Alan Castro Silva

Implementação Inicial da RFC 6897

Sorocaba, SP

06 de Dezembro de 2016

Alan Castro Silva

Implementação Inicial da RFC 6897

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação (PPGCCS) da Universidade Federal de São Carlos como parte dos requisitos exigidos para a obtenção do título de Mestre em Ciência da Computação. Área de concentração: Engenharia de Software e Redes de Computadores.

Universidade Federal de São Carlos – UFSCar

Centro de Ciências em Gestão e Tecnologia – CCGT

Programa de Pós-Graduação em Ciência da Computação – PPGCCS

Orientador: Prof. Dr. Fábio Luciano Verdi

Sorocaba, SP

06 de Dezembro de 2016

Castro Silva, Alan

Implementação Inicial da RFC 6897 / Alan Castro Silva. -- 2016.
71 f. : 30 cm.

Dissertação (mestrado)-Universidade Federal de São Carlos, campus Sorocaba, Sorocaba
Orientador: Prof. Dr. Fábio Luciano Verdi
Banca examinadora: Prof. Dr. Antônio Marcos Alberti, Prof. Dr. Gustavo Maciel Dias Vieira
Bibliografia

1. Protocolos de Transporte de Rede. 2. Interface de Programação para Aplicações em Redes. 3. MPTCP. I. Orientador. II. Universidade Federal de São Carlos. III. Título.

Ficha catalográfica elaborada pelo Programa de Geração Automática da Secretaria Geral de Informática (SIn).
DADOS FORNECIDOS PELO(A) AUTOR(A)

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências em Gestão e Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a Defesa de Dissertação de Mestrado do candidato Alan Castro Silva, realizada em 06/12/2016:

Prof. Dr. Fabio Luciano Verdi
UFSCar

Prof. Dr. Antônio Marcos Alberti
Inatel

Prof. Dr. Gustavo Maciel Dias Vieira
UFSCar

Aos meus pais Carlos e Neli.

Agradecimentos

Agradeço,

a Deus pela vida e pela oportunidade de realizar um sonho;

aos meus pais Carlos e Neli por terem sido sempre presentes e mesmo com todas as dificuldades enfrentadas, sempre me apoiarem na decisão de continuar a estudar;

aos amigos especiais, Fernando Caruso, Renato Alvarenga, Jorge Novaes e William Pascucci por me ouvirem e motivarem nas dificuldades;

aos companheiros de laboratório, em especial aos grandes amigos Marcus Sandri, Prof. Dr. Lucio Agostinho Rocha, Marcelo Frate e André Beltrami pelas dúvidas, sugestões, críticas e pela paciência quando eu quis dar palpite no trabalho deles;

ao amigo, Regis Martins pelo auxilio nas simulações e ambiente de testes que foi montado para atender esse trabalho;

ao pessoal da Beets Jr. em especial ao casal Felipe Novaes e Estela Melo, Enrique Sampaio, João Paulo e Marcelo Marczuk por terem um sonho e acreditarem na minha experiência para ajudar a transformar esse sonho na realidade que hoje é a Empresa Junior de Ciência da Computação;

a Simone Ferlin, do Simula Lab / University of Oslo pela orientação concedida, tratamento e pela paciência em mostrar o melhor caminho para que fosse possível o desenvolvimento desse trabalho;

e, por último, mas não menos importante, ao Prof. Fabio pela oportunidade, paciência e por ter acreditado no potêncial desse trabalho.

"We will pay the price
But we will not count the cost."
(RUSH - Bravado)

Resumo

O protocolo *Multipath TCP (MPTCP)* permite que as aplicações possam explorar melhor os recursos de rede disponíveis para dispositivos multiconectados como os telefones móveis ou sistemas *multi-homed*. Aqui, algumas vantagens são previstas: agregação de banda, a habilidade de manter a conexão estabelecida se houver falha em um dos caminhos de rede e a utilização de múltiplos caminhos. Para estender essas capacidades para a aplicação, a *RFC 6897* define uma *API* que permite um melhor controle de cada subfluxo *MPTCP*, de modo que esses possam ser adicionados ou removidos conforme necessário.

Este trabalho apresenta uma implementação inicial da *API* descrita na *RFC 6897* para o protocolo *MPTCP*. Sendo assim, implementamos algumas das funções de manipulação do protocolo *MPTCP* descritas no documento, quais sejam: ligar e desligar o protocolo, verificar subfluxos existentes e adicionar novos subfluxos. Para testar a *API* e validar a nossa implementação, nós desenvolvemos uma aplicação *HTTP* que detecta fluxos elefantes e utiliza a *API* para abrir novos subfluxos a partir da conexão *TCP* original. Testes de desempenho foram realizados em uma topologia cúbica e mostraram que a utilização da *API* pela aplicação diminuiu o *Flow Completion Time* das conexões *TCP*.

Palavras-chaves: Protocolos de Transporte de Rede; MPTCP; Interface de Programação para Aplicações em Redes; Algoritmos de Controle de Rede; Experimentação em Redes.

Abstract

The Multipath TCP (MPTCP) protocol allows applications to better explore the network resources available to multi-connected devices such as mobile phones or multi-homed systems. Here, some advantages are envisioned: bandwidth aggregation, the ability to maintain the connection, if one of the network path fails and the use of multiple paths. To extend these capabilities to the application, RFC 6897 defines an API to better control each of MPTCP's subflows, so that these can be added or removed as needed.

This work presents an initial API implementation as defined in RFC 6897. We implemented some functions described in the document, such as protocol on/off, check existent subflows and add new subflows. To test the API and validate our implementation we built an HTTP application that detects elephant flows and uses the API for open new subflows using the original TCP connection. Some tests were performed in a network using a cubic topology and showed that the API utilization decreased the Flow Completion time of TCP connections.

Keywords: Network Transport Protocol; MPTCP; Network Programming interfaces; Network Control Algorithms; Network Experimentation.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

IP	*Internet Protocol*
TCP	*Transmission Control Protocol*
UDP	*User Datagram Protocol*
SCTP	*Stream Control Transmission Protocol*
MPTCP	*Multipath TCP*
ECMP	*Equal Cost Multipath Protocol*
HTTP	*Hypertext Transfer Protocol*
API	*Application Programming Interface*
IEEE	*Institute of Electrical and Electronics Engineers*
IETF	*Internet Engineering Task Force*
POSIX	*Portable Operating System Interface*
RFC	*Request For Comments*
LTE	*Long Term Evolution*
NAT	*Network Address Translation*
MAC	*Media Access Control*
RTT	*Round-Trip Time*
WAN	*Wide-Area Network*
CMT	*Concurrent Multipath Transfer*
HMAC	*Hash-based Message Authentication Code*
FCT	*Flow Completion Time*

Sumário

Introdução

Atualmente, a maioria dos dispositivos *endhost* possuem múltiplas interfaces de acesso, permitindo a sua utilização de forma simultânea através de aplicações e redes que possam explorar esses recursos de forma eficiente. Esse tipo de mecanismo pode auxiliar o gerenciamento de tráfego em uma rede permitindo, por exemplo, o controle da utilização de banda e tornando mais eficiente a vazão dos fluxos. Para que esse recurso possa ser explorado é necessário que a camada responsável pelo transporte de dados tenha a capacidade de aproveitar essas interfaces para utilizar vários caminhos através de protocolos que possam suportar essa tarefa.

O desenvolvimento de protocolos que possuem a capacidade de utilização de múltiplos caminhos, permitindo com isso, a divisão de diversos fluxos através das interfaces presentes em um dispositivo ainda continua sendo um dos principais tópicos de pesquisa em protocolos de transporte. Isso ocorre porque cada vez mais surgem dispositivos e tecnologias que utilizam várias interfaces ao mesmo tempo.

Em relação a pesquisa de múltiplos caminhos em protocolos de transporte, podemos dividi-la em dois items:

- Desenvolvimento de novos protocolos que possuem uma pilha totalmente nova;
- Utilização e extensão de um protocolo já existente.

Em relação ao primeiro item, podemos citar o protocolo *SCTP* (STEVENS; FENNER; RUDOFF, 2004) que foi criado utilizando uma pilha própria e com a intenção de substituir os atuais protocolos existentes na camada de transporte *(TCP e UDP)* conseguindo trabalhar de forma nativa com a opção de transmissão de fluxo de dados entre as diversas interfaces disponíveis. Porém, seu principal problema acaba sendo uma quebra de paradigma, pois sendo um protocolo totalmente novo, a sua adoção é limitada, pois depende do desenvolvimento de aplicações que suportam esse protocolo.

Em relação ao segundo item, temos o *MPTCP* (PAASCH; BONAVENTURE, 2014) que é uma extensão ao protocolo *TCP* adicionando recursos que possibilitam a transferência de dados utilizando caminhos múltiplos. Com isso, o protocolo *TCP* consegue distribuir os dados de forma arbitrária dentro desses subfluxos criados. A utilização do protocolo *MPTCP* acaba sendo transparente e praticamente imediata, pois ele utiliza como base o protocolo *TCP*.

O *MPTCP* ganhou reconhecimento comercial e foi adaptado aos sistemas operacionais recentes da *Apple* (HESMANS et al., 2015b), permitindo a sua utilização em

grande escala. Desde 2013, todos os *tablets* e *smartphones* da *Apple* utilizam o *MPTCP* na aplicação de reconhecimento de voz chamada *Siri*, explorando um dos aspectos do protocolo cujo objetivo é manter a conectividade quando há falha em um dos caminhos possíveis na rede. Em julho de 2015, a maior companhia de telefonia sul-coreana, a *Korean Telecom* em parceria com a *Samsung* anunciou a implantação em escala comercial do protocolo *MPTCP* em diversos *smartphones* equipados com *Android*, combinando as interfaces *LTE* com wi-fi disponíveis, aumentando significativamente o ganho na taxa de utilização de banda desses dispositivos (OH; LEE, 2015).

O *MPTCP* é uma extensão do protocolo *TCP* definida pelo *IETF* através da *RFC 6824* (FORD et al., 2009) dando a possibilidade de uma conexão *TCP*, que por padrão é *single path*, poder funcionar como uma conexão de múltiplos caminhos, suportando assim uma maior diversidade para os dados de uma mesma conexão.

O desenvolvimento do protocolo *MPTCP* teve três principais objetivos de compatibilidade:

1. O protocolo deve ser utilizado pelas aplicações aproveitando a *API* de *socket* já existente;
2. O protocolo deve ser compatível com a rede onde está sendo implantado;
3. O protocolo deve manter a equidade entre os usuários da rede.

Este trabalho tem como objetivo rever o primeiro item da lista de compatibilidades que é requerido para o *MPTCP* operar de forma compatível na Internet, ou seja, implementar a *API* proposta pela *RFC 6897*. Esta *API* oferece para as aplicações a capacidade de controlar o protocolo *MPTCP* podendo, por exemplo, ligar ou desligar o suporte ao protocolo e ser capaz de adicionar ou remover um ou mais subfluxos. Utilizando essa extensão do protocolo *TCP*, temos a possibilidade de criar regras na aplicação de forma que ela possa aproveitar uma conexão através dos recursos fornecidos pelo protocolo *MPTCP* para abrir múltiplos fluxos de forma simultânea através de uma única interface ou de várias interfaces (FORD et al., 2009).

Assumindo que a aplicação necessita de um serviço de transmissão de bytes, precisamos de uma interface *(API)* que possa providenciar a adaptação de forma transparente para permitir o controle do protocolo *MPTCP*. A criação da *API* permitirá que cada aplicação possa ajustar de forma individual a quantidade de subfluxos necessários. Sendo assim, este trabalho apresenta uma implementação inicial da *API* descrita na *RFC 6897* para o protocolo *MPTCP* (SILVA; FERLIN; VERDI, 2016). As seguintes funções descritas no documento foram implementadas: `TCP_MULTIPATH_ENABLED`, `TCP_MULTIPATH_CONNID` e `TCP_MULTIPATH_ADD`, que permitem ligar e desligar o protocolo, verificar subfluxos exis-

tentes e adicionar novos subfluxos, respectivamente. Para validar nossa implementação, usamos a *API* para desenvolver uma aplicação que realiza o tratamento de fluxos elefantes.

1 Disposição da Dissertação

1.1 Motivações para esta Dissertação

- O advento do *MPTCP* reacendeu a discussão entre custo e benefício para utilizar *hosts multihoming*, porém pouco se comenta das vantagens que o *MPTCP* agrega ao utilizá-lo em *endhosts*;

- A característica padrão de operação do protocolo *MPTCP* é a de estabelecer um número fixo de subfluxos que são criados no momento de abertura da conexão. Atualmente, a única forma de alterar essa operação é por meio de variável de controle de sistema referente ao protocolo e que pode ser definida através do comando *sysctl*;

- Com a criação de uma *API*, qualquer aplicação que esteja rodando em um sistema compatível com o *MPTCP* pode, de forma individual, utilizar os recursos do protocolo, como por exemplo, ajustar a quantidade de subfluxos abertos automaticamente, criando aplicações mais inteligentes e cientes do fluxo passante, contribuindo com o encaminhamento do tráfego;

- Um dos principais objetivos de compatibilidade, adequação e aceitação do protocolo *MPTCP* é justamente que ele deve ser utilizado pelas aplicações aproveitando a *API* de *socket* já existente. Para isso, se faz necessário estender essa *API* de modo que atenda as operações de controle necessárias pelo protocolo *MPTCP*;

- No artigo apresentado em *Casado et. al.* (CASADO, 2013), os autores demonstram que uma solução efetiva para o tratamento dos fluxos elefantes não deve ser feita apenas por um *hardware* físico, mas sim a partir de uma interface padrão, como por exemplo, a partir da utilização de aplicações inteligentes que conseguem fazer o tratamento no *endhost*. Para que esse comportamento seja possível, a aplicação pode utilizar as funções da *API* desenvolvida para controle do *MPTCP*.

1.2 Objetivo desta Dissertação

O objetivo deste trabalho é implementar a *API* descrita pelo *IETF* na *RFC 6897* e demonstrar o conceito de *application awareness* com *MPTCP* através do uso dessa *API*. Como caso de uso da *API*, criamos uma aplicação ciente dos fluxos passantes com a intenção de combater um problema conhecido em redes de computadores, que é o gerenciamento de fluxos elefantes.

1.3 Atividades Desenvolvidas

1.3.1 Trabalhos Publicados

- SILVA, A. ; FERLIN, S. ; VERDI, FÁBIO L. Implementação Inicial da RFC 6897 para Auxílio no Tratamento de Fluxos Elefantes. *Workshop pré IETF/IRTF (III WPIETF-IRTF LAC) – XXXVI Congresso da Sociedade Brasileira de Computação – CSBC'16*. Porto Alegre – Brasil, Julho 2016.
- SANDRI, M. ; SILVA, A. ; ROCHA, L. A. ; VERDI, F. L. "On the Benefits of Using Multipath TCP and Openflow in Shared Bottlenecks". *The 29th IEEE International Conference on Advanced Information Networking and Applications (AINA'15)*. Gwangju – Korea, March 2015.
- SANDRI, M. ; SILVA, A. ; VERDI, F. L. "MultiFlow: Uma Solução para Distribuição de Subfluxos MPTCP em Redes OpenFlow". *XIII Workshop em Clouds e Aplicações (WCGA2015) – SBRC'15* Vitória – Brasil, Maio 2015.

1.3.2 Projetos Desenvolvidos Parcialmente Relacionados a esta Dissertação

- "The First Nornet Core in Latin America". *UFSCar Sorocaba*, 2014.

1.4 Organização da Dissertação

Esta dissertação está organizada da seguinte forma. No Capítulo 2 apresentamos os conceitos básicos e os trabalhos relacionados. No Capítulo 3 apresentamos o funcionamento e a implementação da solução. Por fim, no Capítulo 4, apresentamos os experimentos realizados e a discussão dos resultados obtidos.

2 Conceitos Básicos e Trabalhos Relacionados

Neste capítulo, serão apresentados os conceitos básicos para uma melhor compreensão do trabalho gerado e descrito nessa dissertação. Primeiramente, serão apresentados os protocolos utilizados seguindo uma linha histórica até chegar no *MPTCP*. Por fim, serão apresentados alguns trabalhos relacionados a essa dissertação.

2.1 Conceitos Básicos

2.1.1 Evolução dos Protocolos de Transporte Para Suportar Multipath

Os protocolos de transporte permitem a transferência de arquivos entre **hosts** e servidores. Nesta seção, faremos um histórico demonstrando a evolução desses protocolos até o *MPTCP*.

A camada de transporte é responsável pela transferência de dados entre um ou vários *hosts* e servidores. O desenvolvimento de protocolos que possuem a capacidade de criação de múltiplos caminhos, permitindo com isso a divisão de diversos subfluxos nas múltiplas interfaces disponíveis, continua sendo um dos principais tópicos de pesquisa em protocolos de transporte. Esse fato ocorre devido a frequência em que surgem cada vez mais dispositivos que utilizam várias interfaces ao mesmo tempo, assim como diferentes tecnologias de comunicação como, por exemplo, *Bluetooth*, *IEEE 802.11* e *LTE (Long-Term Evolution)*.

Hoje em dia, um fato comum em servidores é a utilização de duas ou mais interfaces *Ethernet*. Outra possibilidade, é o aproveitamento de protocolos de múltiplos caminhos utilizando apenas uma interface. Isso permite um espalhamento desses subfluxos utilizando algumas características dos mesmos e permitindo um roteamento pelos caminhos disjuntos que existem em uma rede.

Existem diversas propostas de protocolos para substituírem ou em alguns casos estenderem os protocolos de transporte já consolidados *(TCP e UDP)* (ARYE et al., 2012; GRINNEMO; BRUNSTROM; CHENG, 2014; PAASCH et al., 2012; RAICIU et al., 2011). Com isso, podemos considerar duas linhas de análise para se implementar um protocolo *multipath* que seja funcional:

1. Desenvolver um novo protocolo, com uma pilha própria e aplicando um novo paradigma; ou

2. Utilizar um protocolo já existente, como por exemplo o *TCP* e poder estender ele com a finalidade de suportar múltiplos caminhos.

Em relação ao item 1, podemos citar o protocolo *SCTP* (STEVENS; FENNER; RUDOFF, 2004) que foi criado para substituir os atuais protocolos de transporte existentes na camada de transporte *(TCP e UDP)*. O *SCTP* consegue juntar os dois modos existentes de operação *(por datagrama e por stream)* no mesmo protocolo, permitindo assim que o desenvolvedor possa escolher qual a melhor opção para a aplicação que irá utilizar o protocolo. O modo de *stream* replica um fluxo *SCTP* de um servidor fim-a-fim e consegue transmitir entre as diversas interfaces disponíveis. Em caso de perda de pacote no **enlace** principal, o pacote é substituído por uma réplica de um outro fluxo.

Essa quebra de paradigma que surgiu com a necessidade de se utilizar um novo protocolo em servidores e clientes, sem contar o desenvolvimento de aplicações que o suportassem acabou dificultando a adoção do *SCTP*. O *SCTP* é um protocolo que continua em evolução constante. A versão mais atual do *SCTP* chamada de *(CMT-SCTP)* (DREIBHOLZ et al., 2010) permite que os fluxos de dados possam ter a carga de pacotes realmente distribuídas e não apenas replicadas como em sua versão anterior. Isso ocorre devido ao modo de transmissão utilizado que é chamado de Transferência Concorrente por MultiCaminho *(CMT)*.

Em relação ao item 2, temos o *Multipath TCP (MPTCP)*. O *MPTCP* é uma extensão do protocolo *TCP*, sendo assim, ele se utiliza da mesma pilha já existente no protocolo. Ele também se baseia em Transferência Concorrente por Multicaminhos, distribuindo assim os dados dentro de seus subfluxos de forma arbitrária.

Para funcionar, o *MPTCP* utiliza os mesmos campos dentro do cabeçalho *TCP* para sinalização e, para especificar a sua implementação, utiliza um trecho reservado no campo *options* para sinalização.

Por ser uma extensão do protocolo *TCP*, o *MPTCP* acaba sendo mais transparente que o *SCTP*. Com isso, a sua aceitação é praticamente imediata, pois para habilitá-lo, basta utilizar o protocolo *TCP*.

Na Seção 2.1.3, descreveremos com maiores detalhes o funcionamento do protocolo *MPTCP*.

2.1.2 Evolução dos Protocolos de Roteamento Para Balanceamento de Carga

É conhecido que os protocolos de roteamento são os responsáveis por rotear segmentos ou mensagens de muitos protocolos, como por exemplo, do *TCP, UDP, SCTP* e tantos outros. O principal protocolo utilizado para roteamento é o protocolo *IP*, que engloba a camada inferior, em relação a camada de transporte, do modelo *TCP/IP*.

A rede *IP* utiliza técnicas de troca de tabelas de alcance e roteamento que permitem o cálculo de distâncias e melhores caminhos. Porém, a utilização dessa técnica chamada de *best effort* se mostra em desvantagem quando há a necessidade de um balanceamento de carga a partir da camada de transporte, como é o caso do *MPTCP*.

Para que o *MPTCP* ou qualquer outro protocolo que utilize subfluxos seja roteado de forma a distribuir os subfluxos pela rede, se faz necessário a utilização de protocolos de roteamento que possam garantir a distribuição desses subfluxos em rotas diferentes. Nesse caso, é interessante a utilização de protocolos como o *ECMP* por exemplo. O *ECMP (Equal-Cost-Multipath)* utiliza um algoritmo de *Hash* sobre a tupla de 5-campos (endereços IP origem e destino, portas origem e destino, tipo de protocolo) para definir o espalhamento. Deste modo, os subfluxos *MPTCP* serão espalhados por caminhos distintos em uma rede, justamente por utilizar uma atribuição randômica de portas.

2.1.3 Multipath TCP

O protocolo de controle de transmissão *(TCP)* é um dos principais protocolos utilizados no paradigma atual da *Internet*, pois ele permite criar serviços onde os fluxos de *bytes* que passam são confiáveis e por isso acaba sendo utilizado por diversas aplicações que demandam confiabilidade na transmissão de dados nos mais variados tipos de dispositivos, desde *smartphones* até servidores em redes de *data centers*.

Mesmo pertencendo a uma das partes fundamentais da *Internet* que conhecemos hoje, o protocolo *TCP* continua em constante processo de evolução. Com o surgimento de serviços mais complexos, uma das evoluções mais recentes que acompanhou esta mudanca é o *Multipath TCP (MPTCP)* (RAICIU et al., 2012; RAICIU; HANDLEY; BONAVENTURE, 2013).

O *MPTCP* altera uma das premissas básicas da especificação original do protocolo *TCP* que determina que uma conexão *TCP* será sempre identificada através da tupla de 4 elementos que consiste nos endereços *IP* de origem e destino e das portas de origem e destino. Todos os pacotes que são enviados por uma conexão *TCP* sempre serão formados por essa tupla de 4 elementos. O *TCP*, como conhecemos, não consegue explorar hoje todos os recursos oferecidos pelo *hardware* ou pelas tecnologias atuais. Um exemplo disso são os dispositivos *host multi-homed* ou um *smartphone* que normalmente possuem a capacidade de trabalhar com múltiplas interfaces ou múltiplos caminhos. Nestes casos, uma conexão *TCP* não poderá se beneficiar das múltiplas interfaces e caminhos caso não exista um mecanismo que ofereça suporte para isso.

O *MPTCP* (RAICIU; HANDLEY; BONAVENTURE, 2013) resolve esta questão permitindo que os pacotes pertencentes a uma determinada conexão sejam transmitidos por diferentes interfaces e endereços através dos subfluxos criados para cada conexão que,

inicialmente, se aproveitam da estrutura de uma conexão *TCP* original. Estes subfluxos potencialmente serão enviados individualmente pelas interfaces e caminhos disponíveis, tarefa que convém ao agendador *(scheduler)* e ao mecamismo de controle de congestionamento *(congestion control)* decidirem.

O *MPTCP* ganhou reconhecimento comercial e foi adaptado aos sistemas operacionais recentes da *Apple* (HESMANS et al., 2015b), permitindo a sua utilização em grande escala. Desde 2013, todos os *tablets* e *smartphones* da *Apple* utilizam o *MPTCP* na aplicação de reconhecimento de voz chamada *Siri*, explorando um dos aspectos do protocolo cujo objetivo é manter a conectividade quando há falha em um dos caminhos possíveis na rede. Em julho de 2015, a maior companhia de telefonia sul-coreana, a *Korean Telecom* em parceria com a *Samsung* anunciou a implantação em escala comercial do protocolo *MPTCP* em diversos *smartphones* equipados com *Android*, combinando as interfaces *LTE* com *wi-fi* disponíveis, aumentando significativamente o ganho na taxa de utilização de banda desses dispositivos (OH; LEE, 2015).

O desenvolvimento do protocolo *MPTCP* foi motivado por alguns cenários que necessitam enviar dados utilizando a mesma conexão através de interfaces e endereços diferentes, como por exemplo, *smartphones* equipados com interface celular e *wifi* (PAASCH et al., 2012), ou redes de *datacenters* (RAICIU et al., 2011) cuja malha topológica oferece múltiplos caminhos para os diferentes subfluxos, que seguem possivelmente para um mesmo destino.

O *MPTCP* foi desenvolvido tendo três principais objetivos de compatibilidade em mente (FORD et al., 2011):

1. O protocolo deve ser utilizado pelas aplicações aproveitando a *API* de *socket* já existente. Atualmente, o *Linux* (PAASCH; BARRE et al., 2013) possui uma implementação do protocolo *MPTCP* que resolve em parte essa situação;

2. O protocolo deve ser compatível com a rede onde está sendo implantado. Para alcançar este objetivo, o protocolo possui diversos mecanismos que possibilitam a sua comunicação **com outros ativos** existentes na rede (RAICIU et al., 2012; RAICIU; HANDLEY; BONAVENTURE, 2013);

3. O protocolo deve manter a equidade entre os usuários da rede. Para alcançar esse objetivo, vários esquemas de controle de congestionamento foram propostos e implementados.

Este trabalho tem como objetivo rever o primeiro item da lista de compatibilidades que é requerido para o *MPTCP* operar de forma compatível na Internet, ou seja, implementar a *API* proposta pela *RFC 6897*. Esta *API* oferece para as aplicações a capacidade de controlar o protocolo *MPTCP* podendo, por exemplo, ligar ou desligar o suporte ao

protocolo e ser capaz de adicionar ou remover um ou mais subfluxos. Hoje, a única forma de controlar o número de subfluxos que são fixos e criados no momento de abertura da conexáo é via comando *sysctl*.

A criação da *API* permitirá que todas as aplicações que estão rodando no sistema possam ajustar de forma individual a quantidade de subfluxos necessários, servindo como motivação para este trabalho.

2.1.3.1 Funcionamento do MPTCP

O *MPTCP* (RAICIU; HANDLEY; BONAVENTURE, 2013) permite aos *hosts* trocarem pacotes que pertencem a uma determinada conexão entre várias interfaces ou caminhos na rede. Para que esta tarefa possa ser executada, cada conexão *MPTCP* é composta por várias conexões *TCP* que são denominadas subfluxos (RAICIU; HANDLEY; BONAVENTURE, 2013).

O *MPTCP* é uma extensão do protocolo *TCP* definida pelo *IETF* em (FORD et al., 2009) dando a possibilidade de uma conexão *TCP*, que por padrão é *single path*, poder funcionar como uma conexão de múltiplos caminhos, suportando assim uma maior diversidade para os dados de uma mesma conexão.

Uma das principais vantagens do *MPTCP* em relação a outras soluções, como por exemplo, o *SCTP* (PAASCH; BONAVENTURE, 2014), é poder usar a mesma estrutura do protocolo *TCP* para trafegar informação, fazendo com que o *MPTCP* seja transparente para aplicações que utilizam o protocolo. Isso é possível pois o MPTCP se aproveita da mesma *API* de *socket* do protocolo *TCP* como mostra a Figura 1.

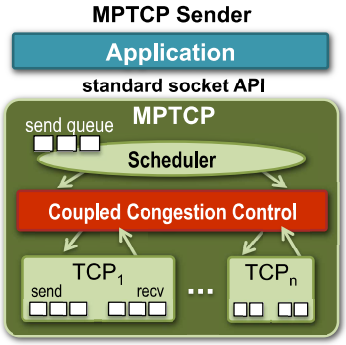

Figura 1 – Pilha TCP com suporte ao MPTCP.

O *MPTCP* se aproveita do processo de *three-way handshake* do *TCP* para efetuar a negociação necessária que permite seu uso. Basicamente, ele possui 3 fases que são:

- *Estabelecer ou fechar uma nova conexão MPTCP;*
- *Adicionar ou remover subfluxos em uma conexão MPTCP;*

- *Transmitir dados através da conexão MPTCP.*

Para estabelecer uma nova conexão *MPTCP*, o *host* de origem envia um pacote *SYN* com a *flag* de *MP_CAPABLE* sinalizada, mais uma chave randômica gerada pelo servidor e que será utilizada para o cálculo do *hash* que servirá de identificador para a conexão. Nesse caso, se o *host* de destino suportar o protocolo *MPTCP*, ele devolve a resposta com um *SYN+ACK* contendo o *MP_CAPABLE* sinalizado mais uma chave aleatória escolhida pelo servidor. O terceiro *ACK* do processo de *three-way handshake* também vai incluir a *flag* de *MP_CAPABLE* sinalizada e as chaves que serão utilizadas como identificador, estabelecendo assim a conexão, conforme a Figura 2.

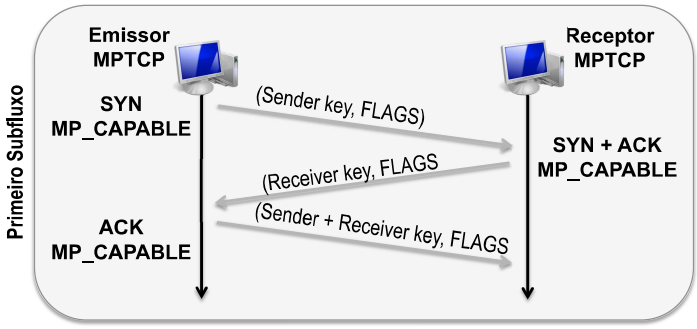

Figura 2 – Criação de uma conexão MPTCP (MP_CAPABLE).

Caso seja necessária a criação de um novo subfluxo em uma conexão *MPTCP* existente, é efetuado um novo processo de *three-way handshake*, onde o *host* de origem envia o primeiro *SYN* com o campo *MP_JOIN* que possui o *token* da conexão *MPTCP* existente. Este *token* é o resultado da chave que foi gerada no estabelecimento da conexão.

O *host* de destino recebe o pacote e calcula o *HMAC (Hash-based Message Authentication Code)* a partir do *token*, devolvendo um *SYN+ACK* com o resultado do *HMAC*. O *host* de origem recebe e valida o *HMAC* devolvendo um *ACK* com o *HMAC*, assim estabelecendo a validação do subfluxo, conforme a Figura 3.

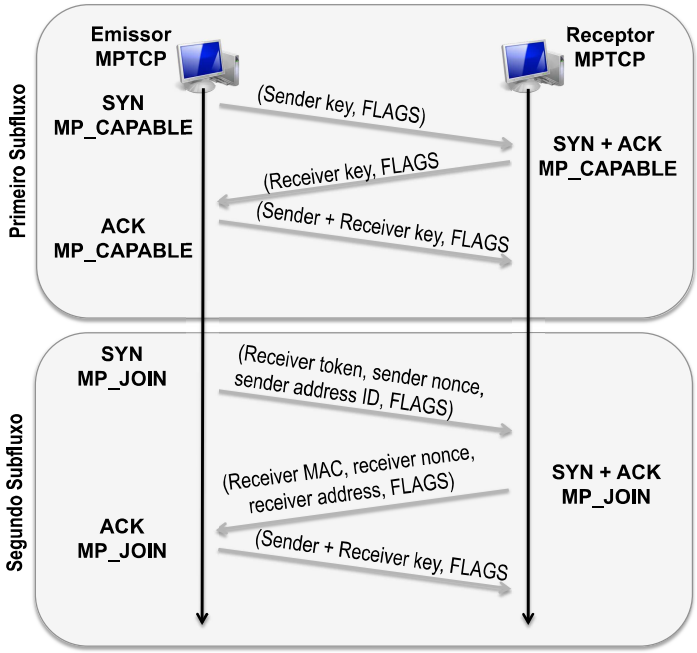

Figura 3 – Criação de um novo subfluxo em uma conexão existente (MP_JOIN).

Um detalhe muito importante na implementação do *MPTCP* são os gerenciadores de caminhos (*path managers*). Eles possuem a função de decidir como os subfluxos serão estabelecidos e atualmente há dois disponíveis: *fullmesh* e *ndiffports*.

Em relação à criação de subfluxos, no caso do *ndiffports*, apenas o cliente é responsável pela criação de subfluxos. O servidor nunca cria subfluxos, pois o cliente pode estar atrás de um *firewall* ou *NAT* que bloqueia a tentativa de conexão (EARDLEY, 2013). Em *fullmesh* é possível a criação do subfluxo a partir do servidor, caso ele conheça o *IP* de conexão e atue como passivo, esperando uma requisição de conexão por parte do cliente. Porém, o modo comum de operação é o ativo (onde o cliente cria o subfluxo).

Os gerenciadores de caminhos também possuem algumas particularidades em relação ao seu modo de operação. O *fullmesh* escuta os eventos a partir das interfaces de rede e cria um subfluxo para o servidor para cada interface ativa. Estes subfluxos são criados imediatamente após a criação da conexão ou quando a interface é ativada a partir da criação do primeiro subfluxo, o subfluxo principal. Isto permite, por exemplo, que os *smartphones* possam reagir no caso de uma perda de conectividade (PAASCH et al., 2012).

Já o gerenciador de caminhos *ndiffports* cria `N` subfluxos em uma mesma interface de forma imediata após o estabelecimento da conexão. Este gerenciador de caminhos foi desenvolvido tendo o foco em *datacenters* onde são permitidos a utilização de diferentes caminhos que possam ser balanceados com *Equal Cost Multipath (ECMP)* (RAICIU et al., 2011).

2.1.3.2 RFC 6897

Conforme citado anteriormente, o protocolo *MPTCP* adiciona a capacidade de se utilizar caminhos múltiplos em uma conexão *TCP*. As motivações para isso incluem o aumento da taxa de transferência e a resiliência no caso de uma possível falha de rede.

O texto encontrado na *RFC 6897* descreve algumas questões sobre a utilização das funcionalidades do protocolo *MPTCP* com as aplicações, e sugere uma interface básica de aplicação que estende a interface *TCP* já existente. A idéia é apresentar quais são os efeitos que o protocolo *MPTCP* exerce sobre as aplicações, como por exemplo, a questão de desempenho comparando-o ao *TCP* e a interoperabilidade entre *MPTCP* e as aplicações.

A motivação para a criação de uma *API* que suporte o protocolo *MPTCP* consiste no fato de permitir que aplicações possam utilizar uma interface já conhecida. Por isso, a implementação é efetuada através da extensão da interface de *sockets* que já é utilizada como padrão na implementação de aplicações que precisam interagir entre si utilizando o protocolo *TCP*. Isso permite a mesma facilidade de implementação, pois quem já utiliza a interface de *sockets TCP* poderá utilizar a *interface MPTCP* da mesma forma.

A *API padrão* já consolidada para aplicações *TCP* é a interface via *socket*. A partir disso, o documento propõe uma maneira abstrata de estender esta interface de forma que possa suportar o *MPTCP* através de operações que obtenham *(get)* e definam *(set)* valores específicos do *MPTCP* através de uma **opção de socket** gerada no nível do protocolo *TCP*. Isso permite que aplicações, linguagens de programação e bibliotecas possam decidir como utilizar essas informações de forma transparente.

Uma *API MPTCP* básica consiste em um conjunto de novos valores que serão associados a um *socket MPTCP*. Estes valores podem ser utilizados para alterar propriedades em uma conexão *MPTCP* ou recuperar uma informação.

Os valores podem ser acessados através das chamadas de sistema já utilizadas na interface de *socket* como *setsockopt()* ou *getsockopt()*, através de símbolos específicos criados especialmente para alterar ou recuperar dados correspondentes ao protocolo *MPTCP* que serão acessados por parâmetros passados para essas chamadas.

As opções que são especificadas na Seção 5.3.1 da *RFC 6897* correspondem a uma versão básica da *API* e devem ser implementadas para uma utlização **dos recursos do protocolo** por outras aplicações:

- *TCP_MULTIPATH_ENABLE*: Habilita ou desabilita o *MPTCP*;
- *TCP_MULTIPATH_ADD*: Conecta o protocolo *MPTCP* em um conjunto de endereços locais definidos ou adiciona um conjunto de novos endereços locais em uma conexão *MPTCP* existente. Na prática esta opção permite que novos subfluxos sejam adicionados à uma conexão *MPTCP*;

- *TCP_MULTIPATH_REMOVE*: Remove um endereço local de uma conexão *MPTCP*. Na prática, esta opção permite que subfluxos existentes sejam removidos de uma conexão *MPTCP*;
- *TCP_MULTIPATH_SUBFLOWS*: Recupera os pares de endereços atualmente utilizados por subfluxos *MPTCP*;
- *TCP_MULTIPATH_CONNID*: Retorna o identificador de conexão local da conexão *MPTCP* atual.

A Tabela 1 mostra o conjunto de funções que devem ser implementadas na *API* de *socket* para permitir a manipulação do *MPTCP* através das aplicações.

Tabela 1 – Operações da API MPTCP (RFC 6897)

Nome	**Get**	**Set**	**Tipo de Dados**
TCP_MULTIPATH_ENABLE	x	x	Booleano
TCP_MULTIPATH_ADD		x	Lista de Endereços/Portas
TCP_MULTIPATH_REMOVE		x	Lista de Endereços/Portas
TCP_MULTIPATH_SUBFLOWS	x		Lista de Pares de Endereços/Portas
TCP_MULTIPATH_CONNID	x		Inteiro

Atualmente temos outras duas implementações da *API*. Uma foi implementada pela *Apple* que decidiu utilizar um tipo diferente de socket *Multipath TCP* com chamadas de sistema dedicadas para utilização com o recurso de voz *Siri*.

A outra implementação foi desenvolvida pelo grupo responsável pela pilha *Multipath TCP* no Linux e apresentada no *ANRW 2016* (HESMANS; BONAVENTURE, 2016) e que segue uma abordagem semelhante a apresentada nesse trabalho que é a extensão da *API* de *socket* já disponível.

A *API* da *Apple* não foi documentada oficialmente. Todo o conhecimento sobre a *API* de *Multipath TCP* deles foi extraído através dos códigos de exemplo que podem ser encontrados no repositório da *Apple* [1].

Ela basicamente define uma família de endereços `AF_MULTIPATH` para a criação de socket e define uma nova chamada de sistema com o nome de `connectx` para efetuar a conexão *Multipath TCP*.

Já a API criada pelo grupo responsável pela pilha do *Multipath TCP* no *Linux* tem uma proposta semelhante a desse trabalho, estendendo a API de socket já existente e expõe as estruturas de controle do *MPTCP* através da criação de novas opções de *socket* que são definidas em (HESMANS; BONAVENTURE, 2016).

[1] Ver em: http://opensource.apple.com/source/network_cmds/network_cmds-457/mptcp_client/mptcp_client.c e http://opensource.apple.com/source/netcat/netcat20/netcat.c

2.1.4 Aplicações Cientes

O conceito de ciência de contexto relacionado a sistemas aparece pela primeira vez em (DOURISH; BELLOTTI, 1992) como sendo um meio de entender as atividades de um grupo como um todo, o que acaba provendo um contexto para efetuar a própria atividade. Isso permite com que essa ciência de contexto ajude a gerenciar o processo de trabalho colaborativo entre os sistemas.

Quando o conceito se trata das aplicações, essa ciência de contexto é iniciada através de uma identificação básica das aplicações presentes na rede. Quando a rede sabe que a aplicação existe, o próximo passo é entender o estado e o fluxo das comunicações que essas aplicações estabelecem e transmitem. Através dessa ciência, a rede pode entender como os protocolos de aplicação trabalham, permitindo assim uma melhor utilização da rede, otimizando o seu tráfego, podendo injetar inteligência através de regras que podem ser aplicadas e com isso efetuando um melhor gerenciamento e dando uma visão mais profunda da rede em que essas aplicações estão sendo carregadas.

No nosso caso de uso, utilizamos o conceito de *application awareness* através de uma regra que estabelece a identificação de fluxos elefantes, permitindo assim, através da utilização do protocolo *MPTCP* que a aplicação consiga abrir subfluxos de forma dinâmica (SILVA; FERLIN; VERDI, 2016), obtendo uma maior vazão e utilizando de forma melhorada os recursos disponíveis na rede.

2.1.5 Fluxos Elefantes

Na literatura, a definição de fluxos elefantes varia de acordo com o trabalho estudado. Sendo assim, identificamos com base nos estudos realizados, três definições que são normalmente utilizadas para caracterizar os fluxos elefantes conforme podemos ver na Figura 4.

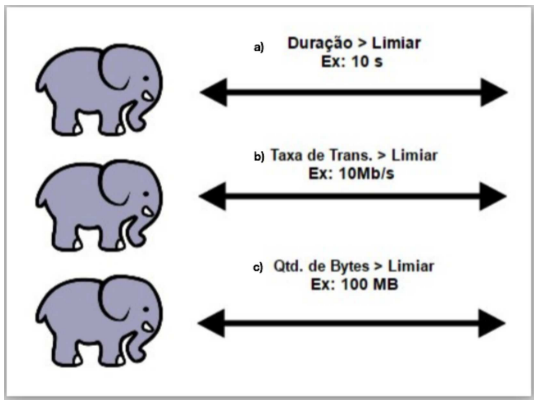

Figura 4 – Tipos de fluxos elefantes identificados.

Um fluxo pode ser considerado elefante quando ele tem uma duração acima de uma determinado limiar de tempo (fluxo a) na Figura 4), o que pode ocasionar atrasos significativos em outros fluxos na rede, conforme visto em (XU; LI,). Um fluxo também pode ser definido como elefante caso a taxa de transmissão (em Mbps) deste fluxo ultrapasse um certo limiar (AL-FARES et al., 2010) (fluxo b) na Figura 4). E por fim, um fluxo pode ser considerado elefante quando a quantidade de bytes transferidos ultrapassa um limiar estabelecido, normalmente, 100MB (GREENBERG et al., 2009; CURTIS; KIM; YALAGANDULA, 2011) (fluxo c) na Figura 4). Esta última definição é a mais utilizada e que, por consequência, também utilizaremos nesse trabalho que está sendo apresentado.

Em conjunto com a definição qualitativa dos fluxos elefantes (determinada pelo seu tamanho e tempo de vida), pode-se utilizar também uma definição quantitativa, onde o fluxo elefante é determinado arbitrariamente por limiares que são estritamente definidos.

Diversas propostas sugerem a criação de mecanismos para controlar a banda e o atraso dos fluxos, garantindo assim a qualidade e prazo das respostas para os usuários de aplicações em um *data center*.

Em *Casado et. al.* (CASADO, 2013), os autores descrevem alguns items que eles acreditam que sejam importantes na criação de uma solução que resolva o problema de fluxos elefantes, como por exemplo:

1. Utilizar filas distintas para separar os fluxos pequenos dos fluxos elefantes;

2. Utilizar caminhos diferentes para separar os fluxos pequenos dos fluxos elefantes;

3. Encaminhar os fluxos elefantes para uma rede exclusiva;

4. Transformar os fluxos elefantes em fluxos pequenos e espalhar esses fluxos através dos múltiplos caminhos existentes na rede.

Atualmente, as soluções já existentes procuram identificar e separar os fluxos elefantes dos fluxos camundongos, porém, normalmente elas acabam possuindo algumas limitações, como por exemplo, diminuição de desempenho quando há gargalos ou colisões na rede. Esse fato nos levou a desenvolver uma aplicação que aproveita a *API* do protocolo *MPTCP* descrita na *RFC 6897* para efetuar, de forma ciente, a quebra dos fluxos elefantes em *endhost* transformando-os em fluxos camundongos e espalhando-os através dos subfluxos abertos, adotando assim a solução 4 listada acima. A Figura 5 ilustra o processo de transformação dos subfluxos.

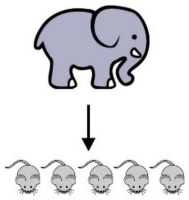

Figura 5 – Quebra dos fluxos elefantes em fluxos camundongos.

2.2 Trabalhos Relacionados

2.2.1 Multipath TCP

Diversos pesquisadores têm explorado como o protocolo *MPTCP* deve gerenciar os fluxos e as interfaces disponíveis. Paasch et Al. (PAASCH et al., 2012) avaliam como os dispositivos *wireless* se adaptam a perdas de conectividade. Este artigo propõe três modos de operação para o protocolo *MPTCP* em *smartphones*: *backup* (quando o protocolo abre subfluxos para todas as interfaces, porém, utiliza apenas um conjunto desses subfluxos), *single path* (tem um processo de funcionamento semelhante ao de *backup*, porém, utiliza apenas um subfluxo para estabelecer uma conexão a qualquer momento) e *full-mptcp* (modo normal de operação do protocolo *MPTCP*, quando todos os subfluxos são utilizados).

Bocassi et al. (BOCCASSI; FAYED; MARINA, 2013) propõem o gerenciador de caminhos chamado *Binder* que explora o roteamento de origem livre e utiliza o *MPTCP* para agregar caminhos diferentes em redes de malha sem fio. Lim et al. (LIM et al., 2014a) propõem uma extensão para o *MPTCP* que permite adaptar a utilização dos subfluxos baseado na informação extraída a partir da camada de *MAC*. Essa extensão foi validada experimentalmente, porém, a implementação não possui muitos detalhes.

Lim et al. (LIM et al., 2014b) também propõem o *eMPTCP* que retarda o estabelecimento dos subfluxos em *smartphones* que estão utilizando a interface *LTE*. Porém, quando o smartphone troca de interface, por exemplo, *(LTE para Wi-fi)*, o artigo propõe a reinicialização da estimativa do *RTT* do subfluxo utilizado na interface *LTE* para forçar a utilização do mesmo na nova interface selecionada. Essa solução melhora a utilização do subfluxo, mas não é elegante para solucionar o problema do gerenciamento de subfluxos.

Schmidt et al. (SCHMIDT et al., 2013) propõem a utilização do conceito de *socket intents* que permite as aplicações informarem à conexão sobre o seu conhecimento da comunicação estabelecida, podendo gerar informação que serve como previsão para melhoria da conexão. Estes *intents* incluem informação sobre o tipo de transferência *(query, bulk, stream)* ou a informação sobre o fluxo (número de bytes, duração).

Hesmans et al. (HESMANS et al., 2015a) definem a criação de um gerenciador de caminhos que utiliza o conceito de *socket intents*, extraindo as informações dos *intents* e usando as mesmas como parâmetros, permitindo que aplicações possam manipular o protocolo *MPTCP*. O trabalho faz a exposição das funções do MPTCP através da criação de um gerenciador de caminhos chamado *Netlink* que é implementado em nível de usuário, porém, escrever código para enviar e receber eventos via *Netlink* pode ser complicado, justificando a criação da *API*.

O recente trabalho apresentado por Hesmans et al. (HESMANS; BONAVENTURE, 2016) é que mais se aproxima da pesquisa sendo realizada nesta dissertação. Tal trabalho também se propõe a implementar uma *API* para manipulação e controle do protocolo *MPTCP*. Porém, tal solução não segue o padrão descrito na *RFC 6897* pois os autores acabam implementando uma *API* que possui um conjunto de funções e estruturas que não seguem o padrão *POSIX*. Uma consequência imediata disso é a dificuldade de portar a solução para outros Sistemas Operacionais já que os autores daquele trabalho focam em uma implementação *Linux*. Por outro lado, nesse trabalho de dissertação, desenvolvemos a *API* baseada na *RFC 6897* respeitando, com isso, os padrões de portabilidade entre os Sistemas Operacionais.

Porém, mesmo que a *RFC 6897* (SCHARF; FORD, 2013) proponha a criação de algumas extensões na interface *(API)* básica de *socket* para permitir que aplicações possam adicionar ou remover subfluxos em uma conexão *MPTCP*, nenhuma implementação atualmente faz uso desta extensão.

2.2.2 Aplicações Cientes de Contexto

Existem alguns trabalhos que demonstram a utilização desse conhecimento prévio da informação ***(context awareness)*** e como ele acaba sendo aplicado visando uma melhor utilização dos recursos fornecidos, sendo o principal trabalho relacionado ao nosso contexto o descrito em (SCHMIDT et al., 2013).

Schmidt et al. (SCHMIDT et al., 2013) propõem a utilização do conceito de *socket intents* que permite às aplicações informarem a conexão sobre o seu conhecimento da comunicação estabelecida, podendo gerar informação que serve como previsão para melhoria da conexão. Estes *intents* incluem informação sobre o tipo de transferência *(query, bulk, stream)* ou a informação sobre o fluxo (número de *bytes* e duração).

No nosso caso de uso, utilizamos o conceito de *application awareness* através de uma regra que estabelece a identificação de fluxos elefantes, através da utilização do protocolo *MPTCP*, permitindo assim que a aplicação consiga de forma dinâmica abrir subfluxos (SILVA; FERLIN; VERDI, 2016), tendo uma maior vazão e utilizando de forma melhorada os recursos disponíveis na rede.

2.2.3 Fluxos Elefantes

Normalmente, as propostas procuram identificar e separar os fluxos elefantes dos fluxos camundongos. Algumas alternativas que tentam resolver o problema serão apresentadas a seguir. As propostas citadas possuem algumas limitações que servem como motivação para o desenvolvimento da solução apresentada neste trabalho.

A proposta *Hedera* apresentada por *Al-Fares et. al.* em (AL-FARES et al., 2010) consiste em criar um controle inspecionando os fluxos que são encaminhados pela rede através do acesso direto aos comutadores *(switches e routers)* disponíveis, criando uma tabela de agendamento dinâmico de caminhos para os fluxos. O *Hedera* possui um centralizador *(controlador OpenFlow)* que gerencia essa tabela de caminhos não-conflitantes. Esse centralizador realiza a atualização e adaptação de caminhos de forma dinâmica conforme o tráfego da rede no instante da inspeção do fluxo. Através desse recurso, ele consegue orientar os comutadores de rede e realizar o redirecionamento e encaminhamento dos fluxos de acordo com a tabela que foi gerada. Entretanto, essa atualização constante de caminhos acaba consumindo muito recurso, pois sobrecarrega o enlace e degrada o desempenho.

A proposta *DARD (Distributed Adaptive Routing for Data Center Networks)* apresentada por *Wu et. al.* em (WU; YANG, 2012) descreve um algoritmo que é implementado usando teoria dos jogos para escolher o melhor caminho. O algoritmo seleciona o caminho sempre convergindo para o *equilíbrio de Nash*, fazendo que com isso seja dada a solução ótima para o problema da escolha do melhor caminho de acordo com o tamanho do fluxo. O algoritmo é distribuído entre diversos controladores que gerenciam essa otimização de caminhos, selecionando por onde o fluxo deve seguir de acordo com o seu tamanho. Para tratar fluxos elefantes, o algoritmo utiliza um dispositivo instalado no *end-host* que sinaliza o fluxo e envia para o controlador. Essa abordagem é considerada problemática pois o tratamento é feito no controlador, podendo degradar o desempenho.

O *DevoFlow* apresentado por *Curtis et. al.* em (CURTIS et al., 2011) sugere algumas modificações no comportamento do protocolo *OpenFlow*. A idéia central é manter no controlador apenas os fluxos significativos. Esses fluxos significativos são determinados de acordo com um levantamento estatístico feito pelo controlador. O controle dos fluxos menos significativos é devolvido aos comutadores *(switches e roteadores)*. O principal problema dessa abordagem é que os comutadores precisam ser robustos, pois eles são essenciais no funcionamento da arquitetura proposta. Como esse é um fator determinante, ele pode gerar um aumento no custo de manutenção da solução.

A idéia apresentada na proposta *ElephantTrap* demonstrada por *Lu et. al.* em parceria com a *Cisco* em (LU et al., 2007) sugere a criação de um dispositivo responsável pela detecção dos fluxos elefantes em um *link* de rede. Usando o protocolo *NetFlow*, o algoritmo consegue monitorar os fluxos e determinar a ação que deve ser tomada de acordo

com o seu tamanho. Uma tabela de gerenciamento de fluxos é criada nos dispositivos, permitindo um controle total sobre o tráfego que passa. No entanto, um dos principais problemas dessa solução é que essa aplicação sendo aplicada diretamente no comutador pode consumir mais recursos e degradar outras aplicações.

Outra idéia que sugere um dispositivo físico é apresentada em *Tracking elephant flows in Internet backbone traffic with an fpga-base cache* por *Zadnik et. al.* em (ZADNIK et al., 2009) que utiliza os recursos iminentes do *FPGA* para atuar no problema de gerenciamento dos fluxos elefantes. A idéia é utilizar os recursos de *cache* do *hardware* para manter e atualizar uma tabela de estado que será utilizada para **detecção, tomada de decisão e ação** de acordo com o tamanho dos fluxos. Um problema encontrado nessa plataforma se refere justamente a atualização do cache, que está em uma área de memória onde o acesso é fácil, porém, não é rápido. Isso compromete a atualização das tabelas de fluxos, ocasionando uma latência considerável que pode degradar as aplicações.

A proposta *TinyFlow* apresentada por *Xu. et. al.* em (XU; LI,) parte do princípio de que o protocolo *ECMP* não consegue distinguir os tipos de fluxos (camundongos ou elefantes). A idéia é utilizar o controlador *OpenFlow* para dividir um fluxo elefante em vários fluxos camundongos. Para que o método funcione, é necessária a detecção do fluxo que atenda ao tamanho definido que é atribuído por uma variável de *Threshold*. Quando o valor de *Threshold* é atingido, o fluxo é quebrado e distribuído via *ECMP*. Como o trabalho de detecção e quebra são feitos no controlador, isso pode aumentar o processamento do controlador, gerando latência e ocasionando problemas no desempenho das aplicações.

A proposta do *Mahout* apresentada por *Curtis et. al.* em (CURTIS; KIM; YALAGANDULA, 2011) demonstra uma arquitetura de gerenciamento de tráfego com baixo *overhead*. Para estabelecer o *overload* baixo, a detecção dos fluxos elefantes é feita no *end-host*. Uma camada de monitoramento é adaptada na pilha de rede do Sistema Operacional do *end-host*, permitindo a sinalização de um possível fluxo elefante. Quando o fluxo elefante é detectado, ele é repassado para o controlador *OpenFlow* que se responsabiliza pela ação a ser tomada. Quando comparado com outros trabalhos, a proposta acaba sendo interessante devido a sua rápida detecção dos fluxos, baixo custo de monitoramento e baixa utilização de recursos nos comutadores. Um dos problemas encontrados no *Mahout* se deve justamente na modificação da pilha de rede, pois dependendo da forma como o processamento de pacotes é feito, pode haver o descarte de alguns pacotes importantes. Outro problema detectado nessa arquitetura é em relação ao valor de *threshold* que é fixo, pois em uma rede de *datacenter* que costuma ter tráfego dinâmico, o tamanho de um fluxo elefante pode ser variável.

Diversos trabalhos na literatura que têm o gerenciamento de fluxos elefantes como tema abordam as soluções tratando o problema diretamente na arquitetura de rede. Quer seja aplicando dispositivos físicos ou mudando o comportamento dos comutadores conforme

visto em (AL-FARES et al., 2010; LU et al., 2007; ZADNIK et al., 2009), mudando o comportamento dos protocolos como em (CURTIS et al., 2011; XU; LI,) ou de forma híbrida conforme visto em (WU; YANG, 2012; CURTIS; KIM; YALAGANDULA, 2011). Porém, conforme relatado, todas as soluções até aqui apresentadas acabam resolvendo o problema, mas trazem consequências que podem degradar as aplicações em uma rede.

Finalização do Capítulo

Neste capítulo, apresentamos alguns conceitos básicos que são relacionados para um melhor entendimento desse trabalho de dissertação. Passamos pela evolução dos protocolos de transporte até chegar no *MPTCP*, onde falamos sobre o seu funcionamento, descrevendo a criação de uma *API* e a possibilidade de criação de aplicações cientes de contexto *(application awareness)*. Também comentamos brevemente sobre o nosso caso de uso. Finalizamos o capítulo com os trabalhos relacionados que são referentes ao desenvolvimento dessa dissertação.

3 Desenvolvimento e Funcionamento da Solução

3.1 Implementação da Pilha de Rede em S.Os baseados em UNIX

Geralmente, a tarefa de um sistema operacional é gerenciar recursos de hardware em dispositivos de computação e fornecer serviços para aplicações. Consequentemente, gerenciar dispositivos de rede e garantir acessos a eles através de aplicações também é parte dessa tarefa. Além disso, os Sistemas Operacionais gerenciam as conexões e direcionam os dados para as aplicações corretas. Para realizar essa atividade, uma *API* de *Socket* normalmente é implementada.

A maioria dos Sistemas Operacionais se dividem em pelo menos dois domínios de proteção de sua memória, espaço de *kernel(kernel space)* e espaço de usuário *(user space)*. No espaço de *kernel*, o sistema operacional carrega e executa tarefas pertinentes a funcionalidade do núcleo do sistema, como por exemplo, o acesso aos recursos de *hardware*. O espaço de usuário *(user space)* é a parte onde as aplicações em nível de usuário são executadas. Se uma aplicação acessa funções no espaço de kernel, ela está acessando funções de chamada de sistema *(system calls)*.

A Figura 6 mostra os componentes de uma implementação de rede em um Sistema Operacional baseado em *UNIX*. Alguns deles encontram-se em espaço de usuário, alguns em espaço de *kernel*.

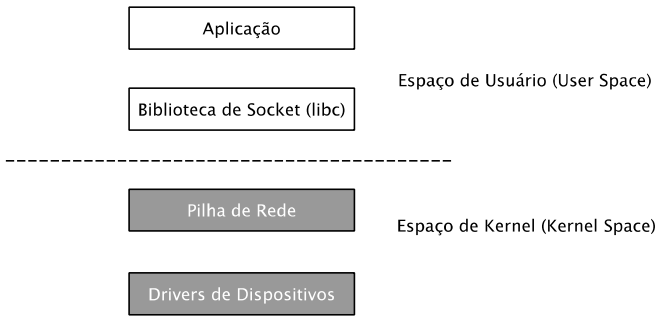

Figura 6 – Esquema da implementação de Rede em um Sistema Operacional Moderno.

As aplicações são executadas em espaço de usuário e criam chamadas para a *API* de *Socket*. As funções chamadas pelas aplicações são implementadas dentro da biblioteca de *Socket* que ainda residem no espaço de usuário e fazem parte da *libc*, uma biblioteca padrão existente em sistemas *UNIX*. A *API* de *Socket* cria uma interface entre a aplicação

e a biblioteca de *socket* e, devido a sua padronização, continua sendo a mesma entre vários Sistemas Operacionais, permitindo que as aplicações possam ser executadas de forma transparente, independente de qual Sistema Operacional está sendo utilizado.

A partir da biblioteca de *socket*, as chamadas do sistema *(system calls)*, no caso as funções em espaço de *kernel* são chamadas. Essas chamadas de sistema podem ser diferentes dependendo de qual Sistema Operacional e arquitetura de *CPU* está sendo utilizada.

Dentro da pilha de rede implementada no *kernel*, os pacotes são montados para serem enviados para a rede ou desmontados para serem recebidos pela aplicação. Isso é feito através da criação de cabeçalhos de protocolo e adicionando eles as mensagens ou lendo esses cabeçalhos de protocolo e os processando. Mensagens são enviadas ou recebidas a partir dos *drivers* de dispositivos **e** *hardwares* de rede. Toda estrutura e implementação dentro do espaço de *kernel* é totalmente dependente do Sistema Operacional que está sendo utilizado. Os componentes em espaço de usuário podem ser similares entre os Sistemas Operacionais, com isso, as alterações efetuadas em espaço de usuário podem ser facilmente portadas entre Sistemas Operacionais diferentes.

3.2 Funcionamento Interno do Protocolo MPTCP

O desenvolvimento da *API* descrita na *RFC 6897* precisou de um estudo do funcionamento da implementação existente da *RFC 6824*, que descreve o funcionamento do protocolo *Multipath TCP (MPTCP)* e que foi implementada atualmente tendo como base o *Kernel* do *Linux*. Para o desenvolvimento da *API* descrita nesse trabalho de dissertação, foi necessário um entendimento detalhado dessa implementação para identificação dos pontos de entrada, estruturas de dados e chamadas de sistema existentes em nível de *kernel*.

3.2.1 Estruturas de Dados

A implementação atual do protocolo *MPTCP* no *kernel* do *Linux* e que atualmente é um *fork*, pois não está na *mainline*, pode ser dividida em duas camadas. A camada *MPTCP* e a de subfluxos representada pela camada de subfluxo. A camada *MPTCP* é também conhecida por ser a camada *metasocket*, representando o estado de máquina de cada conexão *MPTCP*. O estado de uma conexão *MPTCP* completa é composto por várias estruturas ligadas conforme visto na Figura 7. Ela também é dividida em duas partes. Na primeira parte, as estruturas correspondem a camada *MPTCP* e na segunda, a estrutura representa a camada de subfluxo.

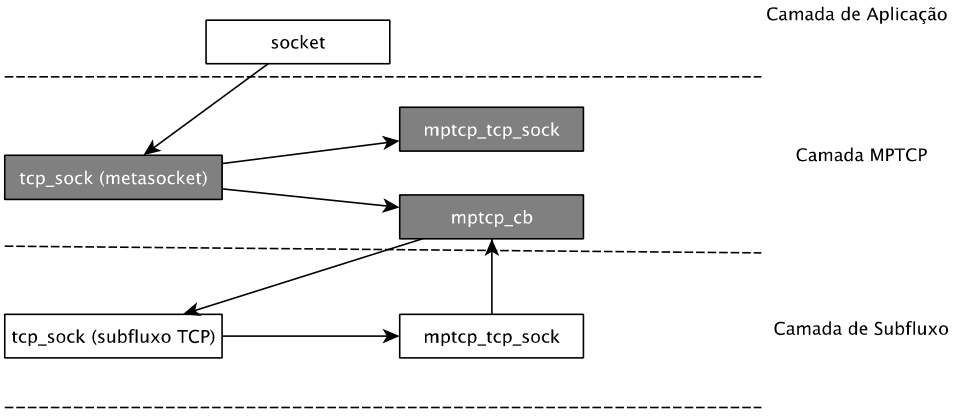

Figura 7 – Diagrama da Divisão por Camadas da Implementação MPTCP no Linux.

O *Kernel* do *Linux* mantém o estado da conexão *TCP* em uma estrutura chamada *tcp_sock*. O protocolo *MPTCP* precisa armazenar sua informação adicional em cada subfluxo *TCP*. Para evitar uma utilização desnecessária de memória da estrutura *tcp_sock*, uma nova estrutura chamada *mptcp_tcp_sock* é adicionada e ligada a estrutura *tcp_sock*. Todo subfluxo *MPTCP*, assim como o *metasocket*, contém um ponteiro para a estrutura *mptcp_tcp_sock* que foi adicionada. Essa estrutura contém algumas informações sobre o subfluxo, como por exemplo, o identificador de endereço correspondente ao subfluxo, necessário para adicionar ou remover novos subfluxos.

A camada *MPTCP* implementa uma máquina de estado *TCP* completa. Sendo assim, o mesmo *tcp_sock*, em conjunto com a estrutura *mptcp_tcp_sock* é utilizado para armazenar esse estado. Adicionalmente, informações gerais sobre a conexão devem ser mantidas. Novamente, com a intenção de se evitar uma utilização desnecessária de memória do protocolo *TCP*, é criada uma estrutura de dados chamada *mptcp_cb*, representando o controle de *buffer* do protocolo *MPTCP*. Essa estrutura é utilizada para guardar informações de apontamento para os subfluxos TCP existentes e a lista de endereços *IP* utilizados pela conexão.

3.2.2 Manipulação de Subfluxos MPTCP

Considerando que o subfluxo inicial foi corretamente estabelecido e dados já estão sendo trocados, subfluxos adicionais podem ser criados. A implementação atual do *MPTCP* permite somente que o cliente possa criar novos subfluxos. Isso ocorre para impedir que múltiplos subfluxos possam ser criados para o mesmo par de endereços *IP*, como por exemplo, se o cliente e o servidor decidirem iniciar um subfluxo ao mesmo tempo. A forma como os subfluxos são criados internamente dentro do protocolo é explicada nessa seção através dos pontos de vista do lado do cliente e do servidor.

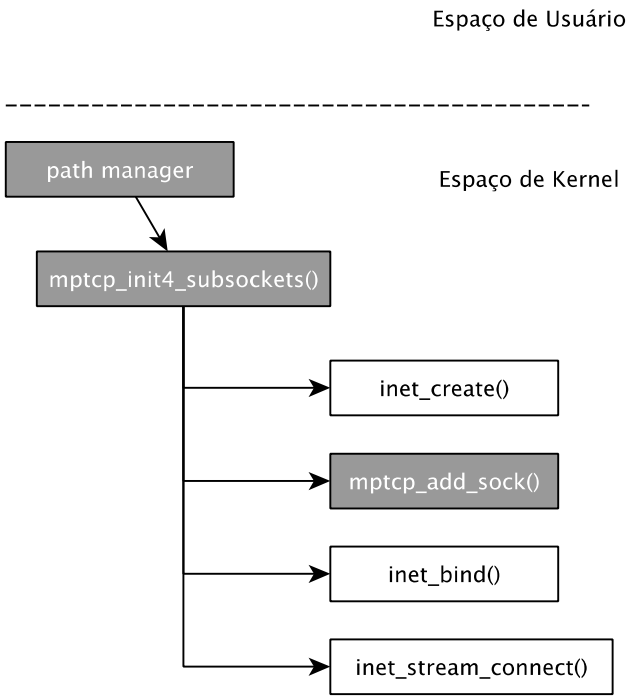

Figura 8 – Criação do processo de subfluxo a partir da função mptcp_init4_subsockets().

3.2.2.1 Funcionamento do MPTCP no Lado do Cliente

A implementação do protocolo *MPTCP* necessita de uma lógica que ajude a decidir quando um novo subfluxo deve ser criado. Esta lógica pode ser influenciada por alguns fatores diferentes. Ela pode tentar estabelecer múltiplas conexões utilizando todos os *IPs* disponíveis nas interfaces existentes ou simplesmente criar vários subfluxos utilizando o mesmo par de endereços *IP* para serem espalhados pela rede utilizando o protocolo *ECMP*. Outros tipos de lógicas podem ser criadas, como por exemplo, a criação de conexões que utilizam as propriedades das interfaces dos dispositivos. Para acomodar cada um desses casos de uso diferentes, são implementados os gerenciadores de caminhos *(path managers)*. O gerenciador de caminhos é chamado pela implementação do *MPTCP* através de chamadas de sistema que são registradas no momento de criação da conexão. Com isso, o gerenciador de caminhos tem controle sobre os eventos e pode realizar ações de controle, como por exemplo, garantir o estabelecimento com sucesso do subfluxo inicial ou a recepção de um endereço remoto através da opção de `ADD_ADDR`. O gerenciador de caminhos pode controlar a criação de novos subfluxos através da chamada da função `mptcp_init4_subsockets()` encontrada em `mptcp/mptcp_ipv4.c`, que será descrita a seguir.

O processo de criação de um novo subfluxo no lado do cliente envolve alguns passos, da mesma forma que a criação de uma conexão *TCP* por uma aplicação do lado do cliente,

conforme mostra a Figura 8. Primeiro, um *tcp_sock* é criado e vinculado a uma interface específica utilizando as funções `inet_create()` e `inet_bind()`. Em seguida, para efetuar o envio do pacote *SYN*, a função `inet_stream_connect()` é chamada. A pilha *TCP* deve incluir a opção `MP_JOIN` dentro do pacote *SYN* no campo reservado, juntamente com um *token* gerado e um número randômico.

Para permitir que a pilha *TCP* possa recuperar o *token* e adicionar a opção de *MP_JOIN*, é necessária a criação e alocação de uma estrutura *mptcp_tcp_sock* antes do envio do pacote *SYN*. Para permitir essa criação, a função `mptcp_add_sock()` precisa ser chamada, diferente do estabelecimento do subfluxo inicial, onde a função `mptcp_add_sock()` é chamada apenas no final do processo. Nesse caso, o argumento de alocação de estruturas auxiliares para reduzir o consumo de memória nas conexões *TCP* citado na Seção 3.2.1 não serve mais para os subfluxos adicionais em uma conexão *MPTCP* existente. O novo subfluxo apenas será referenciado pela pilha *TCP* se o pacote *SYN/ACK* tiver a opção de *MP_JOIN*. Caso contrário, ele será descartado pela pilha *MPTCP*. Além disso, podem surgir problemas de falha de alocação de memória se apenas as estruturas para a recepção do pacote *SYN/ACK* forem alocadas, quando esses pacotes forem descartados. Esse custo é aceito no processo de criação do subfluxo inicial com a finalidade de reduzir o impacto de utilização de memória nas estruturas originais do TCP, porém, gera um custo desnecessário para os novos subfluxos adicionados.

Quando o servidor responde com um pacote *SYN/ACK*, incluindo a opção de *MP_JOIN*, o cliente deve efetuar a verificação do *HMAC*. Caso ele esteja incorreto, o cliente responde com um pacote *RST*, conforme especificado na *RFC6824* e fecha o novo subfluxo. Podemos observar o fluxo de sequência conforme foi descrito para a criação de subfluxo através da Figura 9.

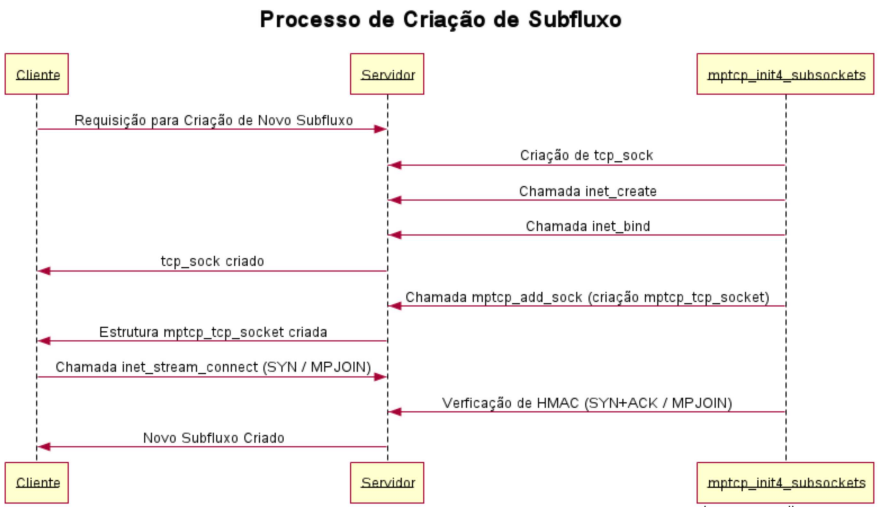

Figura 9 – Diagrama de Sequência para Criação de Subfluxo.

O processo de *three-way handshake* dos novos subfluxos é diferente no *MPTCP* quando comparado ao *TCP*. No caso do *MPTCP*, é exigida uma transmissão confiável do terceiro pacote *ACK* para garantir a entrega correta do *HMAC* incluso no pacote. Isso implica que uma alteração substancial deve ser feita na implementação contemplando esse recurso, assim como a implementação de um temporizador de retransmissão é necessária para garantir o envio desse terceiro pacote *ACK*. O temporizador de retransmissão do protocolo *TCP* não pode ser utilizado porque nesse caso, o pacote com a flag de *ACK* é tratado pela pilha *MPTCP*. A implementação atual utiliza um temporizador de retransmissão adicional para permitir que o terceiro pacote *ACK* possa ser retransmitido até que o servidor responda com uma confirmação para sinalizar a recepção e validação do terceiro pacote *ACK* e o seu *HMAC*. Além disso, qualquer transmissão de dados deve ser impedida nesse subfluxo enquanto o pacote *ACK* final não for recebido pelo cliente. Isso é controlado através de uma *flag* de pré-estabelecimento na estrutura *mptcp_tcp_sock* que deve ser verificada toda vez que o agendador do *MPTCP* decide transmitir dados em um subfluxo. Se essa *flag* estiver definida como 1, o agendador não pode transmitir dados nesse subfluxo.

3.2.2.2 Funcionamento do MPTCP no Lado do Servidor

No lado do servidor, o pacote *SYN* deve ser ligado à conexão *MPTCP* baseada no *token* incluso na opção de *MP_JOIN*. Uma entrada *SYN + MP_JOIN* deve ter uma porta de destino diferente da que foi utilizada no subfluxo inicial ou utilizar um endereço *IP* diferente. A implementação do *MPTCP* deve verificar se cada um dos pacotes *SYN* contém a opção *MP_JOIN* e se o *token* indicado corresponde a conexão existente. Neste caso, o processamento do pacote *SYN* deve ser finalizado de forma que não seja necessária a criação de uma nova conexão *TCP*, mas que esse novo subfluxo esteja ligado a uma conexão *MPTCP* já existente e que seja referenciada pelo *token* indicado. Assim, a opção *MP_JOIN* inserida no campo *TCP options* terá prioridade quando corresponder a um *socket* já existente.

Para redirecionar o processamento do pacote *SYN* para uma conexão *MPTCP* existente, a implementação deve ser capaz de analisar e verificar o campo *TCP options* de cada pacote *SYN*, procurando por uma opção *MP_JOIN*. Somente após esse processo inicial de análise e verificação, o controle de fluxo deve ser redirecionado para a pilha *TCP*, permitindo assim o redirecionamento para o *socket* apropriado. Esse processo de análise e verificação acaba apresentando uma penalidade de processamento em um sistema operacional genérico, pois, provavelmente a maiora dos pacotes *SYN* não terão a opção *MP_JOIN*.

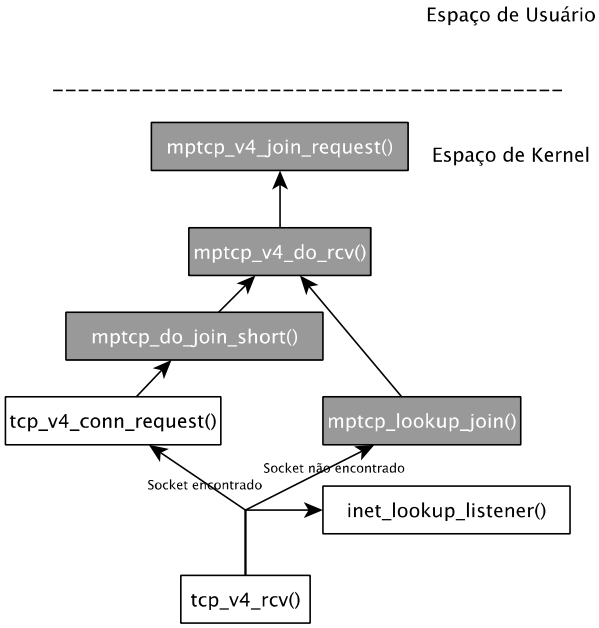

Figura 10 – Diagrama de processamento do SYN + MP_JOIN em um socket.

O processo de verificação e análise é realizado a partir do fluxo de controle demonstrado na Figura 10. Primeiro, o pacote *SYN* deve ter uma porta de destino correspondente ao *socket* em escuta. A função `inet_lookup_listener()` é chamada para verificar qual *socket* deve receber a conexão *TCP* que foi referenciada pelo pacote *SYN*. O campo *TCP options* deve ser analisado utilizando a função `tcp_v4_conn_request()`. Com isso, a pilha *TCP* verifica se a opção de `MP_JOIN` está inclusa no pacote `SYN`, obrigando o redirecionamento do pacote para a conexão *MPTCP*. A função `mptcp_do_join_short()` deve procurar na tabela de conexões *MPTCP* a conexão que está ligada com o *token* indicado e deve então criar um *socket* de requisição e emitir um pacote *SYN/ACK*. A Figura 10 também mostra outro rastreamento de chamada quando nenhum *socket* de escuta foi encontrado para o pacote *SYN* a ser processado. Neste caso, a implementação é forçada a fazer a verificação e análise do campo *TCP options* de forma separada chamando a função *mptcp_lookup_join()* que será responsável por procurar o *MP_JOIN* e verificar a existência de uma conexão *MPTCP* referente na tabela de conexões *MPTCP* existentes.

O servidor então cria através da chamada da função `mptcp_v4_join_request()` um *socket* de solicitação que armazena as informações adicionais sobre o estado atual do subfluxo. Esse *socket* de solicitação deve ser armazenado como parte da conexão *MPTCP* mas também dentro de uma tabela *hash* que serve como identificador para as conexões *MPTCP* existentes. Isto é necessário, pois, durante o processo de *three-way handshake*, o terceiro pacote *ACK* solicita a criação de um *socket TCP* no lado do servidor. O *host* encontra o *socket* de requisição correspondente ao pacote *ACK* verificando a relação

existente com o *socket* de entrada, que armazena esse *socket* de requisição em uma lista. Mas, durante o processo de estabelecimento de um novo subfluxo, não há um *socket* existente aguardando esse socket de requisição. Isso acontece para cada pacote *ACK* que não possui uma relação com um *socket* estabelecido e será necessária uma verificação para saber se existe uma requisição de *socket* pendente esperando pelo pacote *ACK* final em um processo de *three-way handshake* efetuado pela chamada de um *MP_JOIN*.

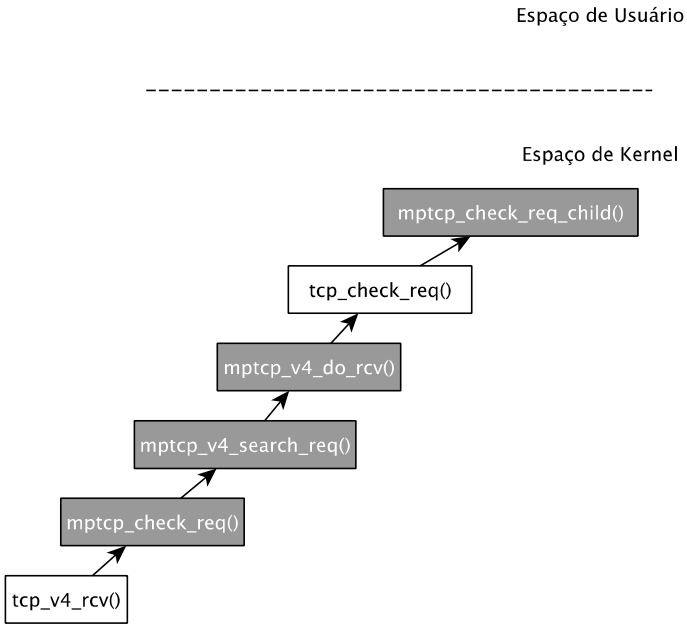

Figura 11 – Verificação de socket de requisição para cada pacote ACK.

A implementação atual tenta fazer uma aproximação através de força-bruta mostrada na Figura 11. A chamada para a função `mptcp_check_req()` verifica e analisa o pacote *ACK* de entrada, que utiliza a função `mptcp_v4_search_req()` para procurar dentro da tabela de conexões a existência de uma requisição de socket pendente. Essa busca é finalizada quando todos os segmentos do pacote *ACK* não são encontrados em um *socket* estabelecido ou existente. Se um *socket* de requisição é encontrado, as chamadas são redirecionadas para a função `tcp_check_req()`, que então cria as estruturas de socket *TCP* necessárias e continua o processo chamando a função `mptcp_check_req_child()` que finaliza a inicialização e aponta o novo subfluxo para uma conexão *MPTCP* através da chamada da função `mptcp_add_sock()`.

3.3 Desenvolvimento da API

O protocolo *MPTCP* (RAICIU; HANDLEY; BONAVENTURE, 2013) possui uma variedade de técnicas para habilitar a transferência de dados entre múltiplos caminhos.

Informações mais detalhadas sobre o funcionamento do protocolo podem ser encontradas em (RAICIU; HANDLEY; BONAVENTURE, 2013; RAICIU et al., 2012). Nessa seção, pretendemos discutir apenas as interações entre uma aplicação e o protocolo *MPTCP* através da *API (RFC 6897)* e deixamos os detalhes de funcionamento do protocolo pertinentes ao entendimento desse trabalho na Seção 2.1.3.

Sobre o ponto de vista de arquitetura, o protocolo *MPTCP* pode ser visto como um conjunto de conexões *TCP* que são chamadas de subfluxos, que são agrupadas e gerenciadas através dos dois pontos em uma conexão *MPTCP* estabelecida. Esse conjunto de subfluxos não é estático e pode ser estabelecido ou terminado durante o tempo de vida de uma conexão *MPTCP*.

Na implementação atual do protocolo, as duas principais estruturas existentes para controle da conexão *MPTCP* são os *metasockets* e os *sockets* de subfluxo. O *metasocket* é a estrutura que faz a ponte entre o socket principal *MPTCP* e o subfluxo *TCP* para efetuar a troca de informação de controle entre as aplicações, ou seja, todos os dados enviados e recebidos através de uma conexão *MPTCP* passam por essa estrutura e por isso ela é visível e possui acesso através da aplicação. Ele também contém os ponteiros necessários para acessar as estruturas internas do *MPTCP*, incluindo uma lista ligada dos subfluxos estabelecidos.

O *socket* de subfluxo ou *subflow socket* é a estrutura interna que faz o controle de criação e remoção de subfluxos, que antes não podia ser acessada pela aplicação, conforme visto na Seção 3.2.1. Por isso, temos a necessidade da criação de uma *API* que possa acessar essa estrutura de controle com a finalidade de permitir que além das informações já obtidas sobre a conexão através do *metasocket*, seja possível também um controle dinâmico de criação, remoção e outras features que foram mostradas na Tabela 1. Para que isso seja possível, é necessária a exposição dessa estrutura *subflow socket* através da *API* desenvolvida.

Para isso, adicionamos as chamadas de interface para o protocolo *MPTCP* dentro da interface de *socket* no *kernel* do *Linux* que são representadas pelas funções `do_tcp_setsockopt`, no caso das funções de alteração de dados, e `do_tcp_getsockopt`, no caso das funções de recuperação de dados. Tais funções que definem o ponto de entrada da interface de *sockets* são implementadas no arquivo `tcp.c` localizado em *net/ipv4* dentro da árvore de código do *kernel*. Logo, a chamada inicial da função que adiciona subfluxos é feita nesse ponto de entrada e o funcionamento geral da *API* é ilustrado através da Figura 12.

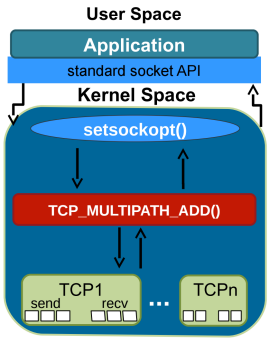

Figura 12 – Criação de Novos Subfluxos utilizando a API.

Atualmente, o conjunto de operações disponíveis na *API* desenvolvida e que estendem a *API* de socket no *kernel* são: `TCP_MULTIPATH_ENABLED` (para ligar o protocolo via aplicação), `TCP_MULTIPATH_CONNID` (obter o ID da conexão TCP) e `TCP_MULTIPATH_ADD` (adiciona um novo subfluxo).

Código 3.1 – Pontos de entrada na API de Socket em Nível de Kernel para a API implementada.

```
static int do_tcp_getsockopt(struct sock *sk, int level,
      int optima, char __user *optval, int __user *optlen)
{
   case TCP_MULTIPATH_ENABLED:
      Tratamento de valores de flag ligado/desligado
        Chamada de sistema para ligar e desligar passando o valor recebido
   case TCP_MULTIPATH_CONNID:
      Recebe identificador da conexao
        Retorna a estrutura preenchida de acordo com o identificador recebido
}

static int do_tcp_setsockopt(struct sock *sk, int level,
      int optima, char __user *optval, int __user *optlen)
{
   case TCP_MULTIPATH_ADD:
      Recebe e trata valores para adicionar subfluxos
        Chamada da funcao mptcp_add_subflow para adicionar subfluxos passando
            valores recebidos
}
```

Para adicionar novos subfluxos, criamos uma função dentro do arquivo `tcp.c` localizado em `net/ipv4` que tem a assinatura descrita no Código 3.2 e que conforme

verificado, recebe uma estrutura de *socket* que será manipulada e a quantidade de subfluxos definida através da chamada em nível de usuário. Essa função foi criada para permitir que o controle, antes feito apenas pelo gerenciador de caminhos possa ser também efetuado a partir de uma chamada via aplicação que utilize a opção de socket `TCP_MULTIPATH_ADD`.

Código 3.2 – Assinatura da Função Desenvolvida na API para adicionar subfluxos.

```
static int mptcp_add_subflow(struct sock *sk, char *num_subflows)
```

A função criada executa alguns passos, para controlar o processo de criação de subfluxos e que antes eram efetuados diretamente pelo gerenciador de caminhos conforme verificado na Seção 3.2.2 e que são descritos na Figura 13.

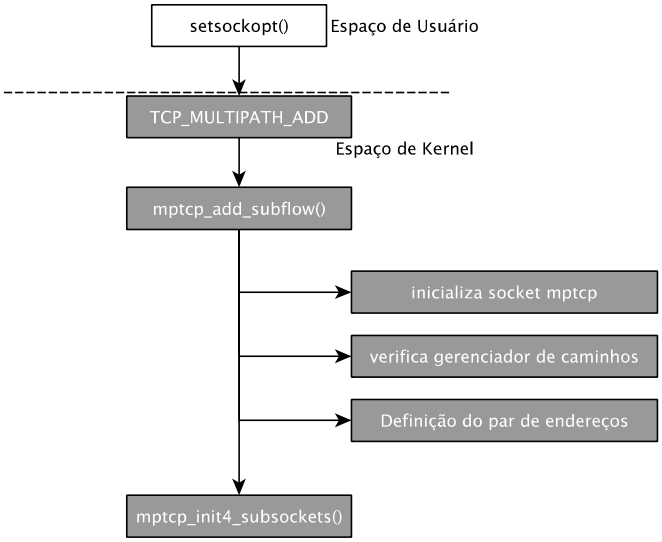

Figura 13 – Funcionamento da chamada da API implementada que adiciona subfluxo.

A flag `TCP_MULTIPATH_ADD` chamada a partir da opção `setsockopt()` em nível de usuário passa os parâmetros necessários para a chamada da função `mptcp_add_subflow()`. A função quando chamada, efetua a inicialização do *metasocket* e *socket MPTCP*, necessários para o controle do subfluxo que será criado. Uma verificação do gerenciador de caminhos que define a lógica que será utilizada pelo protocolo é necessária, pois de acordo com o modelo de lógica escolhido, o processo de criação de subfluxos pode ser diferente. Após a verficação, uma estrutura referente ao par de endereços é definida e inicializada e após esse processo, a função `mptcp_init4_subsockets()` é chamada e tem o seu funcionamento detalhado descrito nas Seções 3.2.2.1 e 3.2.2.2

Para efetuar a chamada a partir das aplicações em nível de usuário através da *API* de *socket*, utilizamos a seguinte sintaxe no código para adicionar subfluxos que é ilustrada através do Código 3.3.

Código 3.3 – Exemplo de chamada em nível de usuário em uma aplicação.

```
error = getsockopt(sockfd, IPPROTO_TCP, TCP_MULTIPATH_CONNID,
      &sub_sso, optlen);

error = setsockopt(sockfd, IPPROTO_TCP, TCP_MULTIPATH_ADD,
               &sub_sso, optlen);
```

No código acima, *sockfd* é o descritor de arquivo necessário para o *socket* e *sub_sso* é a estrutura de dados necessária para a utilização dos subfluxos no protocolo *MPTCP* e *optlen* é o tamanho da estrutura passada.

Para validar a utilização da *API*, desenvolvemos uma aplicação *HTTP* em linguagem C que primeiramente detecta a presença de suporte ao *MPTCP* e estabelece uma conexão em um servidor web requisitando um arquivo de tamanho consideravelmente grande, com a intenção de gerar um fluxo que possa ser monitorado. Quando esse fluxo que está sendo monitorado na rede pela aplicação através do protocolo *MPTCP* atinge a taxa de *threshold*, que neste trabalho é 100MB de dados, a aplicação estabelece a criação de um novo subfluxo para que esse *stream* possa ser dividido. Os detalhes sobre o caso de uso se encontram no próximo capítulo.

Finalização do Capítulo

Neste capítulo, apresentamos uma visão geral sobre a *API* de *socket* existente na maioria dos sistemas operacionais. Demonstramos também uma visão interna do funcionamento do protocolo *MPTCP* necessária para o funcionamento da *API* descrita na *RFC 6897* e do caso de uso aplicado a ela. Finalizamos com a explicação do processo de desenvolvimento da *API* em espaço de *kernel* e a utilização da mesma em espaço de usuário.

4 Resultados e Análise

4.1 Caso de uso da RFC 6897: tratamento de fluxos elefantes

Como caso de uso, desenvolvemos uma aplicação ***context awareness*** utilizando a *API* implementada e que tem a capacidade de verificar se um determinado fluxo de saída contém características de um fluxo elefante. Quando um fluxo elefante é detectado, a aplicação solicita a criação de subfluxos conforme a necessidade. A validação do algoritmo é determinada através de valores considerados mínimo e máximo para atuação do mesmo. Esses valores são utilizados como parâmetros limites de atuação para a detecção de um fluxo elefante, que hoje é definido a partir de 100MB até o limite de 300MB, abrindo um novo subfluxo a cada 50MB até atingir o limite máximo de 4 subfluxos, baseando se na definição de fluxo elefante encontrada em (CURTIS; KIM; YALAGANDULA, 2011).

A aplicação desenvolvida consiste em baixar arquivos grandes de um servidor *web* utilizando uma conexão *MPTCP* pré-estabelecida para controle e monitoramento inicial do fluxo até que o valor estabelecido como limite para um fluxo elefante seja atingido. Quando o valor definido para o limite é alcançado, a aplicação consegue abrir novos subfluxos, permitindo assim que o fluxo inicial seja quebrado em fluxos camundongos. Esses fluxos camundongos são distribuídos através dos subfluxos criados e múltiplos caminhos existentes na rede conforme observado na Figura 14.

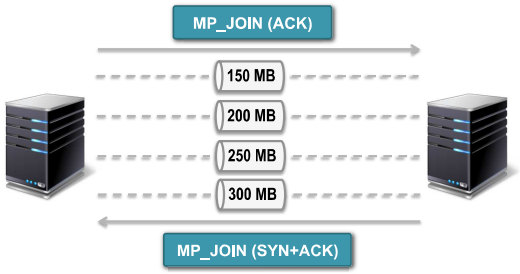

Figura 14 – Visão de Criação e Controle de Subfluxos pelo Módulo.

A aplicação depende de um *kernel* do *linux* que tenha suporte ao *MPTCP* [1]. Importante lembrar que para o funcionamento do módulo se faz necessário o controle de conexões e criação de subfluxos gerados aproveitando um *socket* aberto em uma única *interface* através de rotinas que serão controladas pela aplicação, justificando a necessidade de implementação da *API* que desenvolvemos (SILVA; FERLIN; VERDI, 2016).

[1] **Para adicionar o suporte necessário ao kernel, recomenda-se utilizar as instruções disponíveis no site do Multipath TCP em http://multipath-tcp.org/pmwiki.php/Users/HowToInstallMPTCP?**

4.2 Testes Efetuados

Em nosso caso de testes, montamos todo o ambiente em um servidor *Dell PowerEdge R420* com 2 processadores *Intel Xeon E5-2430* operando com *clock* de *2.2GHz* e com 48 GB de memória, utilizando o *KVM* em um *Ubuntu Linux Server v.14.04* com duas máquinas virtuais para testes, representando o cliente e o servidor, ambas com *Ubuntu Linux Server v. 14.04* e *kernel* com suporte ao *MPTCP* habilitado. Para montar a topologia de rede, utilizamos roteadores *Mikrotik* virtualizados com *RouterOS v. 6.34.4* e com suporte ao protocolo *ECMP* habilitado. O *ECMP* foi escolhido para ser o protocolo de roteamento uma vez que tem sido a solução mais utilizada em redes de *datacenters* para realização de balanceamento de carga entre as diferentes rotas.

Uma vantagem obtida com utilização do *MPTCP*, devido a quebra de um fluxo único em múltiplos fluxos, é a possibilidade de encaminhamento do tráfego por diversas rotas diferentes, de forma concorrente. A Figura 15 ilustra a topologia do ambiente utilizado para os testes.

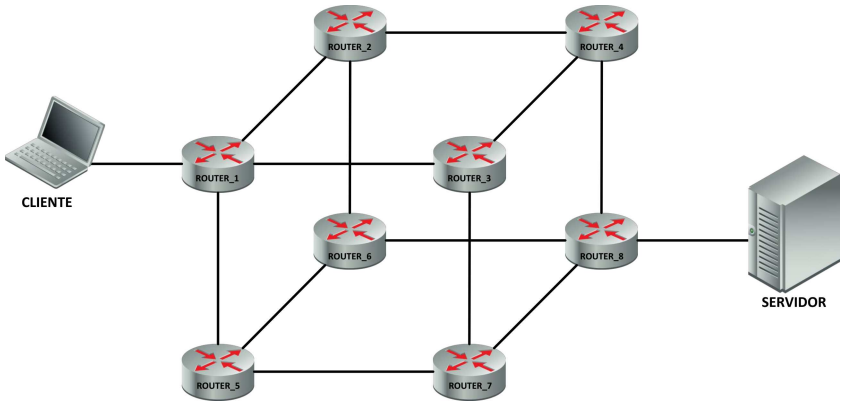

Figura 15 – Topologia de Rede Cubica Utilizada para os Testes.

A topologia em cubo permite o estabelecimento de três caminhos disjuntos de saída nos *links primários* do roteador de acesso. Considerando que para cada um destes caminhos haverão dois outros caminhos possíveis através dos *links secundários*, teremos um total de seis caminhos disjuntos com rotas de mesmo custo entre os hosts conectados aos roteadores em vértices opostos. Essa característica permite a criação de uma ambiente satisfatório para a utilização do protocolo *ECMP*, necessário para demonstrar os resultados.

Como referência, adotamos a nomenclatura de *roteadores de acesso* para os roteadores que possuem *links* com os *hosts* de teste, num total de 2 roteadores *ROUTER_1* e *ROUTER_8*. Os outros roteadores, foram denominados *roteadores de núcleo*, a saber: *ROUTER_2*, *ROUTER_3*, *ROUTER_4*, *ROUTER_5*, *ROUTER_6* e *ROUTER_7*. Os links entre os roteadores de acesso e os roteadores de núcleo foram denominados *links*

primários e os links que conectam os roteadores de núcleo entre sí, foram denominados *links secundários*.

Para avaliar o ganho de execução por abertura de subfluxos, utilizamos a aplicação *HTTP* que foi desenvolvida usando a *API* implementada nessa dissertação e calculamos o tempo de execução médio de 10 rodadas por número de subfluxos, onde limitamos a abertura iniciando por 3 subfluxos até 7 subfluxos. Em cada cenário efetuamos 3 transferências de arquivos de tamanhos diferentes: 300MB, 600MB e 1GB, o que gerou os resultados apresentados a seguir.

Na Figura 16, podemos observar a evolução dos tempos de transferência (*Flow Completion Times- FCTs*) para os arquivos de 300MB, 600MB e 1GB. Com o arquivo de 300MB, percebemos que o *FCT* diminui até o limite de 5 subfluxos, porém, a partir de 6 subfluxos abertos podemos observar o início do aumento do *FCT*. No segundo cenário, com o arquivo de 600MB, observamos um ganho até a abertura de 5 subfluxos, sendo que a partir de 6 subfluxos o *FCT* se estabiliza e mantém o mesmo valor para 7 subfluxos. No terceiro cenário, com o arquivo de 1GB, também observamos ganho até a abertura de 5 subfluxos e, a partir deste ponto, o *FCT* se estabiliza como no caso anterior com 600MB.

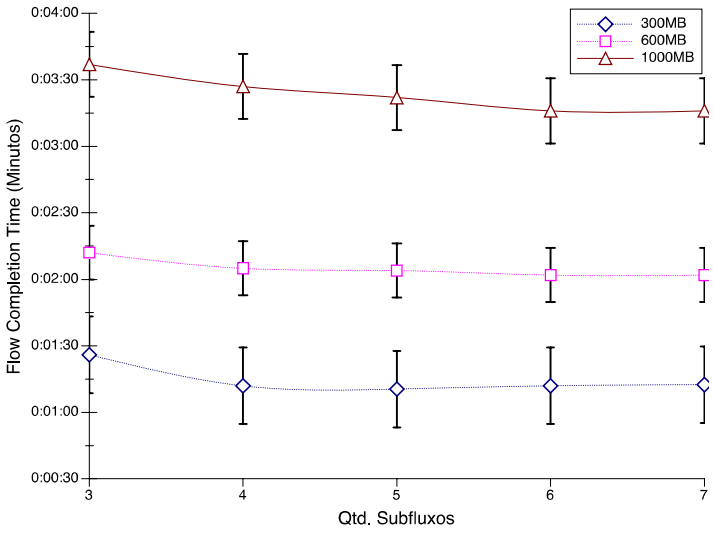

Figura 16 – Gráfico de Tempo de Execução Médio por Quantidade de Subfluxos Abertos com Arquivos de 300MB, 600MB e 1GB.

O *FCT* varia em função da quantidade de subfluxos e a quantidade de caminhos (disjuntos ou não) na topologia da rede. Claramente, quanto mais caminhos disponíveis, de preferência disjuntos, melhor será a distribuição dos subfluxos pelo *ECMP* na rede. A conclusão com os resultados apresentados na Figura 16 é que a abertura de fluxos deve levar em consideração a topologia da rede, mais precisamente a quantidade de caminhos disponíveis. Caso contrário, o *overhead* causado pelo *MPTCP* pode prejudicar o desempenho *(FCT)* final.

Através da execução desses testes, conseguimos visualizar que o ganho mais significativo para a topologia escolhida é quando estabelecemos o limite de abertura de 3 para 4 subfluxos, definindo assim este número para os testes da aplicação *HTTP* que virão a seguir conforme Figuras 17 e 18.

Nos testes executados foram propostos dois cenários no mesmo ambiente, variando a ocupação de banda nas rotas e também o uso ou não do *MPTCP*. A implementação *TCP CUBIC* foi utilizada para os casos sem *MPTCP*. A ocupação de banda (tráfego de fundo) foi realizada através de tráfego *TCP* com o gerador de tráfego *iperf*. A largura de banda dos *links* entre os roteadores foi intencionalmente configurada em *10Mbps* para que não houvesse interferência nos resultados devido a gargalos de processamento nestes roteadores ou nos *hosts*. Os *links* entre os roteadores e os *hosts* foram configurados com a mesma largura de banda *(10 Mbps)*. Para cada transferência de arquivo, 10 execuções foram realizadas e as médias são mostradas nos gráficos.

No primeiro cenário, apresentado na Figura 17, nenhum tráfego de fundo foi inserido nos *links*. Conforme podemos observar na transferência do arquivo de 600MB, por exemplo, o recurso de abertura de subfluxos de modo dinâmico tem uma variação no ganho de tempo significativa, sendo que a aplicação ciente de contexto precisou de 3 minutos e 56 segundos para transferir todo o arquivo e a aplicação que não usa *MPTCP* precisou de 8 minutos e 56 segundos, uma diferença de 125%. Esta mesma análise pode ser feita com os arquivos de 100MB, 300MB e 1GB.

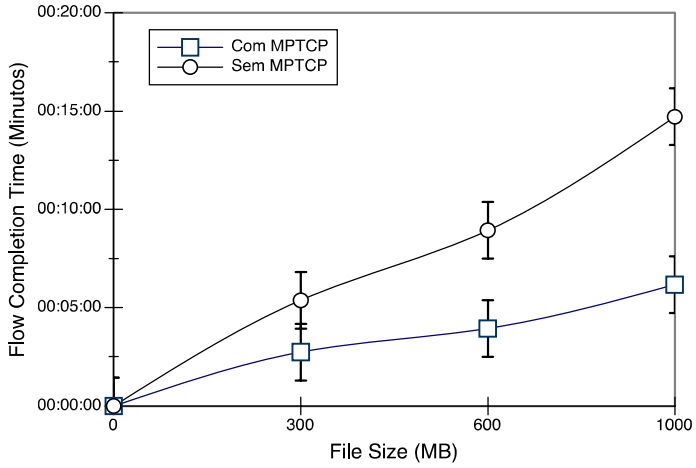

Figura 17 – Gráfico Comparativo de Taxa de Transferência do MPTCP X TCP (Sem Tráfego).

No segundo cenário, com o objetivo de verificar o comportamento do tráfego *MPTCP* ao compartilhar a banda de um *link* com outros fluxos, gerou-se tráfego *TCP* suficiente para ocupar 50% da largura de banda disponível em cada um dos seis caminhos disjuntos usados para o teste. Em seguida, utilizou-se o mesmo procedimento descrito

no primeiro cenário para a transferência dos arquivos. Os resultados obtidos podem ser verificados através do gráfico apresentado na Figura 18, demonstrando um ganho de tempo significativo da aplicação *awareness* que processou a transferência do arquivo de 600MB em 4 minutos e 44 segundos enquanto que a aplicação sem *MPTCP* precisou de 16 minutos e 46 segundos. Há um ganho ainda maior para o arquivo de 1GB que resultou em 7 minutos e 57 segundos para a aplicação *awareness* contra 24 minutos e 46 segundos sem o suporte *awareness*.

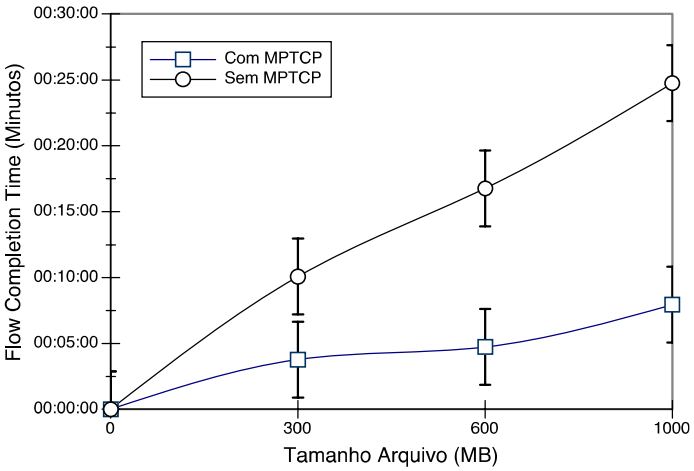

Figura 18 – Gráfico Comparativo de Taxa de Transferência do MPTCP X TCP (50% de Ocupação de Banda).

Através dos testes efetuados, demonstramos os benefícios em se utilizar o protocolo *Multipath TCP (MPTCP)* como alternativa para a criação de aplicações conscientes *(awareness)*, utilizando como caso de uso o tratamento de fluxos elefantes. Com isso, é possível que as aplicações possam se aproveitar dos benefícios oferecidos pelo *MPTCP* através da *API* implementada neste trabalho. As regras de abertura dos subfluxos passam a ser definidas pelo desenvolvedor da aplicação e podem considerar aspectos antes não considerados. Diversos outros casos de uso podem ser suportados, como por exemplo, a criação de balanceadores de carga em nível de aplicação.

Conclusão e Trabalhos Futuros

O *Multipath TCP (MPTCP)* foi desenvolvido para que fluxos *TCP* tradicionais usem com maior eficiência as diferentes interfaces de rede dos dispositivos, assim como os diferentes caminhos disponíveis em uma topologia de rede.

Entretanto, todo potencial fornecido pelo *MPTCP* só pode ser explorado se as aplicações fizerem uso do protocolo de maneira consciente (*MPTCP-awareness*), ideia esta especificada na *RFC 6897* que propõe uma *API* para que um controle fino dos subfluxos *MPTCP* seja realizado pelas aplicações que desejem usar tal protocolo.

Este trabalho, portanto, apresenta uma implementação inicial desta *RFC* em nível de *kernel* para que aplicações façam uso do protocolo *MPTCP* conforme suas necessidades, definindo quando adicionar e remover fluxos assim como quando ligar e desligar o protocolo e como exemplo, transformou uma aplicação em *MPTCP-aware* utilizando o caso de uso de tratamento de fluxos elefantes em *end-host* para a validação da implementação.

Com esta implementação, os desenvolvedores e usuários do protocolo *MPTCP* podem encontrar caminhos para estender as suas aplicações de forma a utilizar os recursos existentes no protocolo com a finalidade de se obter os melhores resultados em sua utilização.

Como trabalhos futuros, o desenvolvimento da API deve ser continuado, pois os items que foram aqui desenvolvidos servem para atender os requisitos necessários para a validação desse trabalho de dissertação. Sendo assim, as funções restantes `TCP_MULTIPATH_SUBFLOWS` e `TCP_MULTIPATH_REMOVE` devem ser implementadas, fornecendo assim uma versão completa da *API* descrita na *RFC 6897*.

A *API* também pode ser utilizada para complementar outros casos de uso, como por exemplo, a possibilidade de utilizar o *MPTCP* para permitir a realização de balanceamento de carga por aplicação e de forma dinâmica, tema que vem sido discutido no *MPTCP Working Group* do *IETF*. Outra idéia de trabalho futuro também seria a portabilidade dessa *API* para outros Sistemas Operacionais, como por exemplo, os *BSDs* que já possuem uma implementação inicial do protocolo *MPTCP*. Por fim, há trabalhos que apontam o *MPTCP* como uma alternativa para aumentar taxas de transferência em redes veículares (WILLIAMS et al., 2014). Tal ideia pode também ser explorada com o uso da *API* implementada nesta dissertação.

Referências

AL-FARES, M. et al. Hedera: Dynamic flow scheduling for data center networks. In: *NSDI*. [S.l.: s.n.], 2010. v. 10, p. 19–33. Citado 3 vezes nas páginas 39, 42 e 44.

ARYE, M. et al. A formally-verified migration protocol for mobile, multi-homed hosts. In: IEEE. *2012 20th IEEE International Conference on Network Protocols (ICNP)*. [S.l.], 2012. p. 1–12. Citado na página 29.

BOCCASSI, L.; FAYED, M. M.; MARINA, M. K. Binder: a system to aggregate multiple internet gateways in community networks. In: ACM. *Proceedings of the 2013 ACM MobiCom workshop on Lowest cost denominator networking for universal access*. [S.l.], 2013. p. 3–8. Citado na página 40.

CASADO, M. *Of Mice and Elephants*. 2013. Disponível em: <http://http://networkheresy.com/2013/11/01/of-mice-and-elephants/>. Citado 2 vezes nas páginas 27 e 39.

CURTIS, A. R.; KIM, W.; YALAGANDULA, P. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In: IEEE. *INFOCOM, 2011 Proceedings IEEE*. [S.l.], 2011. p. 1629–1637. Citado 4 vezes nas páginas 39, 43, 44 e 57.

CURTIS, A. R. et al. Devoflow: scaling flow management for high-performance networks. In: ACM. *ACM SIGCOMM Computer Communication Review*. [S.l.], 2011. v. 41, n. 4, p. 254–265. Citado 2 vezes nas páginas 42 e 44.

DOURISH, P.; BELLOTTI, V. Awareness and coordination in shared workspaces. In: ACM. *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*. [S.l.], 1992. p. 107–114. Citado na página 38.

DREIBHOLZ, T. et al. Implementation and evaluation of concurrent multipath transfer for SCTP in the INET framework. In: ICST (INSTITUTE FOR COMPUTER SCIENCES, SOCIAL-INFORMATICS AND TELECOMMUNICATIONS ENGINEERING). *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*. [S.l.], 2010. p. 15. Citado na página 30.

EARDLEY, P. Survey of MPTCP implementations. 2013. Citado na página 35.

FORD, A. et al. TCP extensions for multipath operation with multiple addresses. *IETF MPTCP proposal-http://tools. ietf. org/id/draft-ford-mptcp-multiaddressed-03. txt*, 2009. Citado 2 vezes nas páginas 24 e 33.

FORD, A. et al. Architectural guidelines for Multipath TCP development. *IETF, Informational RFC*, v. 6182, p. 2070–1721, 2011. Citado na página 32.

GREENBERG, A. et al. Vl2: a scalable and flexible data center network. In: ACM. *ACM SIGCOMM Computer Communication Review*. [S.l.], 2009. v. 39, n. 4, p. 51–62. Citado na página 39.

GRINNEMO, K.-J.; BRUNSTROM, A.; CHENG, J. Using concurrent multipath transfer to improve the SCTP startup behavior for PSTN signaling traffic. In: IEEE. *Advanced Information Networking and Applications Workshops (WAINA), 2014 28th International Conference on*. [S.l.], 2014. p. 772–778. Citado na página 29.

HESMANS, B.; BONAVENTURE, O. An enhanced socket api for multipath tcp. In: ACM. *Proceedings of the 2016 Applied Networking Research Workshop*. [S.l.], 2016. p. 1–6. Citado 2 vezes nas páginas 37 e 41.

HESMANS, B. et al. Smapp: Towards smart Multipath TCP-enabled applications. In: *CoNEXT'15*. [S.l.: s.n.], 2015. Citado na página 41.

HESMANS, B. et al. A first look at real Multipath TCP traffic. In: *Traffic Monitoring and Analysis*. [S.l.]: Springer, 2015. p. 233–246. Citado 2 vezes nas páginas 23 e 32.

LIM, Y.-s. et al. Cross-layer path management in multi-path transport protocol for mobile devices. In: IEEE. *INFOCOM, 2014 Proceedings IEEE*. [S.l.], 2014. p. 1815–1823. Citado na página 40.

LIM, Y.-s. et al. How green is Multipath TCP for mobile devices? In: ACM. *Proceedings of the 4th workshop on All things cellular: operations, applications, & challenges*. [S.l.], 2014. p. 3–8. Citado na página 40.

LU, Y. et al. Elephanttrap: A low cost device for identifying large flows. In: IEEE. *High-Performance Interconnects, 2007. HOTI 2007. 15th Annual IEEE Symposium on*. [S.l.], 2007. p. 99–108. Citado 2 vezes nas páginas 42 e 44.

OH, B.-H.; LEE, J. Constraint-based proactive scheduling for MPTCP in wireless networks. *Computer Networks*, Elsevier, v. 91, p. 548–563, 2015. Citado 2 vezes nas páginas 24 e 32.

PAASCH, C.; BARRE, S. et al. *Multipath TCP in the Linux kernel*. 2013. Citado na página 32.

PAASCH, C.; BONAVENTURE, O. Multipath TCP. *Communications of the ACM*, ACM, v. 57, n. 4, p. 51–57, 2014. Citado 2 vezes nas páginas 23 e 33.

PAASCH, C. et al. Exploring mobile/Wi-Fi handover with Multipath TCP. In: ACM. *Proceedings of the 2012 ACM SIGCOMM workshop on Cellular networks: operations, challenges, and future design*. [S.l.], 2012. p. 31–36. Citado 4 vezes nas páginas 29, 32, 35 e 40.

RAICIU, C. et al. Improving datacenter performance and robustness with Multipath TCP. *ACM SIGCOMM Computer Communication Review*, ACM, v. 41, n. 4, p. 266–277, 2011. Citado 3 vezes nas páginas 29, 32 e 35.

RAICIU, C.; HANDLEY, M.; BONAVENTURE, O. *TCP extensions for multipath operation with multiple addresses*. [S.l.], 2013. Citado 5 vezes nas páginas 31, 32, 33, 52 e 53.

RAICIU, C. et al. How hard can it be? designing and implementing a deployable Multipath TCP. In: USENIX ASSOCIATION. *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. [S.l.], 2012. p. 29–29. Citado 3 vezes nas páginas 31, 32 e 53.

SCHARF, M.; FORD, A. *Multipath TCP (MPTCP) application interface considerations*. [S.l.], 2013. Citado na página 41.

SCHMIDT, P. S. et al. Socket intents: Leveraging application awareness for multi-access connectivity. In: ACM. *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*. [S.l.], 2013. p. 295–300. Citado 2 vezes nas páginas 40 e 41.

SILVA, A. C.; FERLIN, S.; VERDI, F. Implementação inicial da RFC 6897 para auxílio no tratamento de fluxos elefantes. In: *III WPIETF-IRTF - Anais do XXXVI CSBC*. [S.l.: s.n.], 2016. Citado 4 vezes nas páginas 24, 38, 41 e 57.

STEVENS, W. R.; FENNER, B.; RUDOFF, A. M. *UNIX Network Programming: The Sockets Networking API*. [S.l.]: Addison-Wesley Professional, 2004. Citado 2 vezes nas páginas 23 e 30.

WILLIAMS, N. et al. *Multipath TCP in Vehicular to Infrastructure Communications*. [S.l.], 2014. Citado na página 63.

WU, X.; YANG, X. Dard: Distributed adaptive routing for datacenter networks. In: IEEE. *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*. [S.l.], 2012. p. 32–41. Citado 2 vezes nas páginas 42 e 44.

XU, H.; LI, B. Tinyflow: Breaking elephants down into mice in data center networks. Citado 3 vezes nas páginas 39, 43 e 44.

ZADNIK, M. et al. Tracking elephant flows in internet backbone traffic with an FPGA-based cache. In: IEEE. *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*. [S.l.], 2009. p. 640–644. Citado 2 vezes nas páginas 43 e 44.

APÊNDICE A – Documentação da API de Socket

A.0.1 Funções da API de Socket

Tabela 2 – Funções utilizadas na API de Socket.

Nome	Parâmetros	Valor	Tipo de Retorno
`socket`	*familia*	`AF_UNIX` (Unix Domain Sockets), `AF_INET`(IPv4), `AF_INET6`(IPv6)	Descritor de Arquivo para o Socket Criado
	tipo	`SOCK_STREAM`, `SOCK_DGRAM`	
	protocolo	0 ou número de protocolo	
`setsockopt`	*socket*	Descritor de Arquivo do Socket	0 (Sucesso), -1 (Erro)
	level	Nivel de aplicação da operação	
	optname	Identificador da opção	
	optval	Buffer de Armazenamento do valor da opção	
	optlen	Tamanho atual do valor retornado	
`getsockopt`	*socket*	Descritor de Arquivo do Socket	0 (Sucesso), -1 (Erro)
	level	Nivel de aplicação da operação	
	optname	Identificador da opção	
	optval	Buffer de Armazenamento do valor da opção	
	optlen	Tamanho atual do valor retornado	

A.0.2 Chamada de Socket

Tabela 3 – Criação de Socket para a API.

Função	Familia	Tipo	Protocolo	Retorno
`socket`	`AF_INET`(IPv4), `AF_INET6` (IPv6)	`SOCK_STREAM`	0 ou Identificador do Protocolo	Descritor de Arquivo

A.0.3 Utilização das funções da API

Tabela 4 – Funções estendidas para utilização na API MPTCP.

Função	Socket	Level	optname	optval	optlen
`setsockopt`	*Descritor de Arquivo de Socket*	`IPPROTO_TCP`	`TCP_MULTIPATH_ADD`	Estrutura `mptcp_tcp_sock`	*Inteiro com Tamanho da Estrutura*
`setsockopt`	*Descritor de Arquivo de Socket*	`IPPROTO_TCP`	`TCP_MULTIPATH_ENABLE`	0 (Desligado), 1 (Ligado)	*Inteiro com Tamanho de optval*
`getsockopt`	*Descritor de Arquivo de Socket*	`IPPROTO_TCP`	`TCP_MULTIPATH_CONNID`	Estrutura `mptcp_cb`	*Inteiro com Tamanho da Estrutura*
`getsockopt`	*Descritor de Arquivo de Socket*	`IPPROTO_TCP`	`TCP_MULTIPATH_ENABLE`	Estrutura `mptcp_cb`	*Inteiro com Tamanho da Estrutura*

A.0.4 Exemplo de Função

Segue abaixo um exemplo de função implementada para os testes executados nesse trabalho de dissertação que adiciona subfluxos utilizando o `TCP_MULTIPATH_CONNID` e o `TCP_MULTIPATH_ADD` em Linguagem C.

Código A.1 – Implementação da Função que adiciona subfluxos.

```
void duplicate_sub(int sockfd, mptcp_cb *sub_id){
        int error;
        unsigned int optlen;
        struct mptcp_tcp_sock *sub_tuple;

        optlen = 100;

        sub_tuple = malloc(optlen);
        if (!sub_tuple) {
                return ;
        }

        error = getsockopt(sockfd, IPPROTO_TCP, TCP_MULTIPATH_CONNID,
                &sub_id, optlen);

        sub_tuple->id = sub_id->id;

        optlen = sizeof(struct mptcp_sub_tuple) + 2 * sizeof(struct sockaddr_in);

        error = setsockopt(sockfd, IPPROTO_TCP, TCP_MULTIPATH_ADD, sub_tuple,
                        &optlen);
        if (error) {
                perror(NULL);
                db_bug("Erro adicionando novo subfluxo (MP_JOIN) !%s","\n");
                free(sub_tuple);
                return;
        }
        change_tos(sockfd, sub_id);
        free(sub_tuple);
}
```