-fim

As mensagens enviadas pelo cliente ao servidor tiveram comprimento uniforme de 1MB. Para o envio de uma mensagem foram utilizadas requisições (nesse caso, os blocos de dados que partiam da aplicação) de tamanho variado, iniciando em 1KB e aumentando exponencialmente até 1MB. Foi coletada uma amostra de 200 medições de *goodput* para cada uma dessas requisições. A Figura 5.4 apresenta um gráfico que compara as médias obtidas no experimento com e sem o uso das Sessões.

Figura 5.4: *Goodput* da transmissão com e sem o uso das Sessões entre as máquinas cliente e servidor da rede virtualizada ilustrada na Figura 5.3.

A Tabela 5.3 apresenta as médias das amostras do *goodput* com ($\bar{x}_{cs}$) e sem ($\bar{x}_{ss}$) o uso das Sessões, além da sobrecarga (O) em porcentagem para cada bloco de dados utilizado no envio da mensagem de 1MB.

Desconsiderando o comprimento do bloco e analisando toda a amostra, o emissor apresentou média geral de *goodput* de 20.11 MBps sem o uso das sessões e de 18.70 MBps com o uso das sessões. Dessa forma, no experimento realizado, a sobrecarga média $\bar{O}$ das Sessões é de 6.98% de degradação na taxa de transmissão da aplicação.

Tabela 5.3: Sobrecarga do uso das sessões a partir dos blocos utilizados no envio.

bloco	$\bar{x}_{cs}$	$\bar{x}_{ss}$	O
1 KB	16.72	17.32	3.42
2 KB	16.93	17.98	5.86
4 KB	16.58	18.17	8.77
8 KB	17.27	18.68	7.56
16 KB	17.95	19.20	6.50
32 KB	18.57	20.42	9.06
64 KB	18.90	21.38	11.58
128 KB	19.33	21.92	11.79
256 KB	20.40	21.90	6.85
512 KB	21.19	21.93	3.37
1 MB	21.88	22.29	1.83

Conclusões

Esta tese introduz o conceito de Sessões de Comunicação Tolerantes a Rupturas. Tal conceito fundamenta-se em estudos anteriores sobre mobilidade na Internet e considera cenários de comunicação intermitente previstos em sistemas NGN. Diferentemente dos protocolos existentes, o tratamento da mobilidade é realizado através de uma arquitetura modular de propósito geral implementada totalmente no espaço de usuário sobre a interface de Sockets. Tal arquitetura é uma camada abstrata de Sessão que opera na forma de um *middleware* extensível, permitindo que novas funcionalidades possam ser acopladas.

Funcionalidades variadas são agregadas em um único objeto que representa uma Sessão de comunicação. As sessões naturalmente possuem a vantagem da otimização de rotas fornecida no nível IP, uma vez que são implementadas entre as camadas de Transporte e Aplicação. De acordo com a abordagem utilizada, nenhuma modificação na pilha de protocolos TCP/IP é necessária para implantar as Sessões nos nós-fim.

O fluxo de transmissão entre os pares é controlado conforme estados bem definidos na comunicação. Tal controle é determinado pelos mecanismos de detecção de mudança de estado de enlace e de detecção de par indisponível. O primeiro, além de traduzir os eventos disparados pelo *kernel* (com o evento de mobilidade) para o subconjunto de eventos especificados no padrão IEEE 802.21, reage a tais eventos alterando o estado de enlace do nó móvel. O segundo estende o mecanismo de *keepalive* do protocolo TCP para operar no nível das sessões. Ambos os mecanismos proporcionam redução de tempo no restabelecimento de sessões em andamento devido à rápida detecção.

As Sessões mantêm informações sobre os nós e a função que cada um desempenha na comunicação em andamento. Isso evita o envio de requisições desnecessárias à entidade de

registro de localização, otimiza o acesso a estruturas de dados internas e é determinante para a sincronização de cliente e servidor na (re) abertura de sessão.

O mecanismo de autenticação combina autenticação mútua baseada em desafio-resposta, PKI, registro de localização dos nós e validação local de sessão. Uma chave diferente pode ser selecionada para cada comunicação ativa, o que permite especificar a autenticação por cada sessão ativa da Aplicação, ao invés de uma autenticação generalizada por nó. Ataques de rede, como personificação, reprodução e MITM, são tratados durante as fases de abertura e reabertura de sessão com esse mecanismo de proteção.

As Sessões foram avaliadas em experimentos conduzidos em ambiente real e emulado. Os resultados obtidos validam a hipótese formulada através de uma comparação com o desempenho da solução de mobilidade na camada IP e com protocolos legados da pilha TCP/IP. Do ponto de vista da disponibilidade, a reabertura de sessões suspensas é realizada em um curto MTTR, se comparada ao tratamento do protocolo *Mobile IPv6*. Prover curtos intervalos de MTTR no tratamento de falhas de comunicação é fundamental para protocolos de Gerenciamento de Mobilidade IP em redes futuras.

6.1 Contribuições

Com o surgimento de novas arquiteturas de Redes de Computadores, como NGN e VCNs, e tecnologias de comunicação sem fio, há uma tendência de se utilizar células de transmissão menores para ampliar o acesso móvel. Essas células seriam compostas, por exemplo, pelas inúmeras WLANs na borda da Internet. Isso significa que cenários de comunicação móvel tendem a ser mais dinâmicos, o que resultarão em *handovers* e rupturas mais frequentes com os dispositivos de alta mobilidade. Portanto, a solução provida com o mecanismo de sessão mostra-se revelante para as redes futuras.

Recentemente os equipamentos de conectividade IP, como *switches* e roteadores, estão passando por um processo de inovação, em que o controle do tráfego da rede é desacoplado do equipamento. Essa nova abordagem, denominada *Software-Defined Networking* (SDN) [35], propõe gerenciar o tráfego por meio de softwares em PCs que atuam como controladores. O uso de controladores torna a rede programável, permitindo que novos protocolos de comunicação e experimentos possam ser executados em redes reais sem prejudicar os usuários e os recursos compartilhados nessas redes. Isso evita custos com a implementação de *testbeds* experimentais para a validação de novos protocolos de rede. O mecanismo de sessão proposto tem abordagem semelhante de gerenciamento. A modularização é tal que permite que o fluxo da aplicação seja controlado por um monitor assíncrono, o qual observa as condições do enlace e da conexão TCP e toma ações programáveis conforme os eventos de mobilidade.

Com as novas tecnologias para Web e os novos paradigmas, como Computação em Nuvem, onde há transparência de localização, e Internet das Coisas, com a possibilidade de endereçar

uma infinidade de objetos, os navegadores se tornam a interface generalizada para acesso aos recursos distribuídos na Internet. Nesse sentido, cada vez mais soluções de comunicação estão sendo providas no nível da Aplicação. No mecanismo de sessão, o qual é uma camada abstrata de software com operação transparente à Aplicação, a localização do recurso também é transparente. O recurso é referenciado por um identificador único e imutável em um amplo espaço de nomes de 160 bits, de modo que sua localização, a qual é passível de mudanças, passa a ser uma informação associada ao identificador.

Do ponto de vista das funcionalidades, o mecanismo de sessão proposto mostra-se abrangente, contemplando os requisitos (definidos na Seção 3.2) para gerenciar mobilidade fim-a-fim. Esses requisitos, os quais foram utilizados como princípios de projeto, podem ser utilizados como parâmetros para uma comparação entre os principais protocolos de mobilidade. Como apresenta a Tabela 6.1, agrupando as funcionalidades desejadas e características de um gerenciador de mobilidade ideal evidencia-se, então, as contribuições de cada solução.

Tabela 6.1: Comparação entre as principais soluções de mobilidade de acordo com requisitos para gerenciamento de mobilidade fim-a-fim.

Parâmetros	MIPv4	MIPv6	HIP	TCP-Migrate	ROCKS	TIPS	Sessões
Geren. Mobilidade	Inf.	Inf.	F	F	F	F	F
Implementação	EK	EK	EK	EK	EU	EU	EU
Camada de Operação	3	3	3,5	4	4, 5	5	4,5
Suporte ao Transporte	-	-	TCP	TCP	TCP	TCP	TCP
Suporte à Rede	IPv4	IPv6	IPv4/IPv6	IPv4	IPv4	IPv4	IPv4/IPv6
Identificadores	-	-	HIT	Token	IC	<i>mob_id</i>	<i>h_id</i>
Tipo de mobilidade	SJ	SJ	SJ/*DJ	SJ/*DJ	SJ	SJ/*DJ	SJ/*DJ
Geren. Localização	HA/FA	HA	RVS	DDNS	-	DHT	DHT
Atrasos e Rupturas	-	-	-	-	-	-	Tolerante
Sinalização	Alta	Alta	Média	Média	Média	Baixa	Baixa
Confiabilidade	-	-	-	-	IF	BE	BE
Segurança	-	ALR	IPSec	ECDH	DF	-	RFC 6287
Custo de implantação	Alto	Alto	Alto	Médio	Baixo	Baixo	Baixo
Transparente	OT	OT	OT	OT	OLI	OLI	OT
Detecção de Mob.	U-MN	U-MN	U-MN	U-MN-N	U-MN-N	R-B-N	A-B-N
Lat. desconexão	-	10 s	-	-	-	1 s	20 - 180 ms

Legenda: Infraestrutura (Inf.), Fim-a-fim (F), Espaço de *Kernel* (EK), Espaço de Usuário (EU), *Host Identity Tag* (HIT), Identificador de Sessão (Token), Identificador de Conexão negociado (IC), Identificador único de nó móvel (*mob_id*, *h_id*), *Single Jump* (SJ), *Double Jump* (*DJ) – desde de que os servidores móveis forem endereçados com endereços roteáveis na Internet, *Home Agent* (HA), *Foreign Agent* (FA), *Rendezvous Server* (RVS), *Dynamic DNS* (DDNS), *Distributed Hash Table* (DHT), *In-flight-buffers* (IF), *Bufferização no Envio* (BE), Autenticação durante Registros de Localização (ARL), *Elliptic Curve Diffie–Hellman* (ECDH), *Diffie–Hellman* (DF), Operação Transparente (OT), Operação Levemente Intrusiva (LI), Unilateral (U), *Mobile Node* (MN), Notificação ao par corresponde ou a entidade de registro (N), Reativo (R), Bilateral (B), Assíncrono (A).

As Sessões reúnem as principais funcionalidades e características previstas para um gerenciador de mobilidade fim-a-fim, provendo: tolerância a atrasos e rupturas, confiabilidade no nível da aplicação, restabelecimento seguro de sessão, baixa sobrecarga e latência de desconexão, baixo custo de implantação, suporte em rede IPv4 e IPv6, bem como suporte à mobilidade em cenários complexos, como *Double Jump*. Tais funcionalidades não estão previstas na íntegra nas soluções comparadas.

6.2 Publicações geradas

As contribuições, as quais foram geradas a partir de pesquisas direta ou indiretamente relacionadas a esta tese, foram publicadas nos trabalhos citados a seguir.

O artigo [68] descreve um mecanismo de otimização de *handovers* baseado em Ciência de Contexto. As informações contextuais de fontes variadas são capturadas no momento que antecede o *handover* e processadas em conjunto com as preferências do usuário. O mecanismo foi avaliado através de *handovers* de nível 2¹ em uma rede de teste.

Em [37] é descrita uma abordagem complementar para melhorar a exatidão e praticidade na manipulação de informações de contexto, as quais são utilizadas para otimizar procedimentos de *handovers*. As experiências de conectividade dos usuários em movimento são armazenadas e recuperadas através de rotas, agregando tempo e localização para representar o hábito. As experiências então são compartilhadas em comunidades virtuais para atingir outros usuários potencialmente interessados, os quais compartilham movimentos similares.

Um resumo [30] relata as descobertas iniciais a partir da comparação do desempenho do protocolo MIPv6 e o suporte à mobilidade implementado no nível de Aplicação em TIPS. Experimentos iniciais no *testbed* (descrito na Seção 3.1.1) fizeram parte de um conjunto preliminar de testes do projeto de pesquisa de doutorado em que, na época, especulava estratégias de otimização para o Gerenciamento de Mobilidade em camadas superiores. Posteriormente, os resultados preliminares discutidos no Capítulo 3 e as técnicas de implementação para aplicações cientes de mobilidade utilizadas foram publicados em [32].

Características e técnicas fundamentais das atuais tecnologias de *handover* foram descritas no capítulo de livro [41]. Nesse trabalho é discutida uma infraestrutura baseada em contexto e ontologias para gerenciar informação e fomentar cenários inteligentes, eficientes e rentáveis para o gerenciamento de *handovers* em redes NGN.

As Sessões Tolerantes a Rupturas são introduzidas em [29]. Os resultados avaliados a partir dos experimentos realizados para mensurar latência das operações das Sessões são relatados também nesse artigo. Detalhes da abordagem de segurança utilizada nas fases abertura de sessão fechada e reabertura de sessão suspensa, bem como a comparação dos resultados obtidos com outras abordagens de *handshaking* foram publicados em [31].

¹Ver definição de “*Handover L2*” na Seção Glossário.

6.3 Trabalhos futuros

Aspectos que requerem maiores investigações e funcionalidades ainda não suportadas pelas Sessões são descritos a seguir. A expectativa é que tais aspectos e funcionalidades estendam esta pesquisa e sejam desdobradas em outros trabalhos científicos.

6.3.1 Mobilidade do servidor entre redes privadas

Nós móveis geralmente são endereçados nas redes de acesso através de mecanismos automáticos, como DHCP [12]. Essas redes, contudo, operam normalmente como redes privadas e utilizam um *gateway* NAT para prover acesso global à Internet. Nesse caso, a mobilidade do servidor para uma rede privada implica no problema da travessia NAT, não havendo rotas de acesso para clientes atingirem servidores escondidos atrás de roteadores NAT na Internet.

Nesse contexto, seria interessante incorporar suporte no nível das Sessões para permitir também comunicação fim-a-fim mesmo quando servidores são migrados para redes privadas e, ao mesmo tempo, evitar o uso de um terceiro nó *proxy* para intermediar a comunicação entres os pares móveis. Uma entidade que provê a localização de nó móvel com informações completas de acesso, como endereço-porta interno e externo, já é parte do funcionamento das Sessões. Assim, estratégias para estabelecimento de conexões TCP, como *hole punching* [59] em redes P2P, por exemplo, poderiam ser incorporadas nas fases de (re) abertura de sessão.

6.3.2 Proatividade e IEEE 802.21

Com o objetivo de minimizar ainda mais o tempo de desconexão t_{down} , um tópico de pesquisa de interesse é o tratamento de mobilidade proativa. Um caminho, à primeira vista, seria através de mecanismos de predição de rupturas do nó móvel, por exemplo, com o suporte de Redes Neurais Artificiais. Informações da camada de Enlace do nó móvel, como a relação sinal-ruído (SNR), taxa de erro de pacote, taxa de transmissão, etc, poderiam ser utilizadas como entradas para a rede neural em sua fase de treinamento.

Um mecanismo de predição poderia operar juntamente com o Gerenciamento de Gatilhos de Mobilidade das Sessões. Nesse caso, seria importante que a predição operasse também no formato de eventos especificados pelo protocolo IEEE 802.21. Assim, com a predição de ruptura, o evento LINK_GOING_DOWN poderia ser disparado pelo Gerenciador de Gatilhos, o que permitiria antecipar o tratamento de mobilidade. Com a proatividade, especula-se que o tempo de desconexão t_{down} possa ser ainda mais minimizado.

6.3.3 Otimização de transmissões após *handovers*

Ao perceber que o enlace está disponível com a notificação do evento LINK_UP, a estratégia adotada pelas Sessões é imediatamente restabelecer a comunicação fim-a-fim através de

uma nova conexão TCP. Considerando um cenário que, na atual localização do nó móvel, há sobreposição de redes de acesso sem fio com o fornecimento de taxas de transmissão semelhantes entre elas, a Janela de Congestionamento da conexão em andamento pode se expandir, atingindo a capacidade de transmissão da rede atual. Com mudança de rede, a nova conexão TCP reduziria o comprimento da Janela em 1 MSS (*Maximum Segment Size*) novamente com a fase de *Slow Start* iniciada na nova rede. Essa redução do comprimento da Janela, contudo, resultaria em uma diminuição abrupta da taxa de transmissão da Aplicação, considerando a transmissão ao longo da duração de uma sessão.

Seria interessante que, nesses casos, novas conexões mantivessem o histórico de desempenho, preservando o valor de Janela de Congestionamento expandida na conexão na rede anterior. O Controle de Congestionamento então poderia, a partir da janela preservada, descobrir a capacidade de transmissão da nova rede, evitando o início da fase de *Slow Start* de uma nova conexão. No pior caso, se a nova rede estiver congestionada no momento de *handover* e a taxa de transmissão fornecida for inferior ao da rede anterior, o protocolo TCP ajustaria o valor de janela normalmente a essa situação. Entretanto, um problema à primeira vista seria como manipular parâmetros do protocolo TCP, como o comprimento de Janela de Congestionamento, a partir do espaço de usuário. Vale ressaltar que umas contribuições das Sessões propostas é a fácil implantação, a qual é obtida através da implementação de uma camada abstrata de Sessão no espaço de usuário sem necessitar de nenhuma modificação na pilha de protocolos TCP/IP.

6.3.4 *Multihoming e Multipathing*

Aliada à capacidade de comunicação do dispositivo com suporte à múltiplas interfaces de rede, sistemas NGN preveem a disponibilidade de múltiplas redes de acesso sem fio nas localizações do nó móvel. Com o objetivo de maximizar a taxa de transmissão das Sessões, seria interessante aproveitar a diversidade de acesso através do suporte à *multihoming e multipathing*.

Múltiplas conexões TCP poderiam ser estabelecidas com o nó correspondente sobre as diferentes interfaces de rede do dispositivo. Isso permitiria o envio concorrente de pedaços de mensagens sobre os vários *paths* de uma sessão. O desafio, a princípio, é gerenciar no nível das Sessões as diversas conexões entre os pares como *paths* de uma sessão particular. A remontagem de uma mensagem a partir de fragmentos que podem chegar desordenados sobre diferentes *paths* da sessão é um problema logo identificado. Segurança e integridade na transmissão são também questões que requerem investigação aprofundada.

6.3.5 **Cenários de comunicação altamente dinâmicos**

Ao tratar de forma segura e eficiente comunicações sensíveis a ruptura em nós móveis, especula-se, neste momento, que as Sessões propostas possuem requisitos para gerenciar mobilidade IP em cenários de comunicação altamente dinâmicos. Tais cenários são previstos em

redes VCNs e na comunicação entre veículos autônomos, como VANT (Veículo Aéreo Não Tripulado) e VTNT (Veículo Terrestre Não Tripulado).

Entretanto, para avaliar o desempenho das Sessões é fundamental que experimentos sejam conduzidos em tais cenários. Com isso, propor otimizações na arquitetura para as eventuais ineficiências identificadas. Essa avaliação poderia ser realizada em ambientes simulados, uma vez que experimentos dessa natureza em ambientes reais são considerados pouco viáveis, por diversas razões, a contar pelo custo de implementação e implantação de um *testbed* envolvendo veículos autônomos.

Referências Bibliográficas

- [1] *IEEE Standard for Information Technology - Portable Operating System Interface (POSIX) - Base Specifications, Issue 7 - IEEE Std 1003.1-2008: Revision of IEEE Std 1003.1-2004*. IEEE Computer Society and The Open Group, 2008.
- [2] *IEEE Standard for Local and Metropolitan Area Networks- Part 21: Media Independent Handover, IEEE Std 802.21-2008*. IEEE Computer Society - LAN/MAN Standards Committee, 2009.
- [3] AHRENHOLZ, J. Host Identity Protocol Distributed Hash Table Interface. draft-irtf-hiprg-dht-05, Internet-Draft, Internet Engineering Task Force, IETF, December 2011.
- [4] AKYILDIZ, I., MCNAIR, J., HO, J., UZUNALIOGLU, H., AND WANG, W. Mobility management in next-generation wireless systems. *Proceedings of the IEEE* 87, 8 (August 1999), 1347 –1384.
- [5] AKYILDIZ, I., XIE, J., AND MOHANTY, S. A survey of mobility management in next-generation all-ip-based wireless systems. *IEEE Wireless Communications Magazine* 11, 4 (Aug. 2004), 16 – 28.
- [6] AL-SHRAIDEH, F. Host Identity Protocol. In *ICN/ICONS/MCL '06: Proceedings of the IEEE International Conference on Networking, IEEE International Conference on Systems and IEEE International Conference on Mobile Communications and Learning Technologies* (April 2006), p. 203.
- [7] AL-SURMI, I., OTHMAN, M., AND MOHD ALI, B. Mobility management for IP-based next generation mobile networks: Review, challenge and perspective. *Journal of Network and Computer Applications* 35, 1 (2012), 295 – 315.
- [8] ATKINSON, R. ILNP Concept of Operations. RFC draft-rja-ilnp-intro-11.txt, Request for Comments, Internet Engineering Task Force, IETF, July 2011.
- [9] AUDET, F., AND JENNINGS, C. Network Address Translation (NAT) Behavioral Requirements for Unicast UDP. RFC 4787, Request for Comments, Internet Engineering Task Force, IETF, January 2007.
- [10] BRADEN, R. Requirements for Internet Hosts - Communication Layers. RFC 1122, Request for Comments, Internet Engineering Task Force, IETF, October 1989.
- [11] CESPEDES, S., SHEN, X., AND LAZO, C. IP mobility management for vehicular communication networks: challenges and solutions. *IEEE Communications Magazine* 49, 5 (2011), 187 –194.
- [12] DROMS, R. Dynamic Host Configuration Protocol. RFC 2131, Request for Comments, Internet Engineering Task Force, IETF, March 1997.

- [13] EASTLAKE, D. Secure Domain Name System Dynamic Update. RFC 2137, Request for Comments, Internet Engineering Task Force, IETF, April 1997.
- [14] EDDY, W. M. At what layer does mobility belong? *IEEE Communications Magazine* 42, 10 (Oct. 2004), 155–159.
- [15] EGEVANG, K., AND FRANCIS, P. The IP Network Address Translator (NAT). RFC 1631, Request for Comments, Internet Engineering Task Force, IETF, May 1994.
- [16] FARINACCI, D., FULLER, V., MEYER, D., AND LEWIS, D. Locator/ID Separation Protocol (LISP). RFC draft-ietf-lisp-22.txt, Request for Comments, Internet Engineering Task Force, IETF, February 2012.
- [17] FORD, A., RAICIU, C., HANDLEY, M., BARRE, S., AND IYENGAR, J. Architectural Guidelines for Multipath TCP Development. RFC 6182, Request for Comments, Internet Engineering Task Force, IETF, March 2011.
- [18] GUHA, S., AND FRANCIS, P. Characterization and measurement of TCP traversal through NATs and firewalls. In *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement (IMC '05)* (2005), USENIX Association, pp. 18–18.
- [19] GUNDAVELLI, S., LEUNG, K., DEVARAPALLI, V., CHOWDHURY, K., AND B., P. Proxy Mobile IPv6. RFC 5213, Request for Comments, Internet Engineering Task Force, IETF, August 2008.
- [20] GURTOV, A. *Host Identity Protocol (HIP): Towards the Secure Mobile Internet*. John Wiley & Sons Ltd, 2008.
- [21] GUSTAFSSON, E., AND JONSSON, A. Always best connected. *IEEE Wireless Communications* 10, 1 (2003), 49 – 55.
- [22] HUANG, G., BEAULIEU, S., AND ROCHEFORT, D. A Traffic-Based Method of Detecting Dead Internet Key Exchange (IKE) Peers. Request for Comments, Internet Engineering Task Force, IETF.
- [23] JACOBSON, V., BRADEN, R., AND BORMAN, D. TCP Extensions for High Performance. RFC 1323, Request for Comments, Internet Engineering Task Force, IETF, May 1992.
- [24] JOHNSON, D., PERKINS, C., AND ARKKO, J. Mobility Support in IPv6. RFC 3775, Request for Comments, Internet Engineering Task Force, IETF, June 2004.
- [25] JOHNSON, D., PERKINS, C., AND ARKKO, J. Mobility Support in IPv6. RFC 6275, Request for Comments, Internet Engineering Task Force, IETF, July 2011.
- [26] KIMURA, B. Y. L. TIPS: Uma Plataforma para Mobilidade IP sobre a Camada de Transporte. Master's thesis, Universidade Federal de São Carlos - UFSCar, Maio 2008.
- [27] KIMURA, B. Y. L., AND GUARDIA, H. C. TIPS: Mobilidade IP sobre a Camada de Transporte. In *Anais do 26º Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC 2008)* (Maio 2008), pp. 245 – 258.
- [28] KIMURA, B. Y. L., AND GUARDIA, H. C. TIPS: Wrapping the Sockets API for Seamless IP Mobility. In *ACM SAC'08: the 23rd Annual ACM Symposium on Applied Computing* (March 2008), pp. 1940–1945.
- [29] KIMURA, B. Y. L., GUARDIA, H. C., AND MOREIRA, E. S. Disruption-Tolerant Sessions for Seamless Mobility. In *WCNC 2012: the 2012 IEEE Wireless Communications and Networking Conference* (April 2012), pp. 2412 – 2417.

- [30] KIMURA, B. Y. L., AND MOREIRA, E. S. Mobility at the application level. In *INFOCOM 2009: Workshops of the 28th IEEE International Conference on Computer Communications* (2009), pp. 1 – 2.
- [31] KIMURA, B. Y. L., YOKOYAMA, R. S., GUARDIA, H. C., AND MOREIRA, E. S. Secure Connection re-establishment for Session-Based IP Mobility. In *CBSEC 2012: II Brazilian Conference on Critical Embedded Systems* (May 2012), pp. 58 – 63.
- [32] KIMURA, B. Y. L., YOKOYAMA, R. S., LOPES, R. R. F., GUARDIA, H. C., AND MOREIRA, E. S. Prototyping applications to handle connection disruptions in end-to-end Host Mobility. In *WONS 2010: the 7th IEEE International Conference on Wireless On-demand Network Systems and Services* (February 2010), pp. 1 – 8.
- [33] KODALI, R. Fast Handovers for Mobile IPv6. RFC 4068, Request for Comments, Internet Engineering Task Force, IETF, July 2005.
- [34] KOPONEN, T., ERONEN, P., AND SÄRELÄ, M. Resilient connections for SSH and TLS. In *USENIX '06: Proceedings of the Annual Technical Conference* (May 2006), pp. 30–30.
- [35] LANTZ, B., HELLER, B., AND MCKEOWN, N. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks* (2010), pp. 1 – 6.
- [36] LEFFLER, S. J., MCKUSICK, M. K., KARELS, M. J., AND QUARTERMAN, J. S. The Design and Implementation of the 4.3 BSD Unix Operating System. In *Addison-Wesley series in Computer Science* (1989), Addison-Wesley.
- [37] LOPES, R. R. F., YOKOYAMA, R. S., KIMURA, B. Y. L., PAWAR, P., BEIJNUM, B. V., AND MOREIRA, E. S. Exploring user's habits and virtual communities to improve IP-connectivity management. In *WMCNT'09: First IEEE Workshop on Mobile Computing and Networking Technologies* (October 2009), pp. 1 – 6.
- [38] MALTZ, D. A., AND BHAGWAT, P. MSOCKS: An Architecture for Transport Layer Mobility. In *INFOCOM 1998: 17th IEEE Conference on Computer and Communications* (1998), vol. 3, pp. 1037 – 1045.
- [39] MANNER, J., AND KOJO, M. Mobility Related Terminology. RFC 3753, Request for Comments, Internet Engineering Task Force, IETF, June 2004.
- [40] MAO, Y., KNUTSSON, B., LU, H., AND SMITH, J. DHARMA: distributed home agent for robust mobile access. In *INFOCOM 2005: 24th IEEE Conference of the IEEE Computer and Communications Societies*. (March 2005), vol. 2, pp. 1196 – 1206.
- [41] MOREIRA, E. S., YOKOYAMA, R. R., PORTO, R. M. V., AND KIMURA, B. Y. L. *Handbook of Research on Mobility and Computing: Evolving Technologies and Ubiquitous Impacts*. IGI Global, 2011, ch. Technologies to Improve the Quality of Handovers: Ontologies, Contexts and Mobility Management, pp. 522 – 538.
- [42] MOSKOWITZ, R., NIKANDER, P., AND JOKELA, P. Host Identity Protocol. RFC 5201, Request for Comments, Internet Engineering Task Force, IETF, April 2008.
- [43] M'RAIHI, D., RYDELL, J., BAJAJ, S., MACHANI, S., AND NACCACHE, D. OCRA: OATH Challenge-Response Algorithm. RFC 6287, Request for Comments, Internet Engineering Task Force, IETF, June 2011.
- [44] NARTEN, T., NORDMARK, E., AND SIMPSON, W. Neighbor Discovery for IP Version 6 (IPv6). RFC 2461, Request for Comments, Internet Engineering Task Force, IETF, December 1998.

- [45] PERKINS, C. E. IP Mobility Support for IPv4. RFC 3344, Request for Comments, Internet Engineering Task Force, IETF, August 2002.
- [46] PERKINS, C. E. IP Mobility Support for IPv4, Revised. RFC 5944, Request for Comments, Internet Engineering Task Force, IETF, November 2010.
- [47] PIRI, E., AND PENTIKOUSIS, K. Towards a GNU/Linux IEEE 802.21 implementation. In *ICC'09: the 2009 IEEE international conference on Communications* (June 2009), pp. 1 – 5.
- [48] PIZZONIA, M., AND RIMONDINI, M. Netkit: easy emulation of complex networks on inexpensive hardware. In *TridentCom '08: Proceedings of the 4th International Conference on Testbeds and research infrastructures for the development of networks & communities* (2008), pp. 1 – 10.
- [49] POSTEL, J. Transmission Control Protocol. RFC 793, Request for Comments, Internet Engineering Task Force, IETF, September 1981.
- [50] RAZZAQUE, M. A., DOBSON, S., AND NIXON, P. Cross-Layer Architectures for Autonomic Communications. *J. Netw. Syst. Manage.* 15, 1 (2007), 13–27.
- [51] REKHTER, Y., MOSKOWITZ, B., KARREBERG, D., GROOT, G. J., AND LEAR, E. Address Allocation for Private Internets. RFC 1918, Request for Comments, Internet Engineering Task Force, IETF, February 1996.
- [52] RHEA, S., GODFREY, B., KARP, B., KUBIATOWICZ, J., RATNASAMY, S., SHENKER, S., STOICA, I., AND YU, H. OpenDHT: a public DHT service and its uses. *SIGCOMM Comput. Commun. Rev.* 35 (August 2005), 73 – 84.
- [53] ROSENBERG, J., MAHY, R., MATTHEWS, P., AND WING, D. Session Traversal Utilities for NAT (STUN). RFC 5389, Request for Comments, Internet Engineering Task Force, IETF, October 2008.
- [54] ROSENBERG, J., SCHULZRINNE, H., CAMARILLO, G., JOHNSTON, A., PETERSON, J., SPARKS, R., HANDLEY, M., AND SCHOOLER, R. SIP: Session Initiation Protocol. RFC 3261, Request for Comments, Internet Engineering Task Force, IETF, June 2002.
- [55] SHAH, P. A., YOUSAF, M., QAYYUM, A., AND HASBULLAH, H. B. Performance comparison of end-to-end mobility management protocols for TCP. *Journal of Network and Computer Applications* 35, 6 (2012), 1657 – 1673.
- [56] SNOEREN, A. C., AND BALAKRISHNAN, H. An end-to-end Approach to Host Mobility. In *MobiCom '00: 6th ACM Annual International Conference on Mobile computing and Networking* (August 2000), pp. 155 – 166.
- [57] SNOEREN, M. *A Session-Based Architecture for Internet Mobility*. PhD thesis, Massachusetts Institute of Technology, February 2003.
- [58] SOLIMAN, H., CASTELLUCCIA, C., EL MALKI, K., AND L., B. Hierarchical mobile IPv6 mobility. RFC 4140, Request for Comments, Internet Engineering Task Force, IETF, August 2005.
- [59] SRISURESH, P., FORD, B., AND KEGEL, D. State of Peer-to-Peer (P2P) Communication across Network Address Translators (NATs). RFC 5128, Request for Comments, Internet Engineering Task Force, IETF, March 2008.
- [60] STEVENS, W. R., FENNER, B., AND RUDOFF, A. M. *UNIX Network Programming: The sockets networking API*. Addison-Wesley, 2004, ch. Chapter 2, Section 11, Buffer Sizes and Limitations, TCP Output.

- [61] STEVENS, W. R., FENNER, B., AND RUDOFF, A. M. *UNIX Network Programming: The sockets networking API*. Addison-Wesley, 2004, ch. Chapter 7, Section 5, Generic Socket Options, SO_RCVBUF and SO_SNDBUF Socket Options.
- [62] TANENBAUM, A. S., AND WETHERALL, D. J. *Computer Networks, 5th Edition*. Pearson, 2011, ch. Chapter 1, Section 4.1, Reference Models - The OSI Reference Model.
- [63] THE LINUX MAN-PAGES PROJECT. Linux Programmer's Manual, Section 7: tcp - TCP protocol. <http://www.kernel.org/doc/man-pages/online/pages/man7/tcp.7.html>, September 2010.
- [64] THEOFILATOS, D., DAGIUKLAS, T., AND GATZOUNAS, D. Mobility Management Schemes for supporting multimedia services in All-IP Network Infrastructures. In *Proceedings of the 5th IEEE European Personal Mobile Communications Conference* (April 2003), pp. 302 – 306.
- [65] THOMSON, S., AND NARTEN, T. IPv6 Stateless Address Autoconfiguration. RFC 2462, Request for Comments, Internet Engineering Task Force, IETF, December 1998.
- [66] UDUGAMA, A. Manipulating the network environment using RTNETLINK. *Linux Journal* 2006, 145 (May 2006), 7–.
- [67] VIXIE, P., THOMSON, S., REKHTER, Y., AND BOUND, J. Dynamic Updates in the Domain Name System (DNS UPDATE). RFC 2136, Request for Comments, Internet Engineering Task Force, IETF, April 1997.
- [68] YOKOYAMA, R. S., KIMURA, B. Y. L. LOPES, R. R. F., AND MOREIRA, E. S. Uma Abordagem Ciente de Contexto para Handovers Orientados a Serviços em Ambientes NGN. In *I2TS 2008: 7th International Information and Telecommunication Technologies Symposium* (Novembro 2008), pp. –.
- [69] ZANDY, V. C., AND MILLER, B. P. Reliable Network Connections. In *MobiCom '02: 8th ACM International Conference on Mobile computing and Networking* (September 2002), pp. 95 – 106.

Glossário

Cross-layer

Um sistema *Cross-layer* opera de forma diferente de um sistema convencional de rede, em que as camadas da pilha de protocolos utilizam serviços providos somente pelas camadas adjacentes. Na abordagem *cross-layer* as camadas são adaptadas de acordo com informações pertinentes de quaisquer outras camadas para otimizar o desempenho fim-a-fim [50].

CN

Correspondent Node, o nó com o qual o nó MN se comunica na Internet. Pode ser um nó móvel ou estacionário [45].

Double-jump

Cenário de comunicação fim-a-fim onde ambos nós, MN e CN, ou cliente e servidor, são móveis. Esse cenário é complexo e normalmente requer um terceiro nó, que é fixo e acessível, para armazenar e resolver registros de localização dos nós móveis, principalmente dos nós servidores. Caso contrário, nós clientes não seriam capazes de saber de antemão a atual localização de servidores móveis para iniciar a comunicação ou retomar uma transmissão perdida.

Gerenciamento de Mobilidade

Mecanismo fundamental para permitir a continuidade da comunicação e dos serviços dos usuários enquanto seus dispositivos são deslocados para novas redes de acesso sem fio [4]. Refere-se à capacidade da rede e/ou do dispositivo móvel de manter seus usuários em comunicação enquanto eles percorrem células de comunicação de rádio dentro de uma sub-rede, entre sub-redes de mesmo domínio ou, ainda, entre diferentes redes [64].

Handover

handoff, *transição* na qual um nó móvel é submetido quando troca seu ponto de acesso com a rede [39], *i.e.* quando o dispositivo é deslocado em direção a outras células de cobertura ou redes sem fio.

Handover L2

Procedimento pelo qual o nó móvel troca a conectividade no nível de Enlace por outra, *e.g.* mudança de Ponto de Acesso é um *Handover L2* [24]. Na mesma rede IP, o *handover L2* é transparente ao roteamento da Camada de Rede [39], uma vez que o procedimento é conduzido pela própria tecnologia de Enlace, não havendo implicações para as camadas superiores, exceto pelo tempo de desconexão.

Handover L3

Procedimento subsequente ao *Handover L2* pelo qual o nó móvel troca de roteador de acesso e muda para outra rede IP [24]. *Handovers* no nível da Camada de Rede afetam as camadas superiores, pois envolve mudança de endereço IP quando o nó móvel é migrado entre redes de acesso na Internet, causando rupturas de comunicações iniciadas em redes anteriores.

Handover Vertical

Vertical Handover; uma transição (ver *Handover*) no qual o nó móvel é movido entre redes de tecnologias distintas de transmissão sem fio. Por exemplo, um *handover* de uma rede 3G para uma rede WLAN [39].

MN

Mobile Node, um nó IP capaz de mudar seu ponto de acesso de rede [39], o qual é controlado por usuário móvel.

Single-jump

Cenário de comunicação fim-a-fim onde apenas um lado da comunicação é móvel. Geralmente o cenário é configurado por um nó móvel cliente que se comunica com um ou mais nós correspondentes que são servidores fixos (CN). Com a mobilidade de nós cliente, se comparado ao *Double-jump*, *single-jump* é um cenário mais simples, pois normalmente não é necessário registro de localização dos nós.

Transição

Ver *Handover*.

Transição transparente

Seamless Handover, ocorre quando protocolos, aplicações, ou usuário finais não detectam nenhuma mudança, durante ou após as transições (ver *handover*), na capacidade do serviço, na segurança, ou na qualidade [39]. Esse termo muitas vezes é referenciado como “transição transparente” ou “mobilidade transparente” na literatura.