TIPS: Mobilidade IP sobre a Camada de Transporte

Bruno Y. L. Kimura, Hélio C. Guardia

Departamento de Computação – Universidade Federal de São Carlos (UFSCar)
Caixa Postal 676 – 13565-905 – São Carlos – SP– Brasil

{bruno_kimura, helio}@dc.ufscar.br

Abstract. *TIPS (Transparent IP Sockets) is a wrapping for the main functions of the Berkeley Sockets API intended to provide transparent IP Mobility in the application code. Implemented on top of the Transport Layer and in User Space, TIPS allows Horizontal and Vertical Handoffs for both UDP transmissions and TCP connections, as well as client and server migrations (single and double jump). Lossless migrations and lossless data transmissions are also provided. Location information of the mobile terminal is stored in a Distributed Hash Table and is accessible under a hash index proposed as a strategy for terminal/application identification regardless of its locations.*

Resumo. *TIPS (Transparent IP Sockets) é uma adaptação das principais funções fornecidas pela API Berkeley Sockets destinada a prover mobilidade IP transparente no código da aplicação. Implementado sobre a Camada de Transporte e no Espaço de Usuário, TIPS permite o suporte à Mobilidade em Transições (Handoffs) Horizontais e Verticais tanto para as transmissões UDP quanto para as conexões TCP, bem como o deslocamento de terminais clientes e servidores (single e double jump). Deslocamentos e transmissões sem perda de dados também são suportados. Informações de localização dos terminais móveis são armazenadas em uma Tabela Hash Distribuída e acessadas sob um índice hash proposto como uma estratégia de identificação dos terminais/aplicações móveis independente de localização.*

1. Introdução

A difusão cada vez maior de dispositivos portáteis com acesso à Internet tem provocado o constante aumento de aplicações móveis baseadas na plataforma IP. Do ponto de vista do acesso aos serviços, os usuários podem se deslocar por diferentes redes e sub-redes da Internet e utilizar tecnologias distintas de comunicação. Após o deslocamento e a alteração do endereço IP associado à interface ativa do dispositivo, é possível iniciar manualmente um novo acesso aos serviços de e-mail ou www, por exemplo. Algumas tecnologias de acesso sem fio, como as redes de telefonia celular e o padrão *Mobile Wi-Max* (802.16e), permitem a mobilidade do usuário, normalmente fazendo ajustes dinâmicos no roteamento de pacotes para os dispositivos conectados a estações base variadas. Com isso, preserva-se os fluxos de dados estabelecidos, mesmo durante as transições horizontais (*handoff horizontal*).

Outros aspectos da mobilidade, contudo, ainda requerem tratamento dependente de protocolos, de características do dispositivo ou do sistema operacional utilizado. Transições verticais, associadas à mudança de tecnologia, de uma rede celular para uma rede local sem fio, por exemplo, implicam em alterações de endereçamento que podem inviabilizar a continuidade da execução de uma aplicação.

A solução baseada em agentes locais e remotos e roteamento triangular provida pelo protocolo **Mobile IP** [Perkins 2002], por exemplo, ainda é pouco suportada. Ainda, aplicações associadas à transmissão contínua de dados, voz ou imagens e aplicações orientadas à conexão não estão preparadas para suportar tal mobilidade. Conexões estabelecidas são corrompidas após o deslocamento e a aquisição de um novo endereço IP na nova localização do terminal móvel. Assim, um mecanismo para o suporte à Mobilidade IP transparente às aplicações mostra-se necessário.

Neste contexto, esse artigo apresenta uma solução para mobilidade de aplicações, denominada **TIPS** - *Transparent IP Sockets.* Implementado sobre a Camada de Transporte e no Espaço de Usuário, TIPS atende os diferentes tipos de transições, como o *handoff vertical* e *horizontal*, sendo independente de tecnologia de comunicação. Na prática, trata-se de um mecanismo tolerante às falhas de conexão para lidar com alterações de endereçamento IP de forma transparente às aplicações e mantendo a persistência tanto das transmissões UDP quanto das conexões TCP durante o deslocamento dos usuários móveis. Para tanto, TIPS realiza uma adaptação da API Berkeley Sockets [Leffler et al. 1989] de forma a preservar a lógica das aplicações e prover continuidade nos fluxos de dados estabelecidos mesmo após a mobilidade. TIPS combina soluções conhecidas para os problemas envolvidos com a Mobilidade IP, incluindo: identificação única dos terminais/aplicações móveis; deslocamento sem perda de dados; mobilidade de ambas as partes da comunicação; e persistência de conexões IP. A arquitetura TIPS foi implementada considerando os aspectos de escalabilidade, baixo custo de implantação e sem adição de sobrecargas nas transmissões.

Este artigo apresenta uma visão detalhada da operação de TIPS, ampliando a descrição de sua arquitetura e a sua avaliação apresentadas anteriormente em [Kimura e Guardia 2008], tratando os aspectos da mobilidade horizontal e vertical.

2. Trabalhos Relacionados

Soluções para mobilidade existentes atualmente ainda são dependentes de infra-estrutura de rede para suportar transições transparentes. O uso de redirecionadores para intermediar as transmissões é uma abordagem utilizada. Nesse sentido, o modelo de roteamento triangular através de agentes de mobilidade locais e remotos, introduzido pelo protocolo **Mobile IP** [Perkins 2002], também é utilizado para migrar sockets entre diferentes terminais [Bernaschi et al. 2007]. Igualmente, as soluções **MSOCKS** [Maltz e Bhagwat 1998] e **MobCast** [Lee et al. 2007] utilizam *Proxies* para o encaminhamento de datagramas durante o deslocamento dos nós móveis. Outras soluções, ainda, adicionam sobrecargas na comunicação pela formatação própria das mensagens, como o protocolo **HIP** [Moskowitz e Nicander 2006] que embute a identificação dos *hosts* e outras informações nos pacotes de dados. Mesmo existindo mecanismos de tolerância às falhas de conexões e retransmissões de dados perdidos durante os deslocamentos, como o **ROCKS** [Zandy e Miller 2002], a existência de NATs e *firewalls* inviabilizam a comunicação na rede visitada. Outras soluções, como o protocolo **SIP** [Rosenberg et al. 2002] que mesmo provendo o suporte à mobilidade de terminal e de sessão, não suportam comunicações orientadas à conexão.

3. Arquitetura TIPS

TIPS é uma solução que adapta a API Berkeley Sockets [Leffler et al. 1989] para o suporte à Mobilidade IP nas comunicações fim-a-fim considerando o deslocamento de ambas as partes da comunicação de forma transparente à lógica das aplicações. As

primitivas de comunicação fornecidas pela API de *Sockets* foram reimplementadas sobre a Camada de Transporte e no Espaço de Usuário tratando dos principais problemas envolvidos com Mobilidade IP, incluindo: identificação dos terminais/aplicações móveis, tratamento dos dados perdidos durante o deslocamento, e recomposição da comunicação após o deslocamento.

No modelo de comunicação proposto, as transmissões entre os terminais TIPS MN (*Mobile Node*) e TIPS CN (*Correspondent Node*), como mostra a Figura 1, são realizadas de forma fim-a-fim, sem a existência de redirecionadores. No entanto, para suportar o deslocamento de ambas as partes da comunicação (aplicações cliente e servidora, ou ainda, *single* e *double jump*), na arquitetura TIPS existe um mecanismo de armazenamento para as informações de localização dos terminais móveis, denominado ***Servidor de Registro***. Neste modelo, toda vez que ocorre uma mudança de endereço IP um novo registro é armazenado em um *Servidor de Registro* o qual implementa uma *Tabela Hash Distribuída* (DHT) de propósito geral. O registro é uma estrutura XML que contém a localização atual do terminal móvel e é armazenado sob um índice *hash*, *mod_id*, nesta tabela. Este mesmo índice corresponde à identificação única do terminal/aplicação móvel de forma a tornar a informação de localização disponível para as consultas dos terminais envolvidos na comunicação fim-a-fim. Esses procedimentos são explicados mais detalhadamente nas seções seguintes.

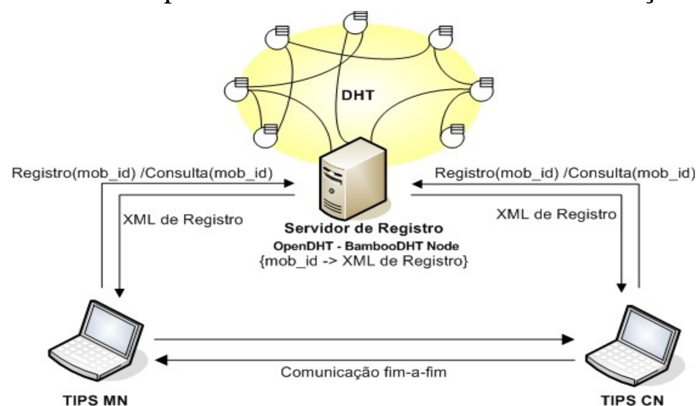

Figura 1: Comunicação através da arquitetura TIPS

3.1. Identificação Única do Terminal/Aplicação Móvel

Para desassociar as informações de localização e identificação, unificadas no endereçamento IP, um espaço de nomes sensível à aplicação é introduzido para a identificação única dos terminais móveis. Essa identificação corresponde a uma seqüência *hash* de 160 bits gerada através do algoritmo SHA-1 (*Secure Hash Algorithm*) sobre uma chave pública de um par de chaves assimétrico pública/privada pertencente ao terminal móvel. Essa abordagem é semelhante ao protocolo HIP [Moskowitz e Nicander 2006]. No entanto, o mecanismo de identificação proposto pode ser utilizado tanto para definir um *host* quanto uma aplicação, ambos sob um índice em uma *Tabela Hash Distribuída* que armazena a localização corrente do terminal móvel em um ***Servidor de Registro***.

Tendo em vista a transparência à lógica das aplicações, a estrutura *sockaddr_in* [Leffler et al. 1989] foi redefinida para *sockaddr_mob*, a qual contém o campo *smob_id* para representar a identificação única proposta (Figura 2). A inclusão dessa nova estrutura atua como um *cast* para a estrutura *sockaddr_in*. Desta forma, as primitivas de comunicação nativas da API Berkeley Sockets não são afetadas com a alteração.

```
1  struct sockaddr_mob {
2    sa_family_t sin_family;
3    in_port_t sin_port;
4    struct in_addr sin_addr;
5    u_char smob_id [SHA_DIGEST_LENGTH];
6  }; /* SHA DIGEST LENGTH = 20 bytes = 160 bits */
```

Figura 2. Estrutura *sockaddr_mob*

Inicialmente, a identificação do terminal é gerada a partir da extração da chave pública do terminal móvel armazenada em um arquivo de chaves RSA. A chave privada também é extraída do mesmo arquivo e utilizada posteriormente nos procedimentos de registro e atualização de localização no *Servidor de Registro*. Uma vez extraídas as chaves, então, são geradas as seqüências *hash* de cada uma das chaves através do algoritmo SHA-1 fornecido pela API de desenvolvimento OpenSSL [Cox et al. 2007].

```
1  int m_init(const char *rsa_fname, const char *dht_serv_loc, int dht_serv_port ){
2    ...
3    key *public_key = rsa_getpublickey(rsa_fname);
4    key *private_key = rsa_getprivatekey(rsa_fname);
5    SHA1(public_key->value, ...,my_mhost->host_id);
6    SHA1(private_key->value, ..., my_secret);
7    my_config->registry_location = dht_serv_loc;
8    my_config->registry_port = dht_serv_port;
9    pthread_create(..., m_periodic_registration, ...);
10   return (OK);
11 }
```

Figura 3. Primitiva *m_init*, que gera a identificação do terminal/aplicação móvel

Na Figura 3 são apresentadas partes da primitiva *m_init* responsável por gerar a identificação do terminal móvel a partir do *path* do arquivo de chaves RSA *rsa_fname*, bem como iniciar o registro periódico no *Servidor de Registro* através da *thread m_periodic_registration*. Além de ser responsável por realizar o primeiro registro no servidor, a primitiva *m_init* deve estar explícita no código das aplicações para ativar os módulos descritos.

Uma vez que os dados armazenados na DHT possuem um tempo de expiração (TTL), a atualização das informações de localização dos nós móveis além de ser realizada após a detecção de perda de conectividade no nó local também é realizada periodicamente (*m_periodic_registration*) de acordo com parâmetros de tempo definidos na aplicação. Isto evita registros desnecessários quando há muitas transições.

A identificação da aplicação *servidora* é gerada da mesma forma descrita anteriormente, desta vez através da primitiva *u_char *m_gethostid(char *certifilepath)*, sobre um certificado X509 auto-assinado pertencente ao terminal móvel da aplicação servidora. Desta forma, para iniciar as transmissões fim-a-fim o terminal cliente deve possuir o certificado X509 do terminal servidor.

Através do mecanismo de identificação única proposto para atuar de forma sensível à aplicação, as transmissões fim-a-fim passam a ser baseadas nos identificadores ao invés de endereços IP ou FQDN sem o intermédio de uma infra-estrutura de DNS, por exemplo. Isto possibilita a independência de localização e o tratamento da perda de conexão e recomposição da comunicação à posteriori sob um identificador que denota a atual localização do terminal móvel.

3.2. Registro, Consulta e Atualização de Localização

O uso de um serviço centralizado para descoberta dos nós é comumente utilizado no âmbito da Internet, e.g., aplicações P2P, jogos *multiplayers*, etc. Nesse sentido, a seleção em TIPS do uso de uma biblioteca aberta aderente à tecnologia de DHTs para

tanto, como o OpenDHT [Rhea et al. 2005], visa favorecer a escalabilidade dos procedimentos de registro, consulta e atualização de localização dos nós móveis. Desta forma, o *Servidor de Registro* implementa um nó BambooDHT [Rhea et al. 2004] que fornece uma *Tabela Hash Distribuída* (DHT) de propósito geral publicamente acessível e comprovadamente escalável através da infra-estrutura OpenDHT [Rhea et al. 2005]. De natureza *opensource*, o OpenDHT possui uma interface simples baseada em funções do tipo *put-get-remove* implementada sobre as arquiteturas *Sun RPC* e *XML RPC*. Deste modo, não é necessário que os clientes implementem um nó DHT ou mesmo possuam credenciais ou contas para utilizarem o serviço de armazenamento de dados na tabela. Essas características foram decisivas na escolha do OpenDHT como mecanismo de armazenamento do Registro na arquitetura proposta.

A Figura 4 apresenta um exemplo da estrutura XML de registro dos terminais móveis armazenada em um *Servidor de Registro*.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <MOBILITY_INFO time_reg="Wed Aug 28 04:14:49 2007">
3  <HOST_NAME>kimura</HOST_NAME>
4  <CURRENT_LOCATION>
5  <APP_ADDR>192.0.2.2</APP_ADDR>
6  </CURRENT_LOCATION>
7  </MOBILITY_INFO>
```

Figura 4. Estrutura XML de Registro de Localização

O procedimento de registro e atualização de localização dos terminais móveis é feito através da primitiva *m_registration_request*, como apresentado na Figura 5. Para executar essa tarefa a primitiva inicialmente abre uma conexão *Sun RPC* com um *Servidor de Registro* (linha 3), remove o registro anterior armazenado na DHT (linha 4) através da variável *my_secret* (seqüência *hash* gerada sobre a chave privada do terminal móvel - Figura 3, linha 6), cria a estrutura XML de registro (linha 5), e então armazena um novo registro na DHT por meio da primitiva *put* (linha 6) junto à identificação do terminal/aplicação móvel *mob_id*.

```
1  int m_registration_request (u_char *mob_id, char *reg_loc, int reg_port) {
2    ...
3    registry_fd = connect_dhtserver(reg_loc,reg_port));
4    rm_all(registry_fd, host_id, my_secret);
5    xml_val = m_create_xmlDTD();
6    put(registry_fd, mob_id, xml_val,...);
7    ...
8    return (OK);
9  }
```

Figura 5. Primitiva *m_registration_request*, que faz o registro no Servidor de Registro

Contudo, os terminais móveis podem estar localizados em redes que implementam o protocolo de tradução de endereçamento NAT ou NAPT impossibilitando a comunicação fim-a-fim. Para contornar este problema o campo *<APP_ADDR>* (Figura 4) da estrutura XML de registro é preenchido pelo próprio *Servidor de Registro* ao invés do terminal móvel. Isto permite identificar a origem do pacote, tal qual retornada na estrutura *sockaddr_in* preenchida na conexão ou nas transmissões com UDP, e o retorno da comunicação ao nó intermediário apropriado. Isso não seria possível utilizando um endereço IP privado fornecido pela origem sobre sua localização em uma rede interna. Contudo, quando o terminal servidor, passivo no estabelecimento da conexão, estiver localizado em uma rede privada é necessário que *middleboxes*, como *firewalls*, permitam redirecionar pacotes à rede interna, e.g., através de regras de encaminhamento baseadas em portas de comunicação (*port forwarding*).

Através desta abordagem as informações de localização dos terminais móveis sempre serão representadas por endereços roteáveis na Internet (i.e., endereços dos *gateways* NAT) e não mais endereços de redes privadas. Para tanto, foi necessário a

criação da classe *MobileHostLocation* e a alteração do método de inserção *put_args_to_put_req* da classe *PutReq* junto ao código Java do nó *gateway* BambooDHT, como mostra a Figura 6. Na linha 4 é atribuída à *string xml* a estrutura XML de registro armazenada na variável de parâmetro *put_args*. Na linha 5, a *string xml* é submetida ao método *setXmlMobileHostInfo* da classe adicionada *MobileHostLocation* para o preenchimento do campo *<APP_ADDR>* com a informação de localização armazenada no objeto *client* da classe *InetAddress* obtido do parâmetro do método *put_args_to_put_req* (linha 1). Essa informação de localização representa o endereço IP de origem da requisição, ou seja, endereço IP do *gateway* NAT.

```
1 protected Dht.PutReq put_args_to_put_req (bamboo_put_arguments put_args, InetAddress client){
2   ...
3   MobileHostLocation mhl = new MobileHostLocation();
4   String xml = mhl.setByteValueToString(ByteBuffer.wrap(put_args.value.value));
5   mhl.setXmlMobileHostInfo(xml, client);
6   return new Dht.PutReq(..., mhl.xmlMobileHostInfo.getBytes(), ...);
7 }
```

Figura 6. Alteração realizada no código *Java* da ferramenta *BambooDHT* para contornar a existência de *NAT* nas redes privadas

As alterações realizadas não influenciam nos procedimentos internos de inserção na DHT, apenas incluem a informação de localização do terminal móvel na estrutura XML de registro e submetem o dado alterado em forma de bytes para ser inserido normalmente na tabela (linha 6).

```
1 struct hostent* m_gethostbyname(const char *mob_id){
2   struct hostent *hst = NULL; ...
3   registry_fd = connect_to_dht_server(my_config->registry_location, my_config->registry_port);
4   get_result = get(clt_brocker_fd, host_id, ...)
5   m_parsexmltohostent(get_result->...bamboo_value_val, hst);
6   return (hst);
7 }
```

Figura 7. Primitiva *m_gethostbyname*, responsável por obter a localização atual do terminal móvel através da consulta ao Servidor de Registro

As consultas são realizadas através da primitiva *m_gethostbyname* (Figura 7). Semelhante à primitiva *gethostbyname* [Leffler et al. 1989], esta primitiva também retorna a estrutura *hostent* que denota a localização do *host,* contudo, baseada em sua identificação ao invés do nome DNS. Assim como na primitiva de registro *m_registration_request*, inicialmente é aberta uma conexão com o endereço IP (ou FQDN) e porta do *Servidor de Registro* (linha 3) obtidas da variável interna de escopo global *my_config* preenchida previamente na primitiva *m_init* (Figura 3, linhas 7 e 8). Então, através da primitiva *get* (linha 4) é obtido o dado armazenado no *Servidor de Registro* baseado na identificação *mob_id*. Este dado é submetido à função *m_parsexmltohostent* (linha 5) que extrai as informações armazenadas na estrutura XML e as aloca na estrutura *hostent hst* que posteriormente é devolvida à aplicação (linha 6).

3.3. Estabelecimento de Conexões

Previamente ao estabelecimento das conexões, a aplicação móvel cliente faz uma consulta *m_gethostbyname(serv_id)* (Figura 7) para resolver a identificação da aplicação servidora em sua respectiva localização corrente.

Conhecidos os endereços IP que representam a atual localização dos terminais móveis, é então estabelecida a conexão entre a aplicação cliente e a servidora através das primitivas *m_connect()* e *m_accept()*, respectivamente.

No lado do cliente, a primitiva *m_connect* conecta à aplicação servidora e envia a última seqüência de bytes recebidos (*recv_seq*) por sua aplicação. Por outro lado, a

primitiva *m_accept* na aplicação servidora recebe essa informação e envia de volta ao cliente também seu *recv_seq*. Através deste formato, em ambos os lados da comunicação as aplicações saberão em qual estado estão suas transmissões. Essas informações são utilizadas quando ocorre uma falha na conexão, de forma que os dados perdidos durante o deslocamento e que não foram consumidos pelas aplicações sejam retransmitidos após a reconexão, como mostra a Figura 8 linha 9. Esses dados são recuperados a partir de um *Buffer Circular* implementado no Espaço de Usuário que armazena uma cópia dos dados escritos no *socket*. Mais detalhes são apresentados na seção 3.5.

3.4. Detecção de perda de conexão e Reconexão após o deslocamento

A perda de conexão é detectada como uma falha na comunicação, provocada, neste caso, pela alteração ou obtenção de um novo endereço IP. Essa detecção é realizada internamente às primitivas de envio e recebimento de dados adaptadas para o contexto da Mobilidade IP, tornando o tratamento da perda de conexão transparente à lógica das aplicações. Para tanto, o *socket* também é ajustado previamente em sua criação através da primitiva *m_socket*. Essa primitiva é responsável por criar o descritor de arquivo e ativar o mecanismo de *KeepAlive* em comunicações orientadas à conexão, bem como ajustar o intervalo de emissão dos *probes* através da utilização da primitiva *setsockopt* [Leffler et al. 1989] sobre o descritor de arquivo do *socket* criado. Desta forma, parâmetros de configuração permitem identificar o período máximo de tentativa de reconexão sobre o *socket* permitindo o tratamento da perda de conexão por longos períodos de tempo, como nas DTNs (*Delay/Disruption Tolerant Networks*).

```
1  ssize_t m_write(int fd, const void *buf, size_t count){
2   ...
3   flags = fcntl(fd, F_GETFL, 0); fcntl(fd, F_SETFL, O_WRONLY | O_NONBLOCK);
4   if ((n = write(fd, buf, count)) == FAILURE){
5     if (is_m_error(errno){
6       m_registration_request(host_id, registry_location , registry_port)
7       if (is_server) new_sock_fd = m_server_accepting();
8       else new_sock_fd = m_client_connecting();
9       m_copy_us_sk_buffer(fd, new_sock_fd);
10      close(fd); dup2(new_sock_fd, fd); close(new_sock_fd);
11      fcntl(fd, F_SETFL, flags);
12      return (m_write(fd, buf, count));
13    }
14  }
15  fcntl(fd, F_SETFL, flags);
16  if (app_transf){
17    us_sk_buffer_write(&us_buffers[fd%NUM_BUFFERS],buf,n);
18    us_buffers[fd%NUM_BUFFERS].sk_byte_seq.send_seq += n;
19  }
20  return n;
21 }
```

Figura 8. Detecção de perda de Conexão, recomposição da comunicação e reenvio dos dados perdidos durante o deslocamento através da primitiva *m_write*

A Figura 8 apresenta o funcionamento interno da primitiva *m_write* para detecção de perda de conexão, recomposição da comunicação e reenvio dos dados perdidos durante o deslocamento. Inicialmente, o descritor de arquivo *fd* é configurado para operar em modo não bloqueante, como mostra a linha 3. Isto possibilita a detecção imediata da possível falha na conexão (e.g., ETIMEOUT, EPIPE, ECONNRESET, EHOSTUNREACH) sobre a variável de retorno *errno* através da primitiva *is_m_error* (linha 5) quando o terminal móvel é deslocado para uma nova rede e obtém um novo endereço IP. Quando este evento ocorre, um novo registro é submetido a um *Servidor de Registro* através da primitiva *m_registration_request* (linha 6) e uma estrutura condicional define o tratamento adequado da perda de conexão para os casos em que a aplicação é servidora ou cliente (linhas 7 e 8). Deste modo a aplicação servidora volta ao estado de *accept* através da primitiva

m_server_accepting (linha 7) para aceitar uma nova conexão do cliente utilizando a primitiva *m_accept*.

Por outro lado, a aplicação cliente através da primitiva *m_client_connecting* cria um novo descritor de *socket* por meio da primitiva *m_socket*, consulta a localização atual da aplicação servidora através da primitiva *m_gethostbyname* e posteriormente tenta uma nova conexão (*m_connect*) baseada na consulta submetida. Contudo, caso a aplicação servidora altere sua localização durante esse período o erro EHOSTUNREACH (*host* inacessível dado um endereço IP) é tratado na aplicação cliente e os passos anteriores são repetidos até que a conexão obtenha sucesso.

Uma vez retomada a conexão, a aplicação emissora recupera os dados perdidos durante o deslocamento e os retransmite através da primitiva *m_copy_us_sk_buffer* (linha 9). Para tanto, uma estratégia de *Buffer Circular* implementado no Espaço de Usuário armazena somente os dados enviados pela aplicação e, quando ocorre o deslocamento, os dados perdidos são recuperados e retransmitidos a partir desse *buffer* auxiliar. Esse mecanismo é explicado detalhadamente na próxima seção. Posteriormente, o descritor corrompido *fd* é sobreposto pelo descritor *new_sock_fd* utilizando a primitiva *dup2* [Leffler et al. 1989], como mostra a linha 10. O novo descritor *new_sock_fd* é fechado (linha 10) e os *flags* do descritor *fd* configurados previamente ao tratamento da recomposição da comunicação são restaurados (linha 11). Após a recomposição da comunicação e a restauração do estado da transmissão, então, os dados do *buffer* da aplicação *buf* são enviados através do retorno recursivo da primitiva *m_write* (linha 12). Em condições normais de transmissão, os dados enviados são armazenados no *buffer auxiliar* (linha 17) e contabilizados na variável *send_seq* (linha 18) para atualizar o estado da transmissão. Para manter a consistência do estado das transmissões a variável booleana de escopo global *app_transf* (linha 16) controla o armazenamento no *buffer* auxiliar. Desta forma, somente os dados vindos do *buffer* da aplicação são armazenados e contabilizados, desconsiderando o re-armazenamento da retransmissão dos dados perdidos durante o deslocamento realizada pela primitiva *m_copy_us_sk_buffer*.

A primitiva de recebimento *m_read* utiliza os mesmos mecanismos de tratamento de perda de conexão, contudo, não existe a cópia dos dados recebidos no *buffer circular*. A variável *recv_seq* apenas contabiliza os bytes recebidos para atualizar o estado da transmissão na aplicação receptora. Esse mesmo tratamento é utilizado nas outras primitivas de recebimento *m_recvfrom* e *m_recv*. Por outro lado, as primitivas de envio *m_sendto* e *m_send* utilizam a mesma estratégia definida na primitiva *m_write*.

3.5. Deslocamento sem perda de dados

Por questões de desempenho, inicialmente a tentativa foi retomar os dados que não foram confirmados diretamente do *kernel socket buffer* de envio do *socket* corrompido da aplicação emissora e retransmiti-los após a reconexão. No entanto, quando ocorre uma falha no *socket* tanto os dados armazenados no *buffer* de envio que não foram confirmados pela Camada de Transporte (TCP) quanto os dados armazenados no *buffer* de recebimento que não foram consumidos pela aplicação são perdidos de imediato.

A estratégia adotada para resolver este problema consiste na criação de um *buffer circular* no Espaço de Usuário em que somente os dados enviados (escritos no *socket*) são armazenados neste *buffer*. No entanto, não há confirmação dos dados recebidos (envio de ACKs) pelas aplicações durante a comunicação. Isto é possível,

pois só são recuperados e retransmitidos dados do *buffer* quando ocorre uma reconexão, em que ocorre também a troca da seqüência do último byte recebido (*recv_seq*) por cada aplicação. Desta forma, são recuperados do *buffer auxiliar* da aplicação emissora e retransmitidos à aplicação receptora os dados que estão entre a seqüência de recebimento *recv_seq* do receptor e a seqüência de envio *send_seq* do emissor, como mostra a Figura 9 linha 3.

```
1  size_t m_copy_us_sk_buffer(int oldfd, int newfd){
2    ...
3    n = lost_bytes = us_buffers[oldfd].sk_bs.send_seq - us_buffers[newfd].rt_bs.recv_seq;
4    if (lost_bytes > 0){
5      us_sk_buffer_read(&us_buffers[oldfd], us_buffers[newfd].rt_bs.recv_seq, lost_data);
6      ptr = lost_data; nw = 0;
7      app_transf = FALSE;
8      while ((nw < lost_data) && (nw != -1)){
9        lost_data -= nw; ptr += nw;
10       nw = m_write(new_sockfd, ptr, lost_data);
11     }
12     app_transf = TRUE;
13     free(lost_data);
14     us_sk_buffer_free(&us_buffers[newfd]);
15   }
16   return n;
17 }
```

Figura 9. Primitiva *m_copy_us_sk_buffer*, responsável por recuperar do *buffer* os dados perdidos no deslocamento e retransmití-los para a nova localização do terminal

Contudo, no momento da retransmissão pode ocorrer que uma das partes, ou mesmo ambas as partes se desloquem causando novamente a perda de conexão. Para evitar a perda de dados sobre a retransmissão, os dados recuperados (*lost_data* - linha 5) são retransmitidos utilizando a primitiva *m_write* (linha 10) provendo o suporte à mobilidade mesmo nesse instante. A variável *app_tranf* (linhas 7 e 12) controla a cópia dos dados enviados no *buffer auxiliar* para que o estado da transmissão não seja corrompido pelo incremento dos bytes retransmitidos. Como o *buffer* auxiliar criado para o novo descritor *newfd* não é utilizado em qualquer outra parte do código, ele então é liberado da memória (linha 14).

Uma preocupação na proposição de um mecanismo de *buffer circular* é a definição do seu tamanho. Em *buffers* pequenos (menores que 64 KBytes) existe a possibilidade do fim do dado armazenado sobrepor seu início perdendo a referência da seqüência correta dos bytes. Para obter um nível seguro de consistência dos dados no armazenamento, o ideal é que o tamanho do *buffer* seja 2 vezes maior que o tamanho do *window size* do *TCP*, ou seja, 128 KBytes. No caso comum, os bytes não confirmados no *kernel socket buffer* de envio somados aos bytes não consumidos do *kernel socket buffer* de recebimento não ultrapassam o tamanho máximo do *buffer* auxiliar, impossibilitando que ocorra sobreposição nos dados.

Com a utilização dessa estratégia de *buffer circular* no Espaço de Usuário para preservar os dados transmitidos é possível garantir que os dados chegarão até à Camada de Aplicação. Além disso, do ponto de vista da transmissão de dados a migração do *socket* é naturalmente suportada, sem a utilização de um esquema de triangulação como proposto no mecanismo SockMi [Bernaschi et al. 2007].

4. Resultados Obtidos

O uso de uma ferramenta escalável, OpenDHT [Rhea et al. 2005], como *Servidor de Registro* nos procedimentos de armazenamento, consulta e atualização de localização, fez com que as avaliações realizadas com o uso da solução de mobilidade desenvolvida fosse concentrada na medição dos tempos envolvidos. Desta forma, três análises foram realizadas:

1) Latência do Procedimento de *Handoff horizontal*: a latência do período de

handoff entre redes 802.11g.

2) Latência do Procedimento de *Handoff Vertical*: a latência do período de *handoff* entre redes 802.11 e 802.15.1 (*bluetooth* - classe 1).
3) Comportamento do *Buffer Circular*: a quantidade média de bytes recuperados do *buffer* auxiliar e retransmitidos após a reconexão.

Na análise de *Latência de Handoff*, a latência representa o intervalo de tempo entre a recepção do dado de seqüência *n* pela aplicação na antiga localização e a seqüência de byte *n + 1* na nova localização. Desta forma, o cálculo é representado por:

$$L_C = t_{det} + t_{reg} + t_{lookup} + t_{rec} + t_{ret} \quad (1)$$

$$L_S = t_{det} + t_{reg} + t_{acc} + t_{ret} \quad (2)$$

Onde L_C e L_S representam a latência de *handoff* da aplicação cliente e servidora, respectivamente; t_{det} é o tempo de detecção da perda de conexão; t_{reg} é o tempo de registro (*m_registration_request*) no *Servidor de Registro*; t_{lookup} é o tempo de consulta (*m_gethostbyname*) no *Servidor de Registro*; t_{rec} é o tempo da reconexão (*m_connect*) após o deslocamento das aplicações fim-a-fim; t_{ret} é o tempo de recuperação dos dados perdidos a partir do *buffer* auxiliar durante o *handoff* e retransmitidos após a reconexão; e t_{acc} é o tempo total de espera por conexão (*m_accept*);

4.1. Latência do Procedimento de *Handoff Horizontal*

Para analisar a latência de *Handoff* horizontal foram submetidas 100 alterações manuais de endereçamento durante a transferência de um arquivo suficientemente grande (550 MB) para dois casos de testes: *a)* cliente móvel transferindo dados para aplicação servidora e *b)* e servidor móvel transferindo dados para aplicação cliente. Em ambos os casos foram considerados a mobilidade de ambas as partes da comunicação e a transmissão contínua de dados sem interrupção pelas aplicações. Os testes foram submetidos em uma rede sem fio *802.11g* com latência média de transmissão de dados de *1,59 ms* e taxa média de transferência de *22,16 Mbps*. A Tabela 1 apresenta os resultados obtidos da média dos tempos descritos nas formulas (1) e (2) .

Tabela 1: Média dos tempos (segundos) obtidos a partir das fórmulas (1) e (2) para os casos de testes *a)* e *b)* na análise da Latência do procedimento de *Handoff Horizontal*

Aplicação	t_{det}	t_{reg}	t_{lookup}	t_{rec}	t_{ret}	t_{acc}	*Latência*	*Desvio padrão*
Cliente	0,055370	0,228262	0,081214	1,510723	0,019856	---	**1,895424**	**0,263818**
Servidor	0,055724	0,340426	---	---	0,017464	1,071234	**1,484848**	**0,367677**

Na análise destes estudos de caso, foi observado que o tempo de latência de *handoff* da aplicação cliente (média de *1,8954 s*) foi em média *0,41 s* maior que a latência de *handoff* da aplicação servidora (média de *1,4848 s*) para amostra submetida. Esta diferença no tempo é decorrente da quantidade de operações necessárias para o *handoff* na aplicação cliente, em que após o deslocamento é necessária uma consulta no *Servidor de Registro* (t_{lookup}) sobre a identificação do terminal da aplicação servidora para restabelecer a conexão corrompida, representado uma operação a mais que no procedimento de *handoff* da aplicação servidora. Essa consulta é imprescindível quando existe mobilidade na aplicação servidora, uma vez que sem ela o cliente *a priori* não tem conhecimento da nova localização da aplicação servidora e, além disso, existe a possibilidade de ocorrer o deslocamento de ambas as

partes simultaneamente.

Por outro lado, se a aplicação servidora não for móvel e aceitar conexões em modo INADDR_ANY então não existe a necessidade do registro, pois somente o cliente irá se deslocar e basta a detecção da perda de conexão por ambas as partes para o restabelecimento da conexão da aplicação cliente com a localização fixa da aplicação servidora. Isto evita eventuais registros e consultas no ***Servidor de Registro*** reduzindo a latência de *handoff* principalmente para a aplicação móvel cliente. Desta forma, um serviço de *broker* para registro só faz sentido quando a aplicação servidora também for móvel.

Tendo em vista que as operações decorrentes em ambos os casos de deslocamento, os tempos de latência observados (L_C e L_S) podem ser considerados aceitáveis para a recomposição da comunicação. Ainda, deve-se considerar, sobretudo, as próprias características de transmissão das redes sem fio (ruído, degradação do sinal, etc.) em que naturalmente pacotes são perdidos e a taxa de erro (*Frame Error Rate* - FER) é relativamente alta comparada à das redes cabeadas.

O tempos observados referem-se à alteração realizada manualmente na interface de rede sem fio. Contudo, considerando a movimentação dos usuários móveis, o tempo de aquisição de um novo endereço IP (geralmente via DHCP) também deve ser avaliado. Este tempo foi medido na mesma rede de teste e o tempo médio de 100 operações para aquisição de endereço IP via DHCP (incluindo as operações de DHCPDISCOVER, DHCPREQUEST e DHCPACK) foi de *3,9917 s* com desvio padrão de *1,32 s*.

4.2. Latência do Procedimento de *Handoff Vertical*

Semelhante à análise anterior, para medir a Latência do procedimento de *Handoff Vertical* foram submetidas também 100 alterações manuais de endereçamento IP, contudo, entre uma rede 802.11g (*Wi-fi*) e outra 802.15.1 (*Bluetooth*) com a aplicação cliente transferindo o mesmo arquivo (550 MB) para aplicação servidora.

A aplicação servidora foi preparada para receber conexões de qualquer origem através da definição INADDR_ANY na operação de *binding*. Desta forma, não foi necessário o procedimento de registro da aplicação cliente. Foram observados os tempos descritos na fórmula (1), exceto o parâmetro t_{reg}. Os resultados são apresentados na Tabela 2.

A rede de teste foi configurada da seguinte forma (Figura 11): um *gateway* atuando como Ponto de Acesso 802.11 b/g e 802.3 (*Ethernet*); um terminal configurado como Ponto de Acesso *Bluetooth* (*Network Access Point – **NAP Bluetooth***) e ***Servidor de Registro*** conectado ao *gateway* via cabo; um terminal estacionário **TIPS CN** (*Correspondent Node* que implementou a aplicação servidora) e um terminal móvel **TIPS MN** (*Mobile Node* que implementou a aplicação cliente). A rede *Bluetooth* corresponde à rede 10.0.0.0 e o *NAP Bluetooth* é identificado com o IP 10.0.0.1. O *Servidor de Registro* é acessível pela rede 192.168.0.0 através do IP 192.168.0.5. O *gateway* é identificado pelo IP 192.168.0.1 e o CN por 192.168.0.6. O terminal móvel MN é acessível com o IP 10.0.0.2 quando na rede *Bluetooth*, e com IP 192.168.0.7 quando na rede 802.11g, como mostra a Figura 11.

Utilizando regras de IPTABLES no *NAP Bluetooth* e adicionando rotas para ambas as redes nos terminais MN e CN foi possível realizar a comunicação fim-a-fim entre os terminais. Desta forma, no *NAP Bluetooth*, pacotes com origem da 10.0.0.0 sobre a interface *Bluetooth* foram encaminhados para rede 192.168.0.0 sobre a

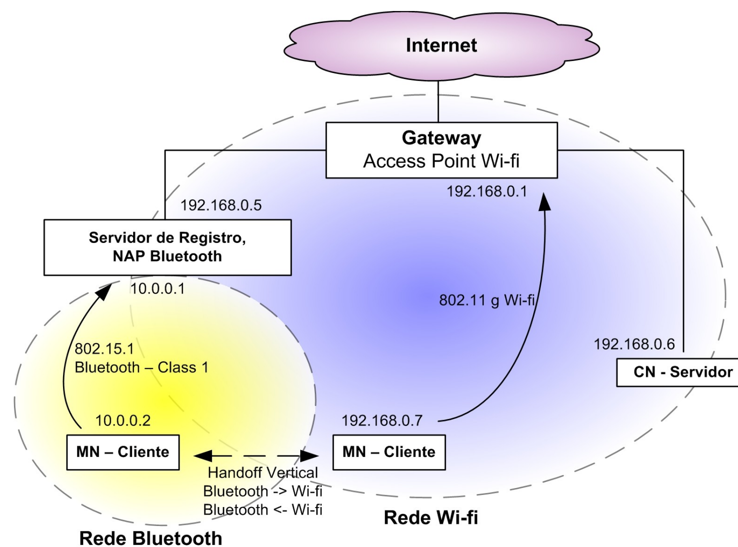

Figura 11: Rede de testes utilizada para análise de *Handoff Vertical*

interface *Ethernet* (cuja a rota padrão foi *gateway* 192.168.0.1), e pacotes destinados a rede 10.0.0.0 com origem da rede 192.168.0.0 foram encaminhados da interface *Ethernet* para a interface *Bluetooth*. Rotas foram adicionadas nos terminais MN e CN para que ambas as redes se tornassem acessíveis. No nó CN a rota padrão foi o *gateway* 192.168.0.1, contudo, os pacotes destinados à rede 10.0.0.0 utilizam a rota para o *NAP Bluetooth* 192.168.0.5 como *gateway* para os pacotes destinados a esta rede. No nó MN, o *gateway default* quando na rede *Bluetooth* foi o NAP com IP 10.0.0.1 e quando na rede *Wi-fi* o *Gateway Access Point* 192.168.0.1 foi a rota padrão.

Se comparados com a Tabela 1 os tempos apresentados na Tabela 2 para as transições entre as redes são maiores, uma vez que média da latência da rede *Bluetooth* foi de *25,42 ms* com taxa de transmissão de *563,36 Kbps*, enquanto a rede 802.11 obteve média da latência de rede de *1,59 ms* com taxa de transmissão de *22,16 Mbps*. A média da latência de *handoff* de *4,52 s* na transição da rede *Wi-fi* para a rede *Bluetooth* foi maior que o dobro da média da latência da transição inversa (*2,12 s*). Sobre os tempos observados, além da latência de rede e taxa de transmissão, deve se considerar algumas operações da pilha de protocolos para transmissões em *Bluetooth*, como por exemplo, o protocolo SDP (*Service Discovery Protocol*) para descoberta de serviços e redes *Bluetooth* disponíveis no ambiente, e a inicialização das interfaces de redes pelo sistema operacional. Neste contexto, o tempos observados para transição da rede *Wi-fi* para rede *Bluetooth* naturalmente são maiores que a transição inversa.

Tabela 2: Média dos tempos (segundos) na análise da Latência do procedimento de *Handoff Vertical* entre redes *Bluetooth* e *Wi-fi*.

Handoff	t_{det}	t_{lookup}	t_{rec}	t_{ret}	*Latência*	*Desvio padrão*
Bluetooth -> Wi-fi	1,052284	0,892346	0,050377	0,132780	**2,127787**	**1,356254**
Wi-fi -> Bluetooth	1,052450	2,6005605	0,104418	0,766695	**4,524124**	**3,753704**

4.3. Análise do comportamento do *Buffer Circular*

Do ponto de vista da recuperação dos dados que não chegaram à aplicação, devido ao deslocamento, a partir do *buffer* auxiliar, pôde-se observar que a quantidade de bytes retransmitidos segue um comportamento linear crescente de acordo com a variação do tamanho dos *buffers* das aplicações, como mostra a Figura 12. Para esta análise, o tamanho do *buffer circular* foi fixado em 128 KB e observado nos seguintes casos: *1)*

o emissor e receptor escrevem e lêem a mesma quantidade de bytes do *socket*; *2)* o receptor lê do *socket* uma quantidade de bytes duas vezes maior do que o emissor escreve; e *3)* o emissor escreve uma quantidade de bytes duas vezes maior do que o receptor lê do socket. Os tamanhos dos *buffers* das aplicações foram alterados em cada um destes testes seguindo uma seqüência exponencial de tamanho, iniciando com 1KB e chegando ao tamanho máximo de escrita no *socket* de 64KB. Desta forma, no caso *2)* os tamanhos do *buffer* da aplicação receptora nas transmissões foram de 2, 4, 8, 16, 32 e 64KB em cada teste, enquanto na aplicação emissora os tamanhos foram de 1, 2, 4, 8, 16 e 32 KB respectivamente. No caso *3)* a análise foi inversa ao caso *2)*, em que os tamanhos para o *buffer* da aplicação emissora foram o dobro da aplicação receptora.

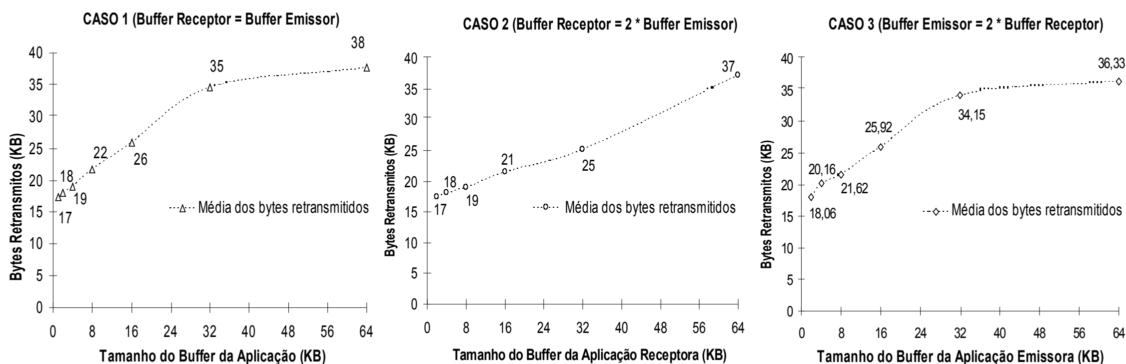

Figura 12: Gráfico da Média do tamanho dos blocos de dados recuperados do *Buffer Circular* e retransmitidos após a reconexão para os casos de testes *1), 2)* e *3)*

A quantidade média de bytes retransmitidos teve variação relativa entre cada um dos três testes. As médias variaram entre 17 e 18 KB para *buffers* da aplicação entre 1 e 2KB; de 18 a 22 KB para *buffers* da aplicação entre 4 e 8 KB; de 21 a 35KB para *buffers* da aplicação entre 16 e 32KB; e de 25 a 38KB para *buffers* da aplicação entre 32 e 64KB. Os dados retransmitidos representam os dados armazenados no *buffer* de envio do *socket* da aplicação emissora que não foram consumidos no *socket* destino pela aplicação receptora. A quantidade de bytes retransmitidos teve um comportamento crescente, de acordo com os tamanhos dos *buffers* das aplicações, não sofrendo variações drásticas quando o emissor escreve uma quantidade de dados maior no *socket* do que o receptor lê, ou vice-versa.

5. Conclusões

TIPS combina novas propostas com soluções conhecidas para os problemas envolvidos com a mobilidade IP, preservando a lógica das aplicações e reduzindo o custo de implantação. Na arquitetura proposta também é evitada a sobrecarga das outras soluções (como o MIP [Perkins 2002] e HIP [Moskowitz e Nicander 2006]) na troca de dados das aplicações, uma vez que a pilha de protocolos não é alterada e não existe formatação das mensagens. Desta forma, aplicações se comunicam sem precisar tratar explicitamente a mobilidade IP e, ainda, com a garantia de transmissão acima da Camada de Transporte. Além disso, um nível de segurança é implícito na utilização desta solução. Para o acesso ao OpenDHT cada terminal móvel utiliza seu par de seqüências *hash* ***mob_id*** (para consulta) e ***secret_key*** (para alteração) gerados sobre seu par de chaves Pública/Privada. Desta forma, os registros armazenados na tabela *hash* só podem ser removidos através do ***secrect_key*** exclusivo à cada terminal, não existindo a possibilidade de um terminal corromper os dados armazenados no ***Servidor de Registro*** de outro. Do ponto de vista das transmissões, a segurança pode ser obtida com o uso de sessões SSL para encriptação fim-a-fim, utilizando as chaves

públicas disponíveis.

Embora não seja possível detectar quebras na comunicação em ambos os lados sobre o suporte às comunicações com UDP, o mecanismo de atualização dinâmica do registro de localização permite viabilizar transmissões futuras para um nó que se moveu em transmissões não orientadas à conexão. Isso é tratado com resoluções periódicas do endereço do destino e, no caso mais crítico, uma resolução antes de cada nova transmissão.

A garantia da transmissão de dados através da estrategia de *buffer circular* no espaço de usuário, capaz de salvar e restaurar os estados das transmissões, é um instrumento que está sendo considerado para introduzir o suporte à Mobilidade de Sessão à ferramenta. Também a criação de um *wrapper* entre o espaço de usuário e kernel para promover a total transparência às aplicações e ao próprio desenvolvedor está sendo investigado. A implementação atual do TIPS suporta a Mobilidade na versão IPv4. Seguindo a mesma estratégia de implementação e modelo, a idéia é oferecer o suporte também à versão IPv6.

Referências

Bernaschi, M., Casadei, F. and Tassotti, P. (2007) "SockMi: a solution for migrating TCP/IP connections", in IEEE PDP.

Cox, M. J., Engelschall, R. S., Henson, S. and Laurie, B. (2007) "OpenSSL", http://www.openssl.org, Novembro.

Kimura, B. Y. L. and Guardia, H. C. (2008) "TIPS: Wrapping the Sockets API for Seamless IP Mobility", in the 23th Annual ACM Symposium On Applied Computing 2008 (ACM SAC '08).

Lee, C. P., Keshav, A., Caballero, C., Feamster, N., Mihail, M. and Copeland, J. A. (2007) "MobCast: Overlay Architecture for Seamless IP Mobility using Scalable Anycast Proxies", in IEEE WCNC.

Leffler, S. J., McKusick, M. K., Karels, M. J. and Quarterman, J. S. (1989) "The Design and Implementation of the 4.3 BSD Unix Operating System", in Addison-Wesley, Reading, Mass.

Maltz D. A. and Bhagwat P. (1998) "MSOCKS: An Architecture for transport layer mobility", in IEEE Infocom.

Moskowitz, R. and Nicander, P. (2006) "Host Identity Protocol (HIP) Architecture", in IETF, RFC 4423.

Perkins, C. E. (2002) "IP Mobility Support for IPv4". IETF, RFC 3344.

Rhea, S., Geels, D., Roscoe, T. and Kubiatowicz, J. (2004) "Handling Churn in a DHT", in Proceedings of the USENIX Annual Technical Conference, June.

Rhea, S., Godfrey, B., Karp, B., Kubiatowicz, J., Ratnasamy, S., Shenker, S., Stoica, I. and Yu, H. (2005) "OpenDHT: A Public DHT Service and Its Uses", in ACM SIGCOMM.

Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M. and Schooler, R. (2002) "SIP: Session Initiation Protocol", in IETF, RFC 3261.

Zandy, V. C. and Miller, B., P. (2002) "Reliable Network Connections", in ACM MOBICOM.