Real time alpha-fairness based traffic engineering

Bill McCormick
Huawei Technologies Canada
Kanata Ontario, Canada
bill.mccormick@huawei.com

Frank Kelly
University of Cambridge
Cambridge, UK
f.p.kelly@statslab.cam.ac.uk

Patrice Plante
Huawei Technologies Canada
Kanata, Ontario, Canada
patrice.plante@huawei.com

Paul Gunning
BT Research & Innovation
Adastral Park, Ipswich, UK
paul.gunning@bt.com

Peter Ashwood-Smith
Huawei Technologies Canada
Kanata, Ontario, Canada
peter.ashwoodsmith@huawei.com

ABSTRACT

Software defined networking (SDN) traffic engineering has proved to be a computationally difficult problem, resulting in long execution times to compute large scale flow assignments. Other authors have taken approximate approaches to achieving fairness to improve computational scalability. We derive a solution to the general alpha-fairness problem that scales near-linearly with the problem size and is well suited to a massively parallel implementation. CPU based results compare scalability and accuracy with proportional fair and max min fair solutions on a simulation of BT's production network. Results from our FPGA implementation show a three order of magnitude reduction in execution time, allowing large scale flow assignment computations to take place in times measured in milliseconds instead of seconds or minutes. We believe that these techniques will finally enable real time traffic engineering in SDN.

1. INTRODUCTION

One of the objectives of SDN is to remove the more complex functions from routers and virtualize them in a data center. In today's networks, routers implement shortest path based routing protocols which are used to route traffic through the network and provide either de jure traffic engineering (as in MPLS-TE) or de facto traffic engineering (as in OSPF). When these shortest path first (SPF) routing functions are no longer present in the router, a new traffic engineering function is required. Instead of being distributed through the network, this function can be centralised. This central function can have global knowledge of the network that is not available to the distributed SPF routing in today's network.

Two key network attributes related to traffic engineering are network throughput and fairness. (Another key attribute is delay, however we do not address delay directly in this paper.) We balance these attributes using the concept of alpha fairness introduced by Mo and Walrand in [11] where α is a parameter in the range $[0, \infty]$ to denote fairness.

There are three specific values of α which are of interest. Setting $\alpha = 0$ corresponds to a flow assignment which maximizes the network throughput, but makes no attempt to ensure fairness among flow assignments.

As $\alpha \to \infty$, the flow assignment becomes max-min fair. A flow assignment is max-min fair when the bandwidth assigned to a flow can only be increased by decreasing the bandwidth assigned to some other flow with an equal or smaller assignment. Thus max-min fairness is focused on making the minimum flow assignment as large as possible without regard to the impact on total throughput.

Setting $\alpha = 1$ corresponds to a proportional fair solution. Proportional fair solutions were first described by Nash [12] as the solution to a negotiation problem. They provide an appealing compromise between max-min fairness - which allocates flows fairly without regard for network resource usage - and maximal throughput - which allocates flows without regard for fairness.

Optimization programs to solve these flow assignment problems are given in [13]. The maximum throughput problem can be solved with a single linear program. The proportional fair problem requires a convex program, so a traditional linear solver will not suffice here. The max-min fair problem requires the solution of a sequence of linear programs which grows polynomially with the problem size. Techniques for solving these problems all exhibit polynomial computation scalability, as traditional solutions require the repeated factoring of a matrix which grows with the problem size.

Recent papers [9] and [10] describe practical traffic engineering implementations on data center networks with less than one hundred nodes. In [9], the authors relax the max-min fairness constraints to reduce the number of linear programs needed to solve the optimization problem. In [10], the authors describe a greedy algorithm which approximates max-min fairness without the cost of executing a sequence of linear programs.

The focus of our work is on large carrier networks,

ranging in size from one hundred to a few thousand nodes. We present a new method for solving these problems which scales near-linearly with the problem size and is also well suited to a massively parallel implementation.

2. ALGORITHM DERIVATION

Model the network as a set of J directed links, individually identified as $j \in J$. Each link has capacity C_j. The term r is used to identify a specific path through the network. An individual flow is identified by the term s. The bandwidth assigned to a specific flow is identified by x_s, and the bandwidth from flow s assigned to path r is identified by y_r. We use the terminology $r \in s$ to denote the paths that are used by a specific flow and $r \in j$ to denote the paths that use link j. When we are referring to a specific path r, we use the expression $s(r)$ to denote the parent flow of the path.

The optimization program we use for a weighted α fair flow assignment is given by

$$\begin{array}{rc} \text{maximize} & \sum_{s\in S} w_s^\alpha \frac{x_s^{1-\alpha}}{1-\alpha} \\ \text{subject to} & \sum_{r\in s} y_r = x_s, \\ & \sum_{r\in j} y_r \le C_j \\ \text{over} & x, y > 0 \end{array}$$

The term w_s is a weight assigned to each flow, allowing the user to request that some flows be assigned proportionally more or less bandwidth than others.

This program has unique values for x, however the solution for y is usually non-unique. If we define

$$x_s = \left(\sum_{r\in s} y_r^q\right)^{\frac{1}{q}}$$

where q is some constant close to, but less than, one, then the optimization problem has a unique solution for both the x values and the y values. With this change, our objective function becomes the convex function

$$\text{maximize} \sum_{s\in S} w_s^\alpha \frac{\left(\sum_{r\in s} y_r^q\right)^{\frac{1-\alpha}{q}}}{1-\alpha}$$

Returning to first principles, we can construct the Lagrangian for this problem as

$$\begin{aligned} L(y,z;\mu) = & \sum_s w_s^\alpha \frac{\left(\sum_{r\in s} {y_r}^q\right)^{\frac{1-\alpha}{q}}}{1-\alpha} + \\ & \sum_j \mu_j \left(C_j - \sum_{r\in j} y_r - z_j\right) \end{aligned}$$

Here z_j and μ_j are slack variables and shadow prices for link j respectively. From complementary slackness, we know that for a given j, either $\mu_j = 0$ or $z_j = 0$ [6]. In other words, at the solution to our optimization problem, if the shadow price is non-zero then link j is saturated, and if link j is under committed then its shadow price is 0.

We can differentiate L with respect to y_r to develop a relationship between y, x and μ.

$$\frac{\partial L}{\partial y_r} = w_{s(r)}^\alpha y_r^{q-1} \left(\sum_{r'\in s(r)} y_{r'}^q\right)^{\frac{1-\alpha}{q}-1} - \sum_{j\in r} \mu_j.$$

At the optimum point this derivative will be equal to zero. Setting $\frac{\partial L}{\partial y_r} = 0$ and rearranging we find that

$$y_r = \left(\left(\frac{w_{s(r)}}{x_{s(r)}}\right)^\alpha \cdot \frac{1}{\sum_{j\in r} \mu_j}\right)^{\frac{1}{1-q}} x_{s(r)}$$

Update rules for X_s and μ_j should be of the form

$$\begin{aligned} \mu_j(t+1) &= \mu_j(t) + k_j \dot{\mu}_j(t)\Delta t \\ x_s(t+1) &= x_s(t) + k_s \dot{x}_s(t)\Delta t \end{aligned}$$

where k_j and k_s are gain parameters for the update rules for μ and x respectively, and the dot notation denotes the time derivative.

In [14], Thomas Voice shows that this class of system converges to the optimal value provided that the gain parameters are constrained appropriately. Following [14] and setting the gain parameters to their maximum stable values gives the optimization algorithm as

$$y_r = \left(\left(\frac{w_{s(r)}}{x_{s(r)}}\right)^\alpha \cdot \frac{1}{\sum_{j\in r} \mu_j}\right)^{\frac{1}{1-q}} x_{s(r)} \quad (1)$$

$$\mu_j(t+1) = \mu_j(t) + \frac{1-q}{2}\mu_j(t)\left[\frac{\sum_{r\in j} y_r(t) - C_j}{C_j}\right] \quad (2)$$

$$x_s(t+1) = x_s(t) + \frac{1-q}{2(\alpha+q-1)} x_s(t) \cdot \left[\frac{\sum_{r\in s} y_r(t)^q - x_s(t)^q}{x_s(t)^q}\right] \quad (3)$$

Note carefully that each of the update rules in equations (1), (2) and (3) can be implemented in parallel. In other words, all of the y_r values in (1) can be computed in parallel, then all of the μ_j values in (2) can be computed and so on. This property makes it straightforward to implement the algorithm on massively parallel hardware.

3. SIMULATION RESULTS

In order to assess the performance of this algorithm, we have run simulations comparing the algorithm results to reference implementations for max-min fairness and proportional fairness. Our Lagrangian based algorithm is implemented in Java 7. For the reference implementations we use general purpose open source solvers written in C and FORTRAN as detailed below.

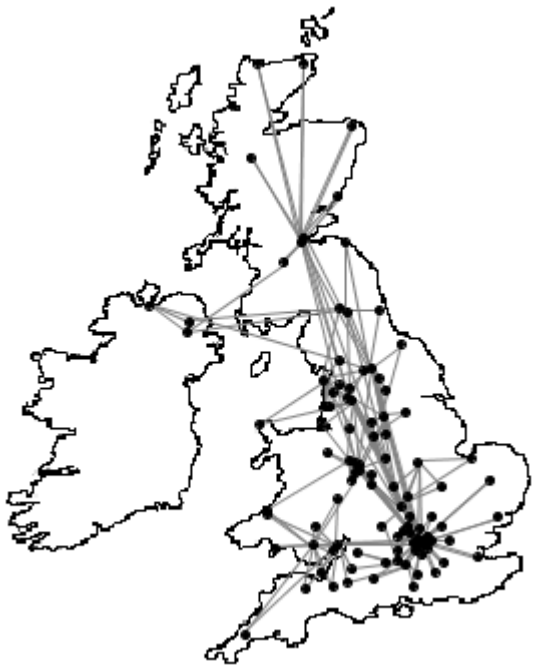

Figure 1: BT 21CN network

The simulations are run on an x86 based virtual machine. We aren't concerned with making absolute performance comparisons at this time - the purpose of this step is to assess accuracy and to obtain a general feel for the computational complexity of the algorithm as compared to the traditional implementations.

We have used BT's 21CN network topology (Figure 1) comprising 106 nodes and 234 links within the United Kingdom as a reference for these simulations. Flows are generated using a pseudo-random number generator so that the end points for each flow are randomly selected. All flows are treated as elastic, so they will consume all network bandwidth available to them.

3.1 Max-min Fairness

Our max-min fairness reference implementation uses the GNU linearing programming kit [1]. This is a scalable open source linear solver written in C. The reference algorithm is Algorithm 8.3 from [13]. For our Lagrangian algorithm, we choose $q = 0.9$ and $\alpha = 4$ as an approximation for max-min fairness.

The simulation results are shown in Figures 2 and 3. As expected, the execution time grows rapidly with the problem size for the reference algorithm as larger problems require execution of a growing number of linear programs. Our algorithm shows a roughly linear increase in execution time with problem size. Choice of $q = 0.9$ provides a good approximation of max-min fair, holding the root mean square error from the reference implementation at around 1%.

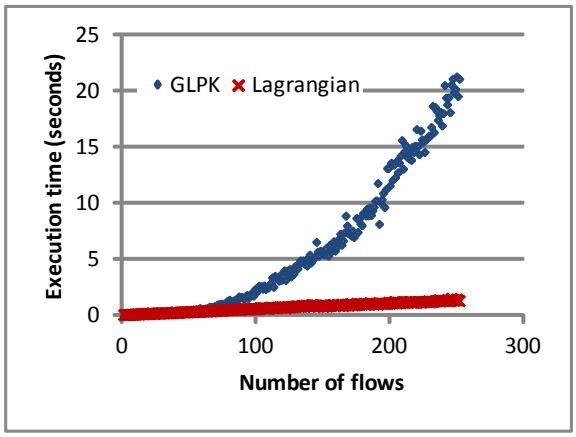

Figure 2: Max-min fair execution time

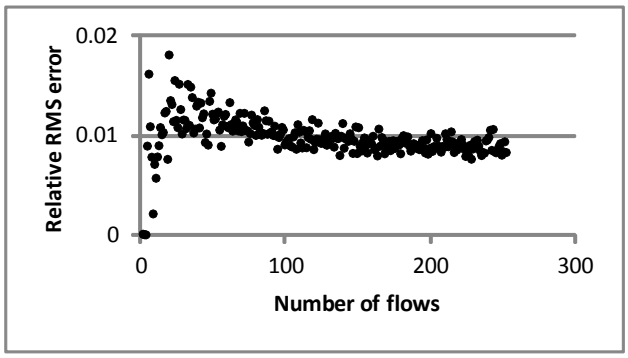

Figure 3: Max-min fair RMS error

3.2 Proportional Fairness

Our proportional fairness reference implementation requires a convex optimizer as it has a non-linear objective function. For this simulation we have used the interior point optimizer Ipopt [3], an open source library known for its good scalability properties. This library is written in C and FORTRAN and we have configured it with the MUMPS linear solver [4]. The reference algorithm here is from section 8.1.3 of [13].

The proportional fair simulation results are shown in Figures 4 and 5. In this case, the reference implementation requires the execution of a single non-linear optimization program, so it doesn't exhibit the higher order polynomial growth of the max-min fair implementation. Our Lagrangian based method generally matches the performance of the reference implementation. As with the max-min fair example, choice of $q = 0.9$ keeps the RMS error to approximately 0.5%.

4. FPGA IMPLEMENTATION

We have implemented our Lagrangian based algorithm in a Xilinx FPGA Virtex-7 XC7VX485T-3. The

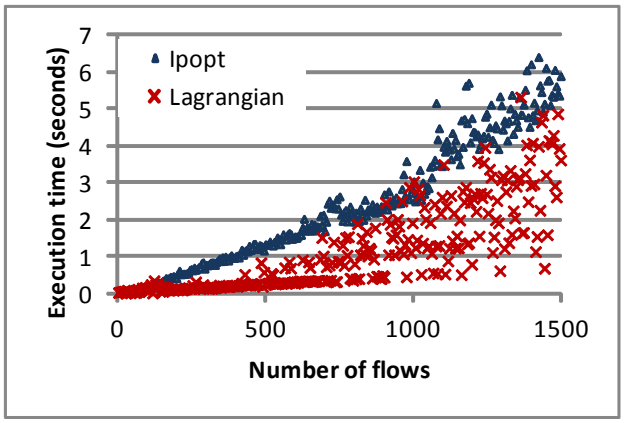

Figure 4: Proportional fair execution time

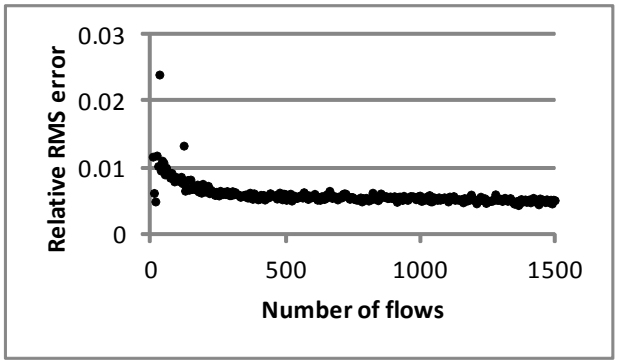

Figure 5: Proportional fair RMS error

goal of the hardware implementation is to take advantage of the parallelism that can be found in the algorithm design to evaluate how we can improve the convergence time by executing concurrently as many operations as possible.

As illustrated in the block diagram of Figure 7, the hardware implementation consists of 1024 parallel adders that feed 32 parallel Deep Pipelines. Pipelining is a powerful concept that allows multiple operations to be executed sequentially and simultaneously on large independent data samples.

In our implementation, each Deep Pipeline is made of 180 pipeline stages consisting of 12 Divide, 15 Multiply, 4 Square Root, 44 Add and 2 Subtract operations. When a Deep Pipeline is fully populated, that single pipeline can produce one partial result of equation (1), (2) and (3) every clock cycle. With a core clock running at 200MHz, the FPGA produces 500G fixed point operations per second.

Performance results for our FPGA implementation are shown in Figure 6. This figure shows flow computation results for flow assignment with $q = 0.75$ and $\alpha = 4.0$ to provide a reasonable approximation of max-

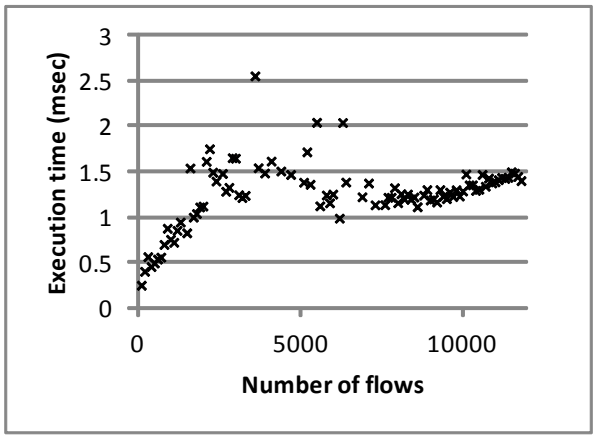

Figure 6: FPGA max-min fair execution time

min fair flow assignment. The notable difference between these results and those in Figure 2 are that the execution time has dropped from seconds to milliseconds.

Comparing the two results, we attribute the 3 order of magnitude improvement to hardware pipelining. In order to benefit from hardware pipelining, the network size must be large enough to keep the pipeline full most of the time. This effect is illustrated in Figure 6, where performance reaches a steady state for greater than 1000 active flows by avoiding the penalty cost of filling and emptying a partially filled pipeline.

5. DISCUSSION

5.1 Performance

When we began work on this project, it was clear that solving flow assignment problems is a compute intensive task. Modern processors have lots of processing power, but its available in a highly parallel form. For example, a NVIDIA K20 has 3.52 TFLOPs spread over 2496 cores [5]. An Intel Core^{TM} i7-980 has 80 GFLOPs available over 6 cores [2]. The compute throughput of a field programmable gate array is not specified in this manner, however in [7], the authors demonstrated 180 GFLOPS of practical compute throughput on a Virtex-7 FPGA.

In order to make effective use of these compute environments, it was necessary that the algorithm design map directly to a highly parallel architecture. We feel that the dramatic performance gains of our hardware implementation validates our approach.

5.2 Convergence

A significant factor in execution time is the number of iterations required to converge. We detect convergence by measuring the relative change in the norm of

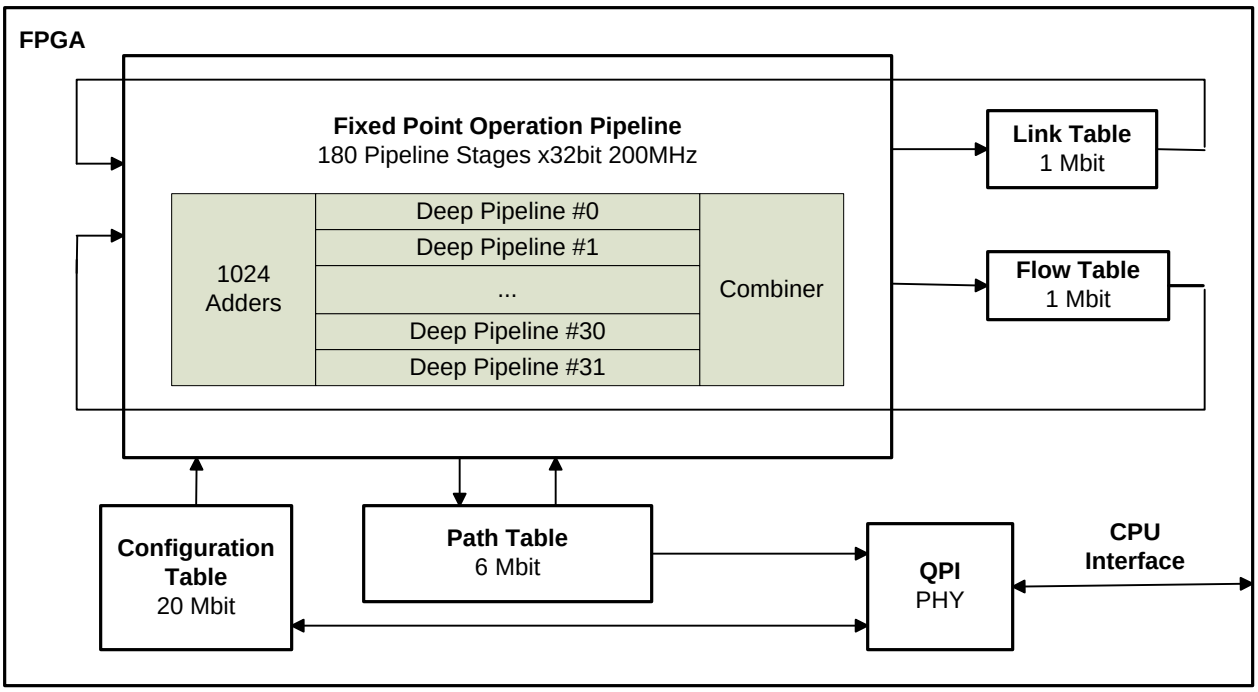

Figure 7: FPGA block diagram

the vector of y_r values. When this change drops below a threshold (10^{-6} in our examples), the algorithm is stopped.

Empirically the number of iterations to convergence has varied in the range $[200, 2400]$. There appears to be a direct relationship between the iterations to converge and the number of link constraints that are active or almost active. As the number of active constraints increases, the algorithm takes more time to explore the problem structure and converge to a solution. This is a topic for further study.

5.3 Further FPGA design considerations

The hardware implementation is limited by the silicon area of the FPGA. The algorithm could be further accelerated by adding even more Deep Pipelines in parallel in a larger FPGA device. To reduce silicon consumption by the Deep Pipeline, our hardware implementation uses fixed point operations instead of floating point operations. To avoid floating point operations, the Deep Pipeline is divided in two distinct regions. The two regions use different fixed point representation and are merged by a multiplication operation.

The bulk of the FPGA real estate is consumed in maintaining the configuration table that holds the network definition. This information could be stored in external RAM which would allow us to increase the size of the Link, Flow and Path tables which are currently configured to support 512 links, 32K flows, 96K paths and 12 links per path.

The latest FPGA offerings support a core clock frequency of 400Mhz, while our implementation uses a 200Mhz core clock. Migrating to one of these FPGAs would double the computational performance.

6. CONCLUSION

Our FPGA implementation is by far the fastest implementation of the max-min fair and the more general α-fairness problem we have seen published. It exhibits near-linear computational and memory scalability which makes it an excellent choice for use on large carrier scale networks.

The algorithm has desirable implementation properties in terms of simplicity. Instead of the complicated matrix factorization used in interior point methods [3] [4] [6], we use straightforward vector operations for update rules. The implementation team does not need to be versed in arcane rules for matrix computations and can focus on the more practical problems of identifying performance bottlenecks.

Our solution runs in millisecond times, making it fast enough to be part of the real time control loop in the network instead of an off-line tool. This means that we finally have line of sight into real time traffic engineering as originally envisaged for Software Defined Networking [8].

7. REFERENCES

[1] Gnu linear programming kit. http://www.gnu.org/software/glpk. Accessed:

2014-03-06.

[2] Intel microprocessor export compliance metrics. http://www.intel.com/support/processors/sb/CS-032814.htm. Accessed: 2014-03-20.

[3] Interior point optimizer. http://project.coin-or.org/Ipopt. Accessed: 2014-03-06.

[4] Mumps: a multifrontal massively parallel sparse direct solver. http://mumps.enseeiht.fr. Accessed: 2014-03-06.

[5] Tesla Kepler GPU accelerators. http://www.nvidia.com/content/tesla/pdf/Tesla-KSeries-Overview-LR.pdf. Accessed: 2014-03-06.

[6] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, New York, 2004.

[7] J. Capello and D. Strenski. A practical measure of FPGA floating point acceleration for high performance computing. In *2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, pages 160–167, 2013.

[8] S. Das, N. McKeown, et al. Application-aware aggregation and traffic engineering in a coverged packet-circuit network. In *Proceedings of OFC/NFOEC 2011*, pages 1–3, 2011.

[9] C.-Y. Hong et al. Achieving high utilization with software-driven wan. In *Proceedings of the ACM SIGCOMM 2013*, pages 15–26, 2013.

[10] S. Jain et al. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013*, pages 3–14, 2013.

[11] J. Mo and J. Walrand. Fair end-to-end window-based congestion control. *IEEE/ACM Transactions on Networking*, 8(5):556–567, October 2000.

[12] J. Nash. The bargaining problem. *Econometrica*, 18(2):155–162, April 1950.

[13] M. Pioro and D. Medhi. *Routing, Flow, and Capacity Design in Communication and Computer Networks*. Morgan Kaufmann Publishers, 2004.

[14] T. Voice. Stability of multi-path dual congestion control algorithms. *IEEE/ACM Transactions on Networking*, 15(6):1231–1239, December 2007.