Trabalho Prático 2 de Sistemas em Rede

Alexander Decker de Sousa

26 de Abril de 2018

1 Descrição do Problema

Este trabalho objetivou a implementação de uma versão paralelizada do *switch* de software do *BPFabric* [3], de forma que cada porta possuísse um agente *eBPF* dedicado e fosse vista independentemente pelo controlador.

2 Motivação

O plano de dados do *BPFabric* (Figura 1.a) consiste em um conjunto de portas de entrada e de saída cujos quadros são multiplexados e decodificados duas vezes para o tratamento sequencial das mensagens em dois mecanismos-gargalo: o *Metadata Prepend* e o *eBPF Execution Engine*. Numa eventual implementação física, a replicação destes mecanismos (Figura 1.b) garantiria melhor desempenho, visto que suas operações são aproximadamente independentes e o paralelismo em *hardware* é, no caso, inato. Seria necessário apenas um mecanismo para evitar colisões nas portas de saída. Tal replicação em software, todavia, não necessariamente leva direta e facilmente a melhorias de desempenho, por razões de concorrência e mesmo escalonamento de *threads*.

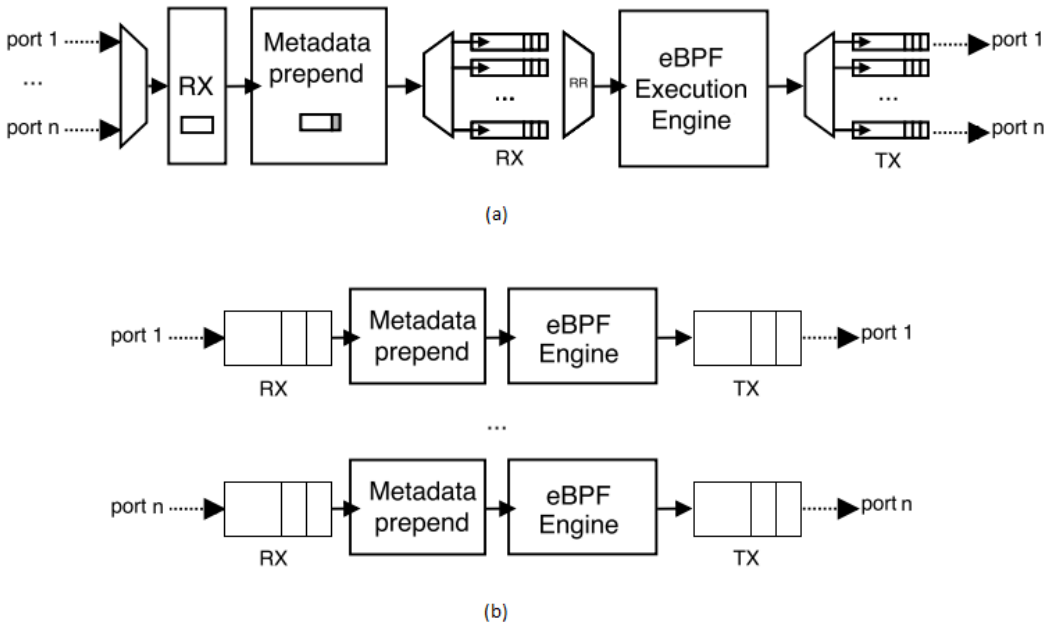

Figure 1: Caminhos de dados no modelo original do *BPFabric* (a) e no modelo paralelizado (b).

Outra vantagem da implementação multi-caminho em *hardware* é o aumento da flexibilidade dos *switches*, visto que cada porta possuiria um agente eBPF independente e, portanto, o *switch* poderia ser particionado em vários *switches* lógicos, cada um rodando uma aplicação independente e isolada.

Um dos objetivos deste trabalho foi, portanto, buscar uma solução para que a replicação dos gargalos em software de forma a melhorar o desempenho do *switch*. Como o *switch* foi desenvolvido

para o *Mininet* [2] e o mesmo é comumente utilizado para avaliações experimentais e provas de conceito no desenvolvimento de protocolos, serviços e aplicações de rede, a própria implementação funcional em *software* do *BPFabric* paralelizado – sem entrar nos méritos de melhorias de desempenho – é sim útil, por oferecer uma ferramenta extra de desenvolvimento para aplicações que envolvam o de fato e indubitavelmente útil *switch* físico de multi-caminho.

3 Trabalhos Relacionados

A primeira implementação de *Redes Defnidas por Software* propriamente ditas veio com o protocolo e plataforma *OpenFlow* [4], que atualmente já é bastante utilizado no mercado. Todavia, o *OpenFlow* oferece uma programabilidade limitada e alta dependência de protocolos, o que levou à elaboração de plataformas como o *P4* [1], que consiste em uma linguem declarativa de alto nível compilável para definir políticas de encaminhamento no plano de dados, e o *POF* [5], que foca sobretudo na independência em relação a protocolos.

Outra plataforma que surgiu buscando independência de protocolos e melhorias de programabilidade foi o *BPFabric*, que permite a inserção de microcódigo em um processador simplificado no plano de controle, o que gera maior liberdade no que se refere às aplicações de rede possíveis. O *BPFabric* ainda não possui bastante visibilidade na comunidade acadêmica, mas pode rodar como um *switch* de *software* convencional ou baseado no *kit* de desenvolvimento de planos de dados de alto desempenho DPDK[1].

4 Decisões de Implementação

Por razões de simplicidade de implementação, o código do pacote *softswitch* do *BPFabric* foi utilizados como base para a criação da versão paralela, que pode ser acessada em [2].

O código original do *BPFabric* agrupa todas as funções diretamente relacionadas às operações principais do *switch* de software - com exceção dos aspectos relativos ao agente *eBPF* - no próprio arquivo *main.c*. Portanto, para melhor legibilidade e organização, apenas as funções e estruturas relacionadas ao *parse* dos argumentos de entrada foram mantidas no arquivo. As funções relacionadas à criação dos *sockets*, da alocação do *ring-buffer* pelo *PACKET_MMAP*, ao mapeamento do *ring-buffer* em espaço de usuário, à inserção de novos quadros nas filas de saída e afins foram movidas para o arquivo *softswitch.c*, de forma que as novas funções, mantidas em sua maioria no arquivo *switchFabric.c*, pudessem acessá-las juntamente com o arquivo principal.

Para a paralelização das operações do *switch*, optou-se por associar uma *thread* a cada porta de entrada, de forma que a mesma ficasse responsável por rodar o interpretador *eBPF* para cada novo quadro e encaminhar para a porta de saída correspondente. Para evitar conflitos, cada porta de saída foi associada a um *mutex* de forma a criar uma sessão crítica durante a inserção de novos quadros. Para fins de simplificação, foi criada uma *thread* responsável por fazer as chamadas periódicas a *send* para cada fila de saída.

O fluxo de operação do *switch* passou a alternar por dois estados mutuamente excludentes. Em um deles, as *threads* dedicadas às portas de entrada processam todos os quadros em suas respectivas filas. Quando o fluxo de execução de cada uma delas trava em razão da escrita de novos quadros nas filas de entrada por parte do sistema operacional, apenas a *thread* que faz as chamadas de *send* permanece ativa. O fluxo de execução, então, passa para o outro estado, em que são enviados os quadros pendentes e é realizada uma chamada à função *poll*, responsável por aguardar a chegada de novos quadros. Após o término da mesma, a *thread* se desativa e ativa as demais, retornando, portanto, ao estado inicial.

Inicialmente, a referida “desativação" de uma *thread* era implementada simplesmente através de um *loop* vazio aguardando o retorno ao estado ativo. Todavia, essa abordagem demonstrou-se ineficiente em demasia, o que levou a alteração para uma verificação do estado inativo que, em caso positivo, realiza uma chamada à função *sched_yield*, capaz de ceder o contexto para outra *thread* determinada pelo escalonador.

Por último, para permitir a coexistência de múltiplos agentes eBPF, as funções de *agent.c* foram adaptadas, dando origem a *multiagent.c*. A implementação original faz uso de duas variáveis globais – *agent*, que guarda o *socket* para a comunicação com o controlador e um ponteiro para a função

[1] http://dpdk.org/
[2] https://github.com/AlexDecker/BPFabricIQSwitch

que processa os quadros, e *vm*, que guarda as tabelas utilizadas pelo programa *eBPF* e algumas definições de função para serem chamadas pelo mesmo e pela *thread* responsável por gerenciar a comunicação entre cada agente e o controlador. As variáveis globais foram replicadas para cada porta de entrada e as funções que as acessam receberam um parâmetro adicional para se referirem ao agente e à máquina virtual corretos.

Como a maioria destas funções é chamada pelas *threads* que gerenciam os agentes, bastou informar a porta à qual cada agente era dedicado no momento da criação das *threads*. Todavia, a função *bpf_notify* poderia também ser chamada diretamente pelos programas *eBPF*, o que obrigou a alteração de seu protótipo em *ebpf_functions.h* para a inserção da identificação da porta de entrada. As chamadas da função nos exemplos também foram alteradas, de forma a permitir a compilação dos mesmos.

Para manter a compatibilidade com os controladores feitos para o *BPFabric* convencional, cada agente *eBPF* envia sua própria mensagem de *Hello* logo que é criado, informando ao controlador um identificador de plano de dados individual. Dessa forma, os controladores enxergam cada porta como um *switch* independente, em que pode ser instalado uma aplicação independente. O controlador, todavia, também pode identificar quais agentes *eBPF* são oriundos do mesmo *switch*. Para permitir isso, o identificador do plano de dados passou de 64 bits para os 32 bits menos significativos. Os 32 bits mais significativos informados ao controlador correspondem à porta de entrada que de fato identifica o agente. Para evitar conflitos, a função de geração aleatória de identificador de plano de dados no arquivo principal também foi alterada para gerar um número de até 32 bits.

5 Experimentos

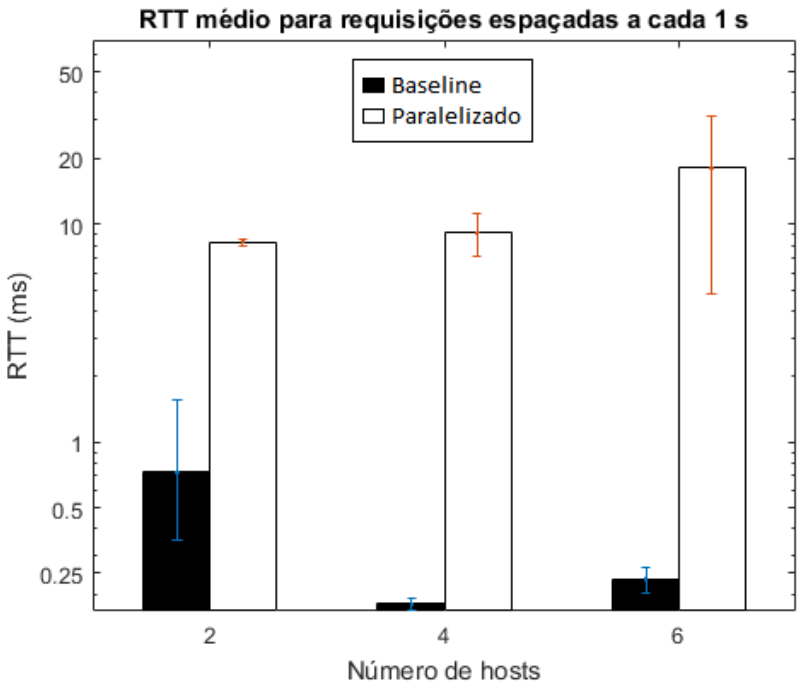

Figure 2: Comparação dos *RTTs* de requisições *Ping* feitas em redes interligadas por um *switch BPFabric* convencional (*baseline*) e sua versão paralelizada. Nesse experimento as requisições *Ping* foram espaçadas em intervalos de um segundo.

Vários testes com os exemplos do *BPFabric* foram testados, mostrando que a implementação paralela de fato funciona, permitindo que cada porta possua uma aplicação diferente e executando os códigos *eBPF* em paralelo. Porém, esta implementação inicial não se mostrou eficiente. Para demonstrar isso, foram feitos dois gráficos com as medidas de RTT para requisições *Ping*, de forma a comparar o desempenho do *switch* paralelizado com o convencional em redes de diferentes tamanhos. Em todos os casos, foi utilizado um *switch* interligado a um número variado de *hosts*.

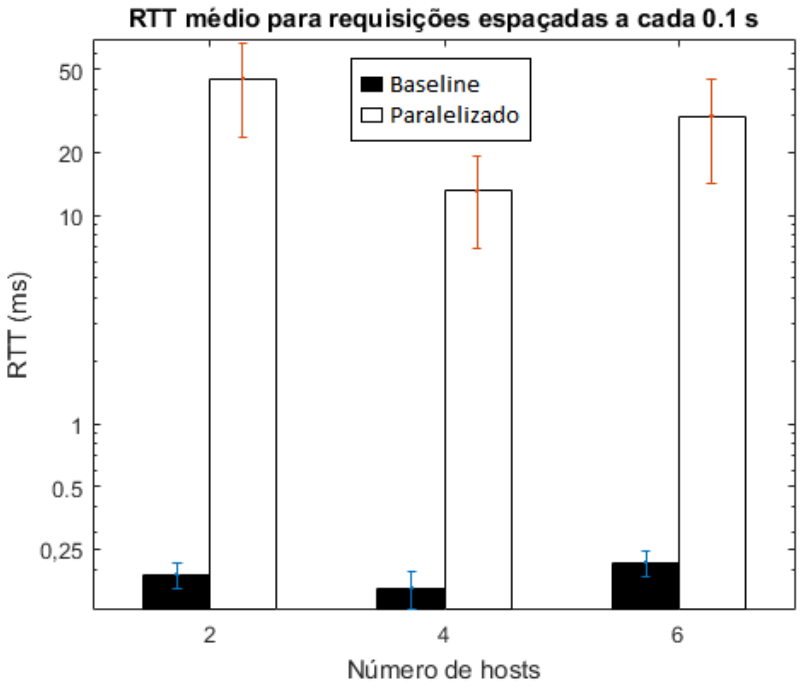

Figure 3: Comparação dos *RTTs* de requisições *Ping* feitas em redes interligadas por um *switch BPFabric* convencional (*baseline*) e sua versão paralelizada. Nesse experimento as requisições *Ping* foram espaçadas em intervalos de cem milissegundos.

Cada *host* fazia requisições sucessivas espaçadas em intervalos de um segundo (Figura 2) ou cem milissegundos (Figura 3) para algum outro *host* da rede. Foram feitas 15 amostragens em cada caso para cada *switch*. A discrepância dos resultados foi tamanha que a escala logarítmica precisou ser utilizada para a comparação dos *RTTs*.

Tais problemas de eficiência são devidos ao baixo custo de execução dos códigos *eBPF* em si. Foi averiguado posteriormente que praticamente todas as vezes que uma *thread* era escalonada a função *sched_ yield* era chamada – e parte significativa das vezes apenas a mesma era chamada. A cada chamada, a *thread* ia para o fim da fila relativa à sua prioridade. Portanto, nem o aumento da prioridade das *threads* conseguiu melhorias notáveis.

6 Conclusão e melhorias

A avaliação experimental demonstrou que a implementação da versão de *software* do *switch* paralelizado possui limitações de eficiência consideráveis. Como o *Mininet* é importante sobretudo por razões de prova de conceito no desenvolvimento de protocolos, serviços e aplicações de rede, o *switch* de *software* aqui proposto pode ser usado para representar sua versão física que, por possuir paralelismo inato e não sofrer com trocas de contexto e concorrência, seria de fato mais eficiente que a versão serial. Todavia, sua utilização para o gerenciamento de máquinas virtuas em servidores, por exemplo, é bastante desaconselhável.

Não obstante, é esperado que o aumento to tamanho e complexidade do código *eBPF* seja menos sentida pela versão paralelizada do que pela original. Além disso, é esperado que a versão paralelizada tenha melhor desempenho caso haja uma quantidade pequena de processos concorrendo no escalonamento.

Uma forma de melhorar o desempenho é escalonar as portas de entrada para os agentes *eBPF*, ao invés de cada um ser dedicado a uma porta específica. Dessa forma, ao invés de chamar *sched_ yield* quando não conseguisse mais obter quadros de uma porta, o agente simplesmente escolheria uma outra porta que não tivesse sido designada para outro agente ainda. Outra otimização notável seria executar as chamadas de *send* de tempos em tempos a partir de uma *thread* qualquer, sem precisar transferir o contexto para uma *thread* dedicada. Outra melhoria consistiria na implementação baseada no pacote *dpdkswitch* do *BPFabric*.

References

[1] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[2] Rogério Leão Santos De Oliveira, Ailton Akira Shinoda, Christiane Marie Schweitzer, and Ligia Rodrigues Prete. Using mininet for emulation and prototyping software-defined networks. In *Communications and Computing (COLCOM), 2014 IEEE Colombian Conference on*, pages 1–6. IEEE, 2014.

[3] Simon Jouet and Dimitrios P Pezaros. Bpfabric: Data plane programmability for software defined networks. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems*, pages 38–48. IEEE Press, 2017.

[4] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[5] Haoyu Song. Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 127–132. ACM, 2013.