Expert Systems With Applications 91 (2018) 492–512

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

A distributed and quiescent max-min fair algorithm for network congestion control☆

Alberto Mozo[a,*], José Luis López-Presa[b,c], Antonio Fernández Anta[d]

[a] Dpto. Sistemas Informáticos., Universidad Politécnica de Madrid, Madrid, Spain
[b] Dpto. Telemática y Electrónica, Universidad Politécnica de Madrid, Madrid, Spain
[c] DCCE, Universidad Técnica Particular de Loja, Loja, Ecuador
[d] IMDEA Networks Institute, Madrid, Spain

ARTICLE INFO

Article history:
Received 30 January 2017
Revised 19 August 2017
Accepted 9 September 2017
Available online 12 September 2017

Keywords:
Max-min fair
Distributed optimization
Proactive algorithm
Quiescent algorithm
Network congestion control

ABSTRACT

Given the higher demands in network bandwidth and speed that the Internet will have to meet in the near future, it is crucial to research and design intelligent and proactive congestion control and avoidance mechanisms able to anticipate decisions before the congestion problems appear. Nowadays, congestion control mechanisms in the Internet are based on TCP, a transport protocol that is totally reactive and cannot adapt to network variability because its convergence speed to the optimal values is slow. In this context, we propose to investigate new congestion control mechanisms that (a) explicitly compute the optimal sessions' sending rates independently of congestion signals (i.e., proactive mechanisms) and (b) take anticipatory decisions (e.g., using forecasting or prediction techniques) in order to avoid the appearance of congestion problems.

In this paper we present B-Neck, a distributed optimization algorithm that can be used as the basic building block for the new generation of proactive and anticipatory congestion control protocols. B-Neck computes proactively the optimal sessions' sending rates independently of congestion signals. B-Neck applies max-min fairness as optimization criterion, since it is very often used in traffic engineering as a way of fairly distributing a network capacity among a set of sessions. B-Neck iterates rapidly until converging to the optimal solution and is also quiescent. The fact that B-Neck is quiescent means that it stops creating traffic when it has converged to the max-min rates, as long as there are no changes in the sessions. B-Neck reacts to variations in the environment, and so, changes in the sessions (e.g., arrivals and departures) reactivate the algorithm, and eventually the new sending rates are found and notified. To the best of our knowledge, B-Neck is the first distributed algorithm that maintains the computed max-min fair rates without the need of continuous traffic injection, which can be advantageous, e.g., in energy efficiency scenarios.

This paper proposes as novelty two theoretical contributions jointly with an in-depth experimental evaluation of the B-Neck optimization algorithm. First, it is formally proven that B-Neck is correct, and second, an upper bound for its convergence time is obtained. In addition, extensive simulations were conducted to validate the theoretical results and compare B-Neck with the most representative competitors. These experiments show that B-Neck behaves nicely in the presence of sessions arriving and departing, and its convergence time is in the same range as that of the fastest (non-quiescent) distributed max-min fair algorithms. These properties encourage to utilize B-Neck as the basic building block of proactive and anticipatory congestion control protocols.

☆ A preliminary version of this work was presented at the 10th IEEE International Symposium on Network Computing and Applications (NCA 2011) (Mozo et al. (2011)). This extended version has several substantial differences from the conference: (1) We provide a new introduction to justify the current interest of researchers in proactive max-min fair algorithms in order to alleviate some of the limitations that TCP exhibits in the high demanding network bandwidth and speed scenarios that the Internet will have to meet in the near future; (2) We formally prove the correctness of the algorithm, i.e., we prove that when no more sessions join or leave the network, B-Neck eventually converges to the max-min fair values and stops injecting packets to the network; (3) We formally prove an upper bound on B-Neck convergence time when no more sessions join or leave the network; and (4) We add a new set of detailed experiments that validate the obtained theoretical results and evaluate how B-Neck performs under real-world settings.

* Corresponding author.
E-mail addresses: a.mozo@upm.es, amozo@etsisi.upm.es (A. Mozo), jllopez@diatel.upm.es (J.L. López-Presa), antonio.fernandez@imdea.org (A. Fernández Anta).

http://dx.doi.org/10.1016/j.eswa.2017.09.015
0957-4174/

1. Introduction

Nowadays, congestion control mechanisms in the Internet are based on the TCP protocol that is widely deployed, scales to existing traffic loads, and shares network bandwidth applying a flow-based fairness to the network sessions. TCP entities implement a closed-control-loop algorithm that reactively recompute sessions' rates when congestion signals are received from the network. There is a general consensus regarding the higher demands in network bandwidth and speed that the Internet will have to meet in the near future. In this context of speeds scaling to 100 Gb/s and beyond, TCP and other closed-control-loops approaches converge slowly to the optimal (fair) sessions' rates. Therefore, it is vital to investigate intelligent and proactive congestion control and avoidance mechanisms that can anticipate decisions before the congestion problems appear. Some current research works propose the usage of proactive congestion control protocols leveraging distributed optimization algorithms to explicitly compute and notify sending rates independently of congestion signals (Jose et al. (2015)). Complementary, a growing research trend is to not just react to network changes, but anticipate them as much as possible by predicting the evolution of network conditions (Bui et al. (2017)). We propose to investigate new congestion control mechanisms that (a) explicitly compute the optimal sessions' sending rates independently of congestion signals (i.e., proactive mechanisms) and (b) can leverage the integration of anticipatory components capable of making predictive decisions in order to avoid the appearance of congestion problems.

In this context, we present B-Neck, a distributed optimization algorithm that can be used as the basic building block for the new generation of proactive and anticipatory congestion control protocols. B-Neck computes proactively the optimal sessions' rates independently of congestion signals iterating rapidly until converging to the optimal solution.

B-Neck applies max-min fairness as optimization criterion, since it is often used in traffic engineering as a way of fairly distributing a network capacity among a set of sessions. The max-min fairness criterion has gained wide acceptance in the networking community and is actively used in traffic engineering and in the modeling of network performance (Bertsekas and Gallager (1992), Nace and Pióro (2008)) as a benchmarking measure in different applications such as routing, congestion control, and performance evaluation. A paradigmatic example of this is the objective function of Google traffic engineering systems in their globally-deployed software defined WAN, which delivers max-min fair bandwidth allocation to applications (Jain et al. (2013)). Max min fairness is closely related to max-min and min-max optimization problems that are extensively studied in the literature. Intuitively, to achieve max-min fairness first the total bandwidth is distributed equally among all the sessions on each link. Then, if a session can not use its allocated bandwidth due to restrictions arising elsewhere on its path, then the residual bandwidth is distributed between the other sessions. Thus, no session is penalized, and a certain minimum quality of service is guaranteed to all sessions. More precisely, max-min fairness takes into account the path of each session and the capacity of each link. Thus each session s is assigned a transmission rate λ_s so that no link is overloaded, and a session could only increase its rate at the expense of a session with the same or smaller rate. In other words, max-min fairness guarantees that no session s can increase its rate λ_s without causing another session s' to end up with a rate $\lambda_{s'} < \lambda_s$.

Up to the date, all proposed proactive max-min fair algorithms require packets being continuously transmitted to compute and maintain the max-min fair rates, even when the set of sessions does not change (e.g., no session is arriving nor leaving). One of the key features of B-Neck is that, in absence of changes (i.e., session arrivals or departures), it becomes quiescent. The quiescence property guarantees that, once the optimal solution is achieved and the max-min fair rates have been assigned, the algorithm does not need to generate (nor assume the existence of) any more control traffic in the network. Moreover, B-Neck reacts to variations in the environment, and so, changes in the sessions (e.g., arrivals and departures) reactivate the algorithm, and eventually the new sending rates are found and notified. As far as we know, this is the first quiescent distributed algorithm that solves the max-min fairness optimization problem. In an exponentially growing IoT scenario of connected nodes (Cisco and Ericsson predict around 30 billions of connected devices by 2020) where different strategies and algorithms are required for energy-efficiency, B-Neck offers, due to its quiescence, an advantageous alternative to the rest of distributed max-min fair algorithms that need to periodically inject traffic into the network to recompute the max-min fair rates.

This paper proposes as novelty two theoretical contributions jointly with an in-depth experimental analysis of the B-Neck algorithm. We formally prove that B-Neck is correct, and second, an upper bound for its convergence time is obtained. Additionally, extensive simulations were conducted to validate the theoretical results and compare B-Neck with the most representative competitors. In these experiments we show that B-Neck behaves nicely in the presence of sessions arriving and departing, and its convergence time is in the same range as that of the fastest (non-quiescent) distributed max-min fair algorithms. These properties encourage to utilize B-Neck as the basic building block of proactive and anticipatory congestion control protocols.

1.1. Related work

We are interested in solving the max-min fair optimization problem in a packet network with given link capacities to compute the max-min fair rate allocation for single path sessions. These rates can be efficiently computed in a centralized way using the Water-Filling algorithm (Bertsekas and Gallager (1992), Nace and Pióro (2008)). From a taxonomic point of view, centralized and distributed algorithms have been proposed. In addition, max-min fair algorithms may also be classified in those which need the routers to store per-session state information, and those which only need a constant amount of information per router. To our knowledge, the proposals of Hahne and Gallager (1986) and Katevenis (1987) were the first to apply max-min fairness to share out the bandwidth, among sessions, in a packet switched network. No max-min fair rate is explicitly calculated, and a window-based flow control is needed in order to control congestion. Per-session state information, which is updated by every data packet processed, is needed at each router link. Additionally, a continuous injection of control traffic is needed to maintain the system in a stable state.

Then, with the surge of ATM networks, there were a collection of distributed algorithms proposed to compute the rates to be used by the virtual circuits in the Available Bit Rate (ABR) traffic mode (Afek, Mansour, and Ostfeld (2000), Bartal, Farach-Colton, Yooseph, and Zhang (2002), Charny, Clark, and Jain (1995), Kalyanaraman, Jain, Fahmy, Goyal, and Vandalore (2000), Cao and Zegura (1999), Hou, Tzeng, and Panwar (1998), Tsai and Kim (1999), Tzeng and Sin (1997)). The computed rates were in fact max-min fair. These algorithms assign the exact max-min fair rates using the ATM explicit End-to-End Rate-based flow-Control protocol (EERC). In this protocol, each source periodically sends special Resource Management (RM) cells. These cells include a field called the Explicit Rate field (ER), which is used by these algorithms to carry per-session state information (e.g., the potential max-min fair rate of a session). Then, router links are in charge of executing the max-min fair algorithm. The EERC protocol, jointly with the former distributed max-min fair algorithms, can be con-

sidered the first versions of proactive congestion control protocols, as they explicitly compute rates independently of congestion signals. The first algorithm with a correctness proof was due to Charny et al. (1995). This algorithm was extended later to consider minimum and maximum rate constraints by Hou et al. (1998). A problem in Charny's algorithm with pseudo-saturated links was unveiled by Tsai and Kim (1999). Additionally, in this article, the authors proposed a centralized way to calculate the max-min fair assignments, they suggested a possible parallelization of this algorithm, and demonstrated several formal properties of it. Later, in Tsai and Iyer (2000), the concept of constraint precedence graph (CPG) of bottleneck links was introduced and, in Ros and Tsai (2001), it was proved that the convergence of max-min rate allocation satisfies a partial ordering on the bottleneck links. They proved that the convergence time of any max-min fair optimization algorithm is at least $(L-1)T$, where L is the number of levels in the CPG graph and T is the time required for a link to converge once all its predecessor links have converged. In fact, the upper bound for the convergence time of B-Neck also depends linearly on the number of levels of the CPG, with $T = 4RTT$, where RTT stands for Round Trip Time (i.e., the length of time it takes for a packet to go from source to destination and back). The algorithms of Tsai and Iyer (2000) and Ros and Tsai (2001) exhibit good convergence speed, that depends linearly on the number of bottleneck levels of the network. (Informally, bottlenecks are links that limit the sessions' rates.) It should be noted that all the abovementioned distributed algorithms use session information on the routers.

Cobb and Gouda (2008) proposed a distributed max-min fair algorithm that does not need per-session information at the routers. Instead, this algorithm depends on a predefined constant parameter T that must be greater than the protocol packet RTT. However, it is not easy to upper bound this value because protocol and data packets share network links, and the RTT will grow when congestion problems arise. Moreover, the experiments in Mozo, López-Presa, and Fernández Anta (2012) showed that, in some non-trivial scenarios, this algorithm did not always converge. Finally, other reduced-state algorithms, like those of Awerbuch and Shavitt (1998) and Afek et al. (2000), only compute *approximations* of the rates. As was shown by Afek, Mansour, and Ostfeld (1999), approximations can lead to large deviations from the max-min fair rates, since they are sensitive to small changes. In a recent work, Mozo et al. (2012) proposed a proactive congestion control protocol in which rates are computed with a scalable and distributed max-min fair optimization algorithm called *SLBN*. Scalability is guaranteed because routers only maintain a constant amount of state information (only three integer variables per link) and only incur a constant amount of computation per protocol packet, independently of the number of sessions that cross the router.

An alternative approach to attempt converging to the max-min fair rates is using algorithms based on reactive closed-control-loops driven by congestion signals. Some research trends have proposed this approach to design explicit congestion control protocols, like Kushwaha and Gupta (2014). In these proposals, the information returned to the source nodes from the routers (e.g., an incremental window size or an explicit rate value) allows the sessions to know approximate values that eventually converge to their max-min fair rates, as the system evolves towards a stable state. In this case, it is not required to process, classify or store per-session information when a packet arrives to the router, and it is guaranteed that the max-min fair rate assignments are achieved when controllers are in a stable state. Thus, scalability is not compromised when the number of sessions that cross a router link grows. For instance, XCP (Katabi, Handley, & Rohrs (2002)) was designed to work well in networks with large bandwidth-delay products. It computes, at each link, window changes which are provided to the sources. However, it was shown in Dukkipati, Kobayashi, Zhang-Shen, and McKeown (2005) that XCP convergence speed can be very slow, and short time duration flows could finish without reaching their fair rates. RCP (Dukkipati et al. (2005)) explicitly computes the rates sent to the sources, what yields more accurate congestion information. Additionally, the computation effort needed in router links per arriving packet is significantly smaller than in the case of XCP. However, in Jain and Loguinov (2007) it was shown that RCP does not always converge, and that it does not properly cope with a large number of session arrivals. Jain and Loguinov (2007) propose PIQI-RCP as an alternative to RCP, trying to alleviate the above drawbacks by being more careful when computing the session rates, considering recent rate assignments history. Unfortunately, all these proposals require processing each data packet at each router link to estimate the fair rates, what hampers scalability. Moreover, we have experimentally observed that they often take very long, or even fail, to converge to the optimal solution when the network topology is not trivial. Additionally, they tend to generate significant oscillations around the max-min fair rates during transient periods, causing link overshoots. A link overshoot scenario implies, sooner or later, a growing number of packets that will be discarded and retransmitted and, in the end, the occurrence of congestion problems. All these problems are mainly caused by the fact that, unlike implicitly assumed, data from different sessions containing congestion signals arrive at different times (due to different and variable RTT) and, hence, the rates (based on the estimation of the number of sessions crossing each link) are computed with data which is not synchronously updated. Moreover, when congestion problems appear, the variance of the RTT distribution increases significantly generating bigger oscillations around the max-min fair rates.

It must be noted that none of these proactive and reactive approaches is quiescent (as B-Neck) and hence, all of them must inject control packets into the network at small intervals to preserve the stability of the system. Being quiescent means that, in absence of changes in the network, once the max-min sessions' rates have been computed, the algorithm stops generating network traffic. To the best of our knowledge, B-Neck is the unique distributed algorithm using max-min fair as the optimization criterion that is also quiescent. This property can be advantageous in practical deployments in which, e.g., energy efficiency needs to be considered.

In addition, only a few of the proactive proposals (Bartal et al. (2002), Tsai and Kim (1999), Charny et al. (1995) and Ros and Tsai (2001)) have formally proved their correctness and/or their convergence. In this paper, we formally prove B-Neck correctness and convergence and obtain an upper bound for B-Neck convergence time. We have observed in our experiments that B-Neck convergence time is in the same range as that of the fastest (non-quiescent) distributed max-min fair algorithms (Bartal et al. (2002)) and faster that any of the reactive closed-control-loops proposals, which were observed to not converge many times.

Finally, recent related works explore different versions of the network congestion control problem applying other optimization algorithms. In Yang, Wu, Dai, and Chen (2011) the authors present a joint congestion control and processor allocation algorithm for task scheduling in grid over Optical Burst Switched networks in which parameters from resource layer are abstracted and provided to a crosslayer optimizer to maximize user's utility function. The proposed non-linear optimizer is executed in a central entity and therefore, its scalability could be compromised in an Internet scenario composed of hundreds of thousands of routers. On the contrary, B-Neck is fully distributed and therefore its scalability should not be compromised in an Internet scenario. In Chen, Kuo, Yan, and Liao (2009) the authors proposed a network congestion control schema based on active queue management (AQM) mecha-

nisms. The manager of the AQM systems is a proportional-integral-derivative (PID) controller that is deployed in each router. They present as novelty an improved genetic algorithm to derive optimal and near optimal PID controller gains. It is worth mentioning that none of the papers provides any formal proof of the correctness and convergence of the proposed algorithms as B-Neck does. In addition, their experimental simulations are restricted to a dozen of routers in the most complex scenario and B-Neck on the contrary, has been experimentally demonstrated in Internet-like topologies with up to 11,000 routers.

1.2. Contributions

In this paper we present a distributed optimization algorithm that can be used as the basic building block for the new generation of proactive and anticipatory congestion control protocols. As far as we know, this algorithm that we call B-Neck is the first max-min fair quiescent algorithm. That B-Neck is quiescent means that it stops creating traffic when it has converged to the max-min rates, as long as there are no changes in the sessions. This contrasts with prior proactive and reactive algorithms that require a continuous injection of traffic to maintain the max-min fair rates. Hence, B-Neck uses a bounded number of packets to iteratively converge to the optimal solution of the problem. In addition, each node only requires information of the sessions that traverse it. When changes in the sessions occur, B-Neck is reactivated in order to recompute the rates and asynchronously inform the affected sessions of their new rates (i.e., sessions do not need to poll the network for changes).

The first contribution of this paper is the formal proof of two theoretical properties of B-Neck. First, we show its *correctness*, which intuitively means that B-Neck eventually finds the max-min fair rates of all the sessions, as long as sessions do not change. Second, we show *quiescence*, which intuitively means that B-Neck eventually stops injecting traffic into the network once the max-min fair rates have been found. If being in this state the set of sessions or their bandwidth requirements change, B-Neck becomes active again to find the new set of max-min fair rates.

In addition, speed convergence of B-Neck is studied and an upper bound on the time to achieve quiescence is obtained. We prove that, if the set of sessions does not change for a long enough period of time, then every session achieves its max-min fair rate and the network becomes quiescent in at most $4 \cdot m \cdot RTT$, where m is the number of bottleneck levels in the network, and RTT is the largest round-trip time of any session in the network.

The second contribution of this paper is an in-depth experimental analysis of the B-Neck algorithm. The properties of B-Neck have been tested with extensive simulations on network topologies that are similar to the Internet. The experiments use networks of very different sizes (from 110 to 11,000 routers) on which different numbers of sessions are routed (up to hundreds of thousands). In addition, two paradigmatic setups are used, representing local area networks (LAN) and wide area networks (WAN). Our first set of experiments has shown that B-Neck becomes quiescent very quickly, even in the presence of many interacting sessions. In the second set of experiments we have compared the performance of B-Neck with several well-known max-min fair proactive and reactive algorithms, which were designed to be used as part of congestion control mechanisms. We can conclude the analysis of this set of experiments observing that (a) B-Neck behaves rather efficiently both in terms of time to quiescence, and in terms of average number of packets per session, (b) B-Neck convergence to the max-min fair rates is realized independently of the churn session pattern, with error distributions at sources and links that are nearly constant during transient periods, and achieving a nearly full utilization of links but never overshooting them on average, and (c) as soon as sessions converge to their max-min fair rates, B-Neck stops injecting packets to the network and the total traffic generated by B-Neck decreases dramatically.

1.3. Structure of the rest of the paper

In the following section we provide basic definitions and notations. Then, the algorithm B-Neck is presented in Section 3. Its correctness is proved in Section 4, along with a convergence upper bound. In Section 5 experimental results are shown. Finally, in the last section we summarize the conclusions.

2. Definitions and notation

In this paper we consider networks formed by hosts and routers, connected with directed links. Links between nodes may be asymmetric. That means that, given two nodes u and v, the bandwidth of link (u, v) may be different from the bandwidth of link (v, u). Hence, a network is modeled as a directed graph $G = (V, E)$, where V is its set of routers and hosts (nodes), and E is the set of directed links. For simplicity, we assume that the network is simple (i.e., it has no loops nor multiple edges between two nodes), but every pair of connected nodes has a bidirectional link (i.e., two directed links in opposite directions). This is common in real networks and B-Neck relies on this property of the network since it must be able to send messages between two nodes in both directions. Each directed link $e \in E$ has a bandwidth C_e allocated to data traffic[1].

In a network, the hosts are the nodes in which sessions start and end. Hence, for every session, there is a host (the *source node*) where the session starts and another host (the *destination node*) where it ends. Hosts are connected via a subnetwork of routers in which the routing from the source node to the destination node occurs. The route of a session s in the network is a static path $\pi(s)$, that starts in the source node of s and ends in its destination node. For simplicity, we restrict that every host is connected to exactly one router. Additionally, each host can only be the source node of one session, since a host with two sessions may be modeled by two virtual hosts, each of them with one session, connected to the same router.

A packet corresponding to session s is said to be sent *downstream* when it traverses a link in this path (towards the destination node), and it is said to be sent *upstream* when it traverses a link in the reverse path of $\pi(s)$ (towards the source node). Reliable communication channels are assumed to be available to transmit B-Neck packets. If reliability is not guaranteed, classical techniques can be used to cope with communication errors, keeping the state consistent in nodes. Besides, we will assume that the latency of each link is fixed, and symmetric. Thus, for each session s, we denote by $RTT(s)$ the time a packet takes to go from the source node of session s to its destination and back. This time includes the time needed to process the packet at each link $e \in \pi(s)$. Let S be the set of active sessions in the network, then we define $RTT = \max_{s \in S} RTT(s)$.

A max-min fair algorithm has to assign a rate to each session that distributes the available bandwidth in a max-min fair fashion. Fig. 1 illustrates the difference between proportional allocation and max-min fair allocation. As it can be seen, the link between $n1$ and $n2$ is shared by sessions $S1$, $S2$ and $S3$. A proportional allocation would assign 1/3 of the link capacity to each of these sessions. In addition, the link between $n2$ and $n3$ is shared by sessions $S3$ and $S4$. The proportional allocation would assign 1/2 of the link capacity to $S3$ and $S4$. If we assume for simplicity that each link

[1] For simplicity, we assume that control traffic does not consume any bandwidth.

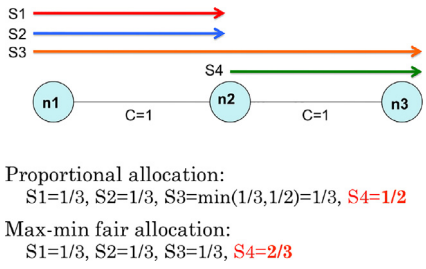

Fig. 1. Max-min fair allocation versus proportional allocation.

has a capacity of 1 Mbps, $S3$ is assigned 1/2 Mbps in the first link and 1/3 Mbps in the second. Regarding that the effective transmission rate of a session will be limited by the smaller of the rate assignments at the links in its path, $S3$ transmission rate will be limited at the first link by 1/3 Mbps. Note that 1/6 of the second link capacity will be unused $(1 - 1/3_{(S3)} - 1/2_{(S4)} = 1/6)$ because each link computes independently their rate assignments and no information is shared among links when proportional assignment is applied. On the contrary, when max-min fair allocation is used, session rate assignments are shared among all the links in the path of each session. In our example, the second link knows that $S3$ is only using 1/3 Mbps at the first link and therefore, the rest of the sessions crossing the second link ($S4$) can take advantage of the remaining bandwidth (2/3). In summary, $S4$ will be assigned 2/3 Mbps when max-min fair allocation is applied and only 1/2 Mbps when proportional allocation is applied. Moreover, no bandwidth capacity is wasted at the second link when max-min fair allocation is used, but 1/3 Mbps would be unused at the second link if proportional allocation were applied.

In order to provide flexibility, in this paper we allow sessions to provide B-Neck with the maximum bandwidth they request (which can be ∞). Hence, sessions want to get the maximum possible rate up to the maximum bandwidth they request. This value can be changed at any point of the execution. Then, the primitives that sessions use to interact with B-Neck allow to signal the session arrival, departure, and change of bandwidth request as follows.

- *API.Join*(s, r): This call is used by session s to notify B-Neck that it has joined the system and requests a maximum bandwidth of r.
- *API.Leave*(s): This call is used by session s to notify B-Neck that it has terminated.
- *API.Change*(s, r): This call is used by session s to notify B-Neck that it has changed the maximum requested bandwidth to r.
- *API.Rate*(s, λ): This upcall is used by B-Neck to notify session s that its max-min-fair rate is λ.

A session s that has joined (calling *API.Join*(s, r)) and has not terminated yet (calling *API.Leave*(s)) is said to be *active*. B-Neck expects from the session to call these primitives consistently, e.g., no active session calls *API.Join*, and only active sessions call *API.Leave* and *API.Change*. Similarly, B-Neck must guarantee that *API.Rate* is called on active sessions.

Knowing when a session is active and when it is no longer active (in order to allocate and deallocate resources to it, the so-called use-it-or-lose-it principle) is key to the success of a rate control mechanism. Therefore, from a user point of view, our model could be seen as unrealistic because flows do not know if they are active or not. Our proposal is to take hosts away from the model, and to delegate in the first router (in the session path) the responsibility of implementing the above primitives. As it is commonly assumed (e.g. Zhang, Duan, Gao, and Hou (2000), and Kaur and Vin (2003)), access routers maintain information about each individual flow, while core routers, for scalability purposes, do not. Hence, explicitly signaling the arrival and departure of sessions does not compromise the scalability of the access router (e.g., an xDSL home router), since it only has to cope with a small number of sessions. Therefore, it is easy for this kind of routers to execute *API.Join* when they detect a new flow, or *API.Leave* when an existing flow times out. Stream oriented flows (e.g., TCP) explicitly signal connection establishment and termination. Hence, these routers can execute the corresponding primitives when they detect the corresponding packets (e.g., *SYN* and *FIN* in TCP). On their hand, datagram oriented flows (e.g., UDP) can be tracked down with the help of an array of active flows (e.g. identified by source-destination pairs), and an inactivity timer for each flow. Additionally, the source router can measure in real time differences between the assigned and the actual bandwidth used by a session, and execute *API.Change* if the actual rate is significantly lower than the rate assigned by the control congestion mechanism.

We say that the network is in *steady state* in a time period if no session calls any of the above primitives *API.Join*, *API.Leave* and *API.Change* in the interval. Then, the set of active session S^* does not change in the interval. We will denote then for each link e as S_e^* the set of sessions in S^* that cross e, and for each $s \in S^*$, the maximum bandwidth requested by s as r_s. For the sake of simplicity we will assume that the first link e in the path of a session s is $D_s = \min(C_e, r_s)$, and hence the maximum requested bandwidth is already limited there. Finally, λ_s^* denotes the max-min fair rate of session s.

Definition 1. A link e is a *bottleneck* if $\sum_{s \in S_e^*} \lambda_s^* = C_e$. A link e is a *bottleneck of a session* $s \in S_e^*$ if e is a bottleneck and $\forall s' \in S_e^*, \lambda_{s'}^* \leq \lambda_s^*$. A link e is a *system bottleneck* if it is a bottleneck of every session $s \in S_e^*$.

Let $s \in S_e^*$. If e is a bottleneck of s, then $B_e^* = \lambda_s^*$, and we say that s is *restricted* at link e. Otherwise we say that s is *unrestricted* at link e. In any max-min fair system, every session is restricted in at least one link and, hence, it has at least one bottleneck. Bertsekas and Gallager (1992) showed that any max-min fair system has at least one system bottleneck.

The set of all the sessions which are restricted at link e is denoted by R_e^*. The set of all the sessions in S_e^* that are unrestricted at link e is denoted by F_e^*. Then, $S_e^* = R_e^* \cup F_e^*$. If e is a bottleneck, then it is easy to see that all the sessions in R_e^* have the same max-min fair rate. We denote this rate as B_e^* and call it the *bottleneck rate* of e.

Observation 1. *If e is a bottleneck, then $B_e^* = (C_e - \sum_{s \in F_e^*} \lambda_s^*)/|R_e^*|$, and for all $s \in F_e^*$, $\lambda_s^* < B_e^*$. If e is a system bottleneck, then $F_e^* = \emptyset$ and $B_e^* = C_e/|R_e^*| = C_e/|S_e^*|$. If e is not a bottleneck, then $\sum_{s \in S_e^*} \lambda_s^* < C_e$.*

Note that, if e is a bottleneck, then, from Definition 1, $C_e = \sum_{s \in S_e^*} \lambda_s^* = B_e^* |R_e^*| + \sum_{s \in F_e^*} \lambda_s^*$. Hence $B_e^* = (C_e - \sum_{s \in F_e^*} \lambda_s^*)/|R_e^*|$. Since $C_e \geq \sum_{s \in S_e^*} \lambda_s^*$, if e is not a bottleneck, then $C_e > \sum_{s \in S_e^*} \lambda_s^*$.

Claim 1. *If e is a bottleneck and $F_e^* \neq \emptyset$, then $B_e^* > C_e/|S_e^*|$.*

Proof. If e is a bottleneck, then $C_e = B_e^*|R_e^*| + \sum_{s \in F_e^*} \lambda_s^*$. From Observation 1, for all $s \in F_e^*$, $\lambda_s^* < B_e^*$. Hence $C_e < B_e^*|R_e^*| + B_e^*|F_e^*| = B_e^*(|R_e^*| + |F_e^*|) = B_e^*|S_e^*|$. Thus $B_e^* > C_e/|S_e^*|$. □

Claim 2. *If e is a bottleneck and $F_e^* \neq \emptyset$, then there is a session $s \in F_e^*$ such that $\lambda_s^* < C_e/|S_e^*|$.*

Proof. If e is a bottleneck, then $C_e = B_e^*|R_e^*| + \sum_{s \in F_e^*} \lambda_s^*$. By the way of contradiction, assume that for all $s \in F_e^*$, $\lambda_s^* \geq (C_e/|S_e^*|)$. Then, $\sum_{s \in F_e^*} \lambda_s^* \geq |F_e^*|(C_e/|S_e^*|)$. Hence, $C_e \geq B_e^*|R_e^*| + |F_e^*|(C_e/|S_e^*|)$.

Thus $B_e^* \leq (C_e - |F_e^*|(C_e/|S_e^*|))/|R_e^*| = ((C_e|S_e^*| - C_e|F_e^*|)/|S_e^*|)/|R_e^*|$. Since $|S_e^*| - |F_e^*| = |R_e^*|$, then $B_e^* \leq ((C_e|R_e^*|)/|S_e^*|)/|R_e^*| = C_e/|S_e^*|$. However, from Claim 1, $B_e^* > C_e/|S_e^*|$, so we have reached a contradiction. □

The following property serves as the base for computing the max-min fair rates of the sessions. It comes directly from Observation 1 and the fact that every session has, at least, one bottleneck.

Property 1. *For each session* $s \in S^*$, $\lambda_s^* = \min_{e \in \pi(s)} B_e^*$.

Now we define the concept of *dependency* among sessions and among links. This concept will be essential in the analysis of B-Neck, since it will be shown that the sessions will get their max-min fair assignments in the (partial) order induced by dependency.

Definition 2. A session s *depends* on another session s' if there is a link e such that $s \in S_e^*$, $s' \in S_e^*$ and $\lambda_s^* \geq \lambda_{s'}^*$. Conversely, s' *affects* s.

Definition 3. A link e *depends* on another link e' if there is a session $s \in S_e^*$ that depends on another session $s' \in R_{e'}^*$. Conversely, e' affects e.

Note that two sessions might not be comparable for dependency. For example, two sessions s and s' such that $\pi(s) \cap \pi(s') = \emptyset$ would not be comparable. Similarly, a link e would not be comparable with a link e' if for all $s \in S_e^*$, $s' \in R_{e'}$, s and s' are not comparable. The following observation follows directly from the previous definitions.

Observation 2. *If a bottleneck e depends on another bottleneck e', then $B_e^* \geq B_{e'}^*$. If e depends on e' but e' does not depend on e, then $B_e^* > B_{e'}^*$. If e and e' depend among each other, then $B_e^* = B_{e'}^*$.*

For each link $e \in E$, let $DB(e) = \{e' \in E : (e' \text{ affects } e) \wedge (e \text{ does not affect } e')\}$. If $DB(e) = \emptyset$, then e is a *core bottleneck*. The set of all the bottlenecks of the network is denoted by *BS*. Note that every core bottleneck is a system bottleneck. However, not every system bottleneck is a core bottleneck.

Definition 4. The *bottleneck level* of a bottleneck e, denoted by $BL(e)$, is computed as follows. If e is a core bottleneck, then $BL(e) = 1$. If e is not a core bottleneck, then $BL(e) = 1 + \max_{e' \in DB(e)} BL(e')$. The bottleneck level of the network is computed as $BL = \max_{e \in BS} BL(e)$.

The bottleneck level of the network will determine the time needed by B-Neck to converge to the optimal solution computing the max-min fair rates of the sessions and become quiescent.

3. B-Neck algorithm

In this section we describe the basic intuition of how B-Neck works. For that, we start by describing an algorithm that is centralized, but that roughly operates in a similar manner as B-Neck, to then fill the details on how it is possible to move from a centralized to a distributed algorithm.

3.1. Centralized B-Neck

Before tackling the problem of computing max-min fair rates with a distributed algorithm, we will introduce a centralized iterative algorithm that solves the max-min fair optimization problem having global knowledge of the network (and sessions) configuration. This simple algorithm illustrates how these rates may be computed in a network that is in a steady state. This algorithm computes the max-min fair rates in a form that is similar to the Water-Filling algorithm of Bertsekas and Gallager (1992). Centralized B-Neck is not intended for being used in real deployments since having a centralized controller with global knowledge of the network configuration is not realistic. However, this algorithm serves as a reference to evaluate the precision of other algorithms for computing max-main fair rates. The Centralized B-Neck algorithm is presented in Fig. 2. This algorithm converges to the max-min fair rates by discovering bottlenecks one after the other, starting with the bottlenecks of smallest rates and always finding bottlenecks of the next larger rate. In every iteration, the *estimated bottleneck rate* of a link e (with $R_e \neq \emptyset$) is computed as $B_e = (C_e - \sum_{s \in F_e} \lambda_s^*)/|R_e|$. For each link e, it recomputes variables F_e and R_e, if necessary, at each iteration. When the algorithm ends its execution, $F_e = F_e^*$, $R_e = R_e^*$ and $B_e = B_e^*$ for each link e, and λ_s^* is the max-min fair rate for each session $s \in S^*$.

```
1  for each e ∈ E do
2      R_e ← S_e^*
3      F_e ← ∅
4  end for
5  L ← {e ∈ E : R_e ≠ ∅}
6  while L ≠ ∅ do
7      for each e ∈ L do
8          B_e ← (C_e − Σ_{s∈F_e} λ_s^*)/|R_e|
9      end for
10     B ← min_{e∈L}{B_e}
11     L' ← {e ∈ L : B_e = B}
12     X ← ⋃_{e∈L'} R_e
13     for each s ∈ X do
14         λ_s^* ← B
15     end for
16     for each e ∈ L \ L' do
17         F_e ← F_e ∪ (R_e ∩ X)
18         R_e ← R_e \ F_e
19     end for
20     L ← {e ∈ (L \ L') : R_e ≠ ∅}
21 end while
```

Fig. 2. Centralized B-Neck algorithm.

Note that, from Observation 1, for all system bottleneck e, $B_e^* = C_e/|R_e^*|$, $F_e^* = \emptyset$ and $S_e^* = R_e^*$. Since, in the first iteration, for all system bottleneck e, $F_e = \emptyset = F_e^*$ and $R_e = S_e^* = R_e^*$, then $B_e = (C_e - \sum_{s \in F_e} \lambda_s^*)/|R_e| = C_e/S_e = C_e/|R_e^*| = B_e^*$, i.e. their estimated bottleneck rates correspond to their correct bottleneck rates. The problem is that it is not possible to determine, in this first iteration, which links are the system bottlenecks, since F_e is empty for all the links in the system. Then, we take the minimum among the estimated bottleneck rates, $B \leftarrow \min_{e \in L}\{B_e\}$. Thus, the links in L' are definitely system bottlenecks, since their sessions cannot be restricted in any other link, although there might be more system bottlenecks that will be discovered lately. Then, all the sessions s that cross the links in L' are assigned their max-min fair rates $\lambda_s^* = B$, which correspond to the bottleneck rate of these links. Once these sessions have got their rates assigned, a new network configuration is generated. First, the sessions that are restricted in the bottlenecks with the lowest bottleneck rate are moved to the sets of unrestricted sessions in all the other links of their paths. Then, the bottlenecks in L' are removed from L for the next iteration, along with those links which have all their sessions restricted somewhere else (and, hence, are not bottlenecks).

At each iteration, the bottlenecks with the minimum bottleneck rate, among those in L, are identified, and the max-min fair rate of their restricted sessions is correctly computed. Note that for all $e \in L'$, $F_e = F_e^*$, $R_e = R_e^*$ and for all $s \in F_e$, the value of λ_s^* is correct. Hence, for all the links $e \in L'$, $B_e = (C_e - \sum_{s \in F_e} \lambda_s^*)/|R_e| = (C_e - \sum_{s \in F_e^*} \lambda_s^*)/|R_e^*| = B_e^*$. This process continues until there are no more bottlenecks to be identified (i.e., $L = \emptyset$). Since the set of links of the network is finite and, at each iteration, at least one bottleneck is identified and removed from L, this process ends in a finite number of iterations. Hence, this centralized algorithm correctly computes the max-min fair rates of all the sessions in the system. The main loop of the algorithm is executed at most once

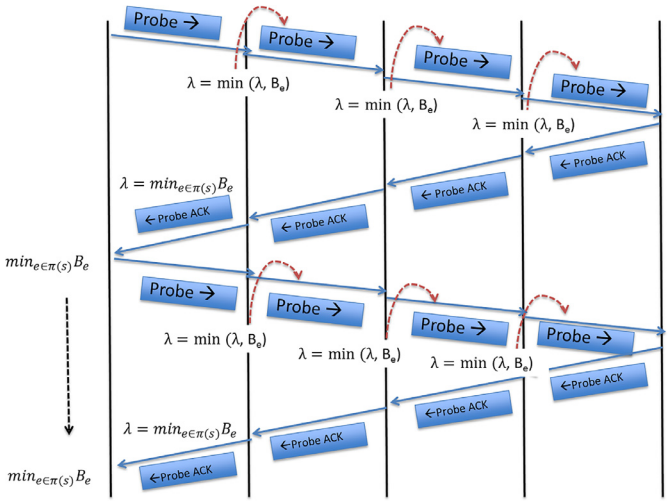

Fig. 3. EERC protocol.

per link in the network since, at each iteration, at least one link is removed from set L (initially $L = E$ in the worst case). The internal loops iterate through L and X which is a subset of S^*. Thus, its computational complexity is $O(|E|(|E| + |S^*|))$.

As previously mentioned, B-Neck follows a similar logic to that used in the Centralized B-Neck algorithm. At each link e, the sets F_e and R_e are maintained in order to compute its estimated bottleneck rate B_e, which is notified to the sessions that cross it. While Centralized B-Neck computes the bottleneck rates iteratively in ascending order of the bottleneck rates of the links, the distributed algorithm computes the bottleneck rates in ascending order of the bottleneck level of the links. As it will be seen, it makes use of Property 1 to compute the rates of the sessions. Note that, since the computation of the estimated bottleneck rates is performed concurrently at the different links, several bottlenecks might be detected at the same time, no matter their bottleneck rate (unlike the centralized algorithm). However, the dynamic and distributed nature of B-Neck also generates several concurrency issues that must be solved to achieve convergence. Each time a bottleneck is detected, the affected links are notified, so they can update their sets of unrestricted and restricted sessions, similarly to the movement of sessions from R_e to F_e for each link $e \in L$ in the centralized algorithm. Besides, when a link has been identified as a bottleneck, and the sessions that are restricted at that link are notified, the link stops recomputing its bottleneck rate, what resembles the elimination of nodes from L in the centralized algorithm.

3.2. A global perspective of B-Neck

B-Neck is an event driven, distributed and quiescent algorithm that solves the max-min fair optimization problem computing max-min fair rate allocations and notifying them to the sessions, in a network environment where sessions can asynchronously join and leave the network, or change their maximum desired rate.

B-Neck is executed in every network link, and in the hosts that are source and destination nodes of every session. The processes executed in these elements communicate with B-Neck packets. As described above, the sessions use the primitives *API.Join*, *API.Leave*, and *API.Change* to interact with the algorithm. B-Neck resembles EERC protocols because it relies on the session sources performing *Probe* cycles to discover their max-min fair rates. In Fig. 3 we show how a *Probe* cycle traverses the links in the path of a session computing the minimum value of B_e. Note also that *Probe* cycles do not overlap. The estimated rate obtained in a *Probe* cycle is used as the initial value of λ in the next one.

However, unlike EERC protocols, B-Neck sources do not generate periodic *Probe* cycles because, if a link detects a possible change in the available bandwidth of a session, it notifies that session that it needs to perform a new *Probe* cycle to recompute its assigned rate.

The packets used by the B-Neck algorithm (B-Neck packets) are the following.

- *Join*(s, λ, η): This packet is sent from the source to the destination (downstream) following the path of session s to notify the links in the path and the destination node that s has joined the system. The parameters λ and η carried in the packet are the smallest estimated bottleneck rate found in the path so far, and one of the links that have that estimated bottleneck rate.
- *Probe*(s, λ, η): This packet is sent downstream on the path of session s to notify the links in the path and the destination node that the rate for s has to be recomputed.
- *Response*(s, τ, λ, η): This packet is sent from the destination to the source (upstream) following the path of session s. It carries the smallest bottleneck rate λ that was found downstream by a *Join* or *Probe* packet, some link e with that rate, and a parameter τ that tells the source the next action that has to be taken.
- *Update*(s): This packet is sent upstream to notify the source of session s that it has to start a new *Probe* cycle (defined below).
- *Bottleneck*(s): This packet is sent upstream to notify the links in the path and the source node of session s that the current rate is the max-min fair rate.
- *SetBottleneck*(s, β): This packet is sent downstream to notify the links in the path and the destination node of session s that the current rate is the max-min fair rate. As a consequence, any link e that does not restrict s must move it from R_e to F_e. The flag β records whether any of the traversed links was a bottleneck.
- *Leave*(s): This packet is sent downstream to notify the links in the path and the destination node of session s that the session has left the system, and hence they can remove all data regarding that session. This may also trigger other necessary actions, like sending an *Update* packet in other sessions.

Like other distributed algorithms that compute max-min fair rates, B-Neck needs to keep some information at the links. In particular, every link e has two sets R_e and F_e (like the centralized algorithm), that maintain the sessions restricted at e and those that are not restricted at e, respectively. Additionally, it has a state variable $\mu_s^e \in \{\text{IDLE, WAITING\_PROBE, WAITING\_RESPONSE}\}$ for every session s such that $e \in \pi(s)$. Finally, for the same sessions s it has the *assigned rate* λ_s^e (which is relevnat only when $s \in F_e$ or $\mu_s^e = \text{IDLE}$). As described in the centralized algroithm, link e can compute its estimated bottleneck rate at any point in time as $B_e = (C_e - \sum_{s \in F_e} \lambda_s^e)/|R_e|$.

During a *Probe* cycle of a session s, Property 1 is applied in the following way: at each link e in the path of s, the currently estimated bandwidth λ for session s is compared against the estimated bottleneck level of link e, and λ is set to the minimum of them. Thus, when the *Probe* packet reaches the destination node, $\lambda = \min_{e \in \pi(s)} B_e$.

While in the centralized version the rates assigned to the sessions in F_e are always the max-min fair rates λ_s^*, in B-Neck the rates λ_s^e might not be the max-min fair rates. Despite that, as the most restrictive bottlenecks are discovered, the least restrictive ones will compute their bottleneck rates correctly, and the system will converge to the max-min fair rate assignment.

Note that, when $B_e = B_e^*$ for all link $e \in E$, the value $\lambda = \min_{e \in \pi(s)} B_e$ computed during a *Probe* cycle of a session s corresponds to its max-min fair rate and no more *Probe* cycles for session s will be needed unless the sessions configuration changes.

At the source node of a session s, the session's maximum desired rate $D_s = \min(r, C_e)$ is kept, where e is the link connecting the source node to the first route in $\pi(s)$. This value will be used

to start new *Probe* cycles in the future. Additionally, three flags are used: $bneck\_rcv_s$ which indicates that a max-min-fair assignment has been made to session s (to avoid sending unnecessary *SetBottleneck* packets due to the reception of redundant *Bottleneck* packets), $pending\_probe_s$ which indicates that *API.Change*(s) has been called while a *Probe* cycle for session s was being performed (to force the start of a new *Probe* cycle immediately after the current one ends), and $pending\_leave_s$ which indicates that *API.Leave*(s) has been called while a *Probe* cycle for session s was being performed (to force the sending of a *Leave* packet immediately after the current one ends). Flag $pending\_probe_s$ is considered only after $pending\_leave_s$ has been tested.

Once we have introduced how the max-min fair rates can be computed and the packets used by the distributed algorithm B-Neck, we give a simple intuition of the way it works. B-Neck is formally specified as three asynchronous tasks[2] that run: (1) in the source nodes (those that initiate the sessions), shown in Fig. 5; (2) in the destination nodes, shown in Fig. 6; and (3) in the internal routers to control each network link, shown in Fig. 4. B-Neck is structured as a set of routines that are executed atomically, and activated asynchronously when an event is triggered. This happens when an API primitive is called for a session, or when a B-Neck packet is received. This is specified using **when** blocks in the formal specification of the algorithm.

The source node of a session s starts a *Probe* cycle when it joins, receives an *Update* packet, or the session changes its rate with primitive *API.Change*. The *Probe cycle* is the basic process used by B-Neck to converge to the max-min fair rates. The *Probe* cycle starts by the source node sending a *Probe* packet on the path of session s. (When the session joins, the packet used is *Join*). This packet is processed at every link and resent across the next link until it reaches the destination node. A (or *Join* when a session arrives) cycle starts with the source node sending downstream a (or) packet which, regenerated at each link, traverses the whole path of the session. Then, a *Response* packet is generated at the destination node, and sent upstream (regenerated at each link of the reverse path). The *Probe* (or *Join*) cycle ends when the *Response* packet is processed at the source node. A source node only starts a *Probe* cycle for a session s if there is no cycle already underway, and a *Probe* cycle is always completed. Note that, since new *Probe* cycles are triggered by the network only when a change in the sessions configuration is detected, and only the affected sessions are notified, the traffic generated by B-Neck is very limited (unlike previous distributed algorithms that relayed directly on the sessions periodically polling the network to recompute their rate assignment). Once sessions stop changing (arriving, leaving or changing rate), the network gets eventually stable and then B-Neck becomes quiescent.

The *Response* packets are used to close the *Probe* cycles. They force the links to assign rates to the sessions and detect bottlenecks. The condition for a link e to be a bottleneck is that all unrestricted sessions that cross e satisfy the following. (1) They have completed a *Probe* cycle, (2) are in the IDLE state, and (3) have been assigned the estimated bottleneck rate (see Line RL38). When a link e detects that it is bottleneck, it sends *Bottleneck* packets to all unrestricted sessions, which tell the links in their paths that their rates are the max-min fair rates. The session that is closing the *Probe* cycle is notified with a tag $\tau = \text{BOTTLENECK}$ in the *Response* packet.

The source node of a session s that joins the network sends a *Join* packet downstream. As it traverses the links of the path $\pi(s)$, this packet reports them about the new session s and serves as a *Probe* packet of an initial *Probe* cycle of the session. Every link e in the path adds s to the set R_e, recomputes the bottleneck rates, and sends *Update* packets to the sessions that may have their rates reduced (if any). The source node of a session s that leaves the network sends a *Leave* packet downstream. As it traverses the links of the path $\pi(s)$, these links remove the session from their state, recompute the bottleneck rates, and send *Update* packets to the sessions that may be affected.

When the source node of a session s receives a *Response* packet with $\tau = \text{BOTTLENECK}$ or a *Bottleneck* packet (which are similar events), it sends downstream a *SetBottleneck* packet. As it traverses the path $\pi(s)$, this packet informs the links that the current rate of the session is stable. Additionally, a link e that receives the *SetBottleneck* packet checks if it is a bottleneck for s. If it is not, session s is moved to F_e, the rate B_e is reevaluated, and the sessions that are restricted at e are notified with *Update* packets (so they check if they can increase their rates). If no link in the path is a bottleneck, this is detected at the destination node with the flag β of the *SetBottleneck* packet. Then, an *Update* packet is sent upstream to force a new *Probe* cycle. Note that, during a *Probe* cycle, the session is moved back to R_e at each link in $\pi(s)$ before B_e is recomputed.

The way B-Neck discovers bottlenecks is similar to the way the Centralized algorithm does it. However, the parallelism of B-Neck may allow to reduce the number of iterations to converge. E.g., all core bottlenecks can be found in parallel. Sometimes, due to a transient state, a bottleneck e might be incorrectly flagged, because it depends on a different bottleneck e' that has not been identified yet. Luckily, eventually e' will be identified, *SetBottleneck* packets will be sent, these will trigger *Update* packets, and e will be correctly identified.

4. Proof of correctness of B-Neck

When there is a period of time during which no calls to *API.Join*, *API.Leave* or *API.Change* are executed, we say that the network executing B-Neck is in a steady state during that period of time. In particular, for each session $s \in S^*$, there is a time after which for all $e \in \pi(s)$, $S_e = S_e^*$ permanently and no session $s' \in \cup_{e \in \pi(s)} S_e^*$ changes its requested rate. We call that time the *Steadiness Starting Point* of session s and denote it $SSP(s)$. The Steadiness Starting Point of a link e is $SSP(e) = \max_{s \in S_e^*} SSP(s)$. Similarly, the Steadiness Starting Point of the network is $SSP = \max_{s \in S^*} SSP(s)$.

In this section we prove the correctness of the proposed algorithm B-Neck, i.e., we prove that, when B-Neck becomes quiescent, then each session in the network has been assigned its max-min fair rate. Besides, we give an upper bound on the time B-Neck needs to become quiescent, once the network gets to a steady state. To start with, we are going to prove some basic properties of the algorithm.

Note first that, in algorithm B-Neck, all **when** blocks complete in a finite time, since there are no blocking (waiting) instructions and every loop has a finite number of iterations (they iterate over finite sets R_e or F_e). Hence, packets cannot be processed forever in a **when** block. Thus, we consider that the packets are processed atomically, i.e. we do not consider the state of the network while a packet is being processed, but only when packets have been fully processed.

Claim 3. *Every Probe cycle started for a session is always completed, and two Probe cycles of the same session never overlap.*

Proof. Note that *Join* and *Probe* packets are always relayed at Line RL24 and Line RL58 respectively. When these packets reach the destination node, a *Response* packet is sent upstream at Line DN10 and Line DN14 respectively. Besides, the *Response* packets are always relayed at Line RL46. Finally, new *Probe* cycles are only started when the session joins the network (Line SN20), when

[2] References to lines in these tasks will contain the task acronym (SN, DN, RL).

```
1  task RouterLink (e)
2  var
3      R_e ← ∅; F_e ← ∅
4
5  procedure ProcessNewRestricted()
6      while ∃s ∈ F_e : λ_s^e ≥ B_e do
7          λ_m ← max_{s∈F_e}{λ_s^e}
8          R' ← {r ∈ F_e : λ_r^e = λ_m}
9          F_e ← F_e \ R'; R_e ← R_e ∪ R'
10     end while
11     foreach s ∈ R_e : μ_s^e = IDLE ∧ λ_s^e > B_e do
12         μ_s^e ← WAITING_PROBE
13         send upstream Update (s)
14     end foreach
15 end procedure
16
17 when received Join (s, λ, η) do
18     R_e ← R_e ∪ {s}
19     μ_s^e ← WAITING_RESPONSE
20     ProcessNewRestricted()
21     if λ > B_e then
22         λ ← B_e; η ← e
23     end if
24     send downstream Join (s, λ, η)
25 end when
26
27 when received Response (s, τ, λ, η) do
28     if τ = UPDATE then
29         μ_s^e ← WAITING_PROBE
30     else
31         if ((λ > B_e) ∨ (η = e ∧ λ < B_e)) then
32             τ ← UPDATE
33             μ_s^e ← WAITING_PROBE
34         else // ((λ = B_e) ∨ (η ≠ e ∧ λ ≤ B_e))
35             μ_s^e ← IDLE
36             λ_s^e ← λ
37         end if
38         if ∀r ∈ R_e, μ_r^e = IDLE ∧ λ_r^e = B_e then
39             τ ← BOTTLENECK
40             η ← e
41             foreach r ∈ R_e \ {s} do
42                 send upstream Bottleneck (r)
43             end foreach
44         end if
45     end if
46     send upstream Response (s, τ, λ, η)
47 end when
48
49 when received Probe (s, λ, η) do
50     μ_s^r ← WAITING_RESPONSE
51     if s ∈ F_e then
52         F_e ← F_e \ {s}; R_e ← R_e ∪ {s}
53         ProcessNewRestricted()
54     end if
55     if λ > B_e then
56         λ ← B_e; η ← e
57     end if
58     send downstream Probe (s, λ, η)
59 end when
60
61 when received Update (s) do
62     if μ_s^e = IDLE then
63         μ_s^e ← WAITING_PROBE
64         send upstream Update (s)
65     end if
66 end when
67
68 when received Bottleneck (s) do
69     if μ_s^e = IDLE ∧ s ∈ R_e then
70         send upstream Bottleneck (s)
71     end if
72 end when
73
74 when received SetBottleneck (s, β) do
75     if ∀r ∈ R_e, μ_r^e = IDLE ∧ λ_r^e = B_e then
76         send downstream SetBottleneck (s, TRUE)
77     else if μ_s^e = IDLE ∧ λ_s^e < B_e then
78         R' ← {r ∈ R_e : μ_r^e = IDLE ∧ λ_r^e = B_e}
79         foreach r ∈ R' do
80             μ_r^e ← WAITING_PROBE
81             send upstream Update (r)
82         end foreach
83         R_e ← R_e \ {s}; F_e ← F_e ∪ {s}
84         send downstream SetBottleneck (s, β)
85     else if μ_s^e = IDLE ∧ λ_s^e = B_e then
86         send downstream SetBottleneck (s, β)
87     end if
88 end when
89
90 when received Leave (s)
91     R' ← {r ∈ R_e \ {s} : μ_r^e = IDLE ∧ λ_r^e = B_e}
92     if s ∈ F_e then
93         F_e ← F_e \ {s}
94     else // s ∈ R_e
95         R_e ← R_e \ {s}
96     end if
97     foreach r ∈ R' do
98         μ_r^e ← WAITING_PROBE
99         send upstream Update (r)
100    end foreach
101    send downstream Leave (s)
102 end when
103
104 end task
```

Fig. 4. Task router link (RL).

a previous *Probe* cycle has already finished (Line SN63), or when no *Probe* cycle is in process, i.e. $\mu_{s'}^e = \text{IDLE}$ (Lines SN35 and SN43). □

Claim 4. *After a B-Neck packet has been processed at link e, the set of sessions at link e is correct, i.e.* $S_e = R_e \cup F_e$.

Proof. Note that, when B-Neck is started, F_e and R_e are empty (Line RL3) at every router link e. Recall that, from Claim 3, every *Probe* cycle for a session is always completed. Thus, when a session s joins the network, it is included in set R_e for all $e \in \pi(s)$ at Line RL18 and Line SN14. Then, each time it is removed from R_e, it is included in F_e (Lines RL83, SN52 and SN70), unless it leaves the network (Line RL95 and Line SN25). Likewise, each time it is removed from F_e, it is included in R_e (Lines RL9 RL52, RL83 and SN25), unless it leaves the network (Lines RL93 and SN25). Since reliable communication channels are assumed, no B-Neck packet is lost. Hence, session counting is performed correctly. □

Claim 5. *After a B-Neck packet has been processed at link e, for all* $s \in F_e$, $\lambda_s^e < B_e$.

Proof. Note that, whenever a value of B_e is computed which might be smaller than its previous value, at Procedure *ProcessNewRestricted*, all the sessions $s \in F_e$ such that $\lambda_s^e \geq B_e$ are moved to R_e. Besides, when a session is moved from R_e to F_e (Line RL83), at Line RL77 it was tested that $\lambda_s^e < B_e$. Hence, after a packet has been completely processed at a link e, it is guaranteed that for all $s \in F_e$, $\lambda_s^e < B_e$. □

The following two claims resemble Claims 1 and 2 but applied to any instant of time when Distributed B-Neck is running, not necessarily when all the bottlenecks have been discovered by the

```
1  task SourceNode (s, e)
2  // e is the first link of s and it is
3  // not shared with any other session
4
5  procedure StartProbeCycle()
6      F_e ← ∅; R_e ← {s}
7      pending_probe_s ← FALSE
8      bneck_rcv_s ← FALSE
9      μ_s^e ← WAITING_RESPONSE
10     send downstream Probe (s, D_s, e)
11 end procedure
12
13 when API.Join(s, r) do
14     F_e ← ∅; R_e ← {s}
15     D_s ← min(r, C_e)
16     pending_probe_s ← FALSE
17     pending_leave_s ← FALSE
18     bneck_rcv_s ← FALSE
19     μ_s^e ← WAITING_RESPONSE
20     send downstream Join (s, D_s, e)
21 end when
22
23 when API.Leave(s) do
24     if μ_s^e = IDLE then
25         F_e ← ∅; R_e ← ∅
26         send downstream Leave (s)
27     else
28         pending_leave_s ← TRUE
29     end if
30 end when
31
32 when API.Change(s, r) do
33     D_s ← min(r, C_e)
34     if μ_s^e = IDLE then
35         StartProbeCycle()
36     else
37         pending_probe_s ← TRUE
38     end if
39 end when
40
41 when received Update (s) do
42     if μ_s^e = IDLE then
43         StartProbeCycle()
44     end if
45 end when
46
47 when received Bottleneck (s) do
48     if μ_s^e = IDLE ∧ ¬bneck_rcv_s then
49         bneck_rcv_s ← TRUE
50         API.Rate (s, λ_s^e)
51         if D_s > λ_s^e then
52             F_e ← {s}; R_e ← ∅
53         end if
54         send downstream SetBottleneck (s, D_s = λ_s^e)
55     end if
56 end when
57
58 when received Response (s, τ, λ, η) do
59     if pending_leave_s then
60         F_e ← ∅; R_e ← ∅
61         send downstream Leave (s)
62     else if τ = UPDATE ∨ pending_probe_s then
63         StartProbeCycle()
64     else if τ = BOTTLENECK then
65         λ_s^e ← λ
66         μ_s^e ← IDLE
67         bneck_rcv_s ← TRUE
68         API.Rate (s, λ_s^e)
69         if D_s > λ_s^e then
70             F_e ← {s}; R_e ← ∅
71         end if
72         send downstream SetBottleneck (s, D_s = λ_s^e)
73     else // τ = RESPONSE
74         λ_s^e ← λ
75         μ_s^e ← IDLE
76         if D_s = λ_s^e then // s is restricted at link e
77             bneck_rcv_s ← TRUE
78             API.Rate (s, λ_s^e)
79             send downstream SetBottleneck (s, TRUE)
80         end if
81     end if
82 end when
83
84 end task
```

Fig. 5. Task source node (SN).

```
1  task DestinationNode (s)
2
3  when received SetBottleneck (s, β) do
4      if ¬β then
5          send upstream Update (s)
6      end if
7  end when
8
9  when received Join (s, λ, η) do
10     send upstream Response (s, RESPONSE, λ, η)
11 end when
12
13 when received Probe (s, λ, η) do
14     send upstream Response (s, RESPONSE, λ, η)
15 end when
16
17 end task
```

Fig. 6. Task destination node (DN).

algorithm. This illustrates how Distributed B-Neck follows a similar logic to that of Centralized B-Neck.

Claim 6. *After a B-Neck packet has been processed at link e, $B_e \geq C_e/|S_e|$. If $F_e \neq \emptyset$, then $B_e > C_e/|S_e|$.*

Proof. Recall that $B_e = (C_e - \sum_{s\in F_e} \lambda_s^e)/|R_e|$. Then, $B_e|R_e| = C_e - \sum_{s\in F_e} \lambda_s^e$. If $F_e \neq \emptyset$, then, from Claim 5, for all $s \in F_e$, $\lambda_s^e < B_e$, so $\sum_{s\in F_e} \lambda_s^e < \sum_{s\in F_e} B_e = B_e|F_e|$. Hence, $B_e|R_e| > C_e - B_e|F_e|$. Thus, $B_e|R_e| + B_e|F_e| > C_e$. Since $|S_e| = |R_e| + |F_e|$, $B_e > C_e/|S_e|$. If F_e is empty, $\sum_{s\in F_e} \lambda_s^e = 0$. Thus, $C_e = B_e|R_e| = B_e|S_e|$ (recall that, if $F_e = \emptyset$, then $R_e = S_e$), so $B_e = C_e/|S_e|$. Thus, in any case, $B_e \geq C_e/|S_e|$. □

Claim 7. *Let e be a link, if $F_e \neq \emptyset$, then there is some session $s \in F_e$ such that $\lambda_s^e < C_e/|S_e|$.*

Proof. Recall that $B_e = (C_e - \sum_{s\in F_e} \lambda_s^e)/|R_e|$. By the way of contradiction, let us assume that for all $s \in F_e$, $\lambda_s^e \geq (C_e/|S_e|)$. Then, $\sum_{s\in F_e} \lambda_s^e \geq |F_e|(C_e/|S_e|)$. Hence, $B_e \leq (C_e - |F_e|(C_e/|S_e|))/(|S_e| - |F_e|) = ((C_e|S_e| - C_e|F_e|)/|S_e|)/(|S_e| - |F_e|) = (C_e/|S_e|)(|S_e| - |F_e|)/(|S_e| - |F_e|) = C_e/|S_e|$. However, from Claim 6, $B_e > C_e/|S_e|$, so we have reached a contradiction. □

The following claim shows that, when the estimated bandwidth of a session exceeds the estimated bottleneck rate of a link in its path, it will recompute it soon. If its state is not IDLE, then some packet of that session must be under process or transmission.

Claim 8. *After a B-Neck packet has been processed at link e, for all $s \in R_e$, $\lambda_s^e > B_e$ implies $\mu_s^e \neq$ IDLE.*

Proof. Note that the value of λ_s^e is set (to λ), only, at Line RL36, and μ_s^e is set to IDLE, only, at Line RL35, in both cases when

processing a *Response* packet, in which case $\lambda \leq B_e$. Besides, the only situation in which the value of B_e may be decreased is during a *Probe* cycle, more precisely when executing *ProcessNewRestricted*. In this case, if $\lambda_s^e > B_e$, tested at Line RL11, then μ_s^e is set to WAITING_PROBE at Line RL12. Hence, if $\lambda_s^e > B_e$, then $\mu_s^e \neq$ IDLE. □

Now we show that Distributed B-Neck follows Property 1 in its *Probe* cycles.

Claim 9. *When a Probe packet of a session s reaches the destination node,* $\lambda = \min_{e \in \pi(s)} B_e = B_\eta$ *for some* $\eta \in \pi(s)$ *(considering the values of* B_e *and* B_η *when the Probe packet was processed at each link e).*

Proof. Note that, at the source node, $\lambda = D_s$ and η is set to e when a *Probe* cycle is started (see Lines SN10 and SN20). Then, at each link $e \in \pi(s)$, if $\lambda > B_e$, then λ is set to B_e and η is set to e (Lines RL21–23 and RL55–57). □

The next two claims show that when the state of a session is set to WAITING_PROBE at some link in its path, then a *Probe* packet of that session will soon reach this link, so its estimated rate can be updated.

Claim 10. *Every time that* μ_s^e *is set to* WAITING_PROBE, *an Update packet or a Response packet with* $\tau =$ UPDATE *is sent upwards.*

Proof. This happens at Lines RL12–12, RL28–29, RL32–33, RL46, RL63–64, RL80–81, and RL98–99. □

Claim 11. *Update packets for a session s are relayed unless a previous Update packet (or a Response packet with* $\tau =$ UPDATE*) for session s is in its way to the source, or a Probe cycle for session s is already in course.*

Proof. At a link $e \in \pi(s)$, *Update* packets for session s are relayed at Line RL64 unless $\mu_s^e \neq$ IDLE, i.e. $\mu_s^e \in$ {WAITING_PROBE, WAITING_RESPONSE}. If $\mu_s^e =$ WAITING_PROBE, from Claim 10, an *Update* packet or a *Response* packet with $\tau =$ UPDATE was sent upwards, If $\mu_s^e =$ WAITING_RESPONSE, then a *Probe* cycle for session s is already in course. Note that μ_s^e is set to WAITING_RESPONSE when a *Join* or *Probe* packet is processed (Lines RL19, RL50, SN9 and SN19). Finally, note that when an *Update* packet or a *Response* packet with $\tau =$ UPDATE is received at the source node, a new *Probe* cycle is always started (Lines SN43 and SN63). Hence, if an *Update* packet is not relayed, another one (or a *Response* packet with $\tau =$ UPDATE) is in its way to the source, or a *Probe* cycle is already in course. □

Now we will formally define the concept of a session being *idle*, which will be useful in the following discussion to prove additional properties of Algorithm B-Neck.

Definition 5. A session s is *idle* if for all $e \in \pi(s)$, $\mu_s^e =$ IDLE.

Lemma 1. *If a session s is idle, then there is some* $\eta \in \pi(s)$ *such that for all* $e \in \pi(s)$, $\lambda_s^e = B_\eta = \min_{e' \in \pi(s)} B_{e'}$ *(considering the values of* $B_{e'}$ *and* B_η *when the Probe packet was processed at each link e).*

Proof. Recall that the state of the network is considered when no packet is being processed by any node. Note also that, in the case of a router link, the value of μ_s^e is set to IDLE, only, at Line RL35 when processing a *Response* packet. In this case, λ_s^e is also set to λ at Line RL36 (note that this is the only place where λ_s^e is modified). Besides, in the case of the source node, the value of μ_s^e is set to IDLE at Lines SN66 and SN75, in which cases, λ_s^e is also set to λ at Lines SN65 and SN74 (the only places where λ_s^e is modified). From Claim 9, when the corresponding *Probe* packet reached the destination node, $\lambda = \min_{e \in \pi(s)} B_e = B_\eta$ for some $\eta \in \pi(s)$. Besides, as it can be observed in the algorithm specification, during the processing of a *Response* packet, the value of λ is never changed. Furthermore, from Claim 3, every *Probe* cycle started for a session is always completed, and two *Probe* cycles of the same session never overlap. Hence, the *Response* packet is processed at each router link $e \in \pi(s)$ with the same value of λ and η. Therefore, if for all $e \in \pi(s)$, $\mu_s^e =$ IDLE, then there is some $\eta \in \pi(s)$ such that for all $e \in \pi(s)$, $\lambda_s^e = B_\eta = \min_{e' \in \pi(s)} B_{e'}$. □

Lemma 2. *If there is a session s such that, for some link* $e \in \pi(s)$, $\lambda_s^e < \min_{e' \in \pi(s)} B_{e'}$, *then s is not idle.*

Proof. By the way of contradiction, let us assume that session s is idle, and $\lambda_s^e < \min_{e' \in \pi(s)} B_{e'}$ for some link $e \in \pi(s)$. From Lemma 1, if s is idle, then there is some $\eta \in \pi(s)$ such that for all $e \in \pi(s)$, $\lambda_s^e = B_\eta = \min_{e' \in \pi(s)} B_{e'}$ (considering the values of $B_{e'}$ and B_η when the *Probe* packet was processed at each link e). Hence, the value of B_η must have been increased since it was computed during the *Probe* cycle that set the value of λ_s^e (recall that the value of λ_s^e is only changed during the processing of a *Response* packet). Consider now in which cases the value of B_η may be raised.

In the case η is the source node, the only case where D_s may be raised is at Line SN33, in which case, if $\mu_s^e =$ IDLE, then a new *Probe* cycle is started, which sets μ_s^e to WAITING_RESPONSE at Line SN9. Hence, s is not idle, what contradicts the initial assumption.

In the case η is a router link, recall that the value of B_η is computed as $B_\eta = (C_\eta - \sum_{s' \in F_\eta} \lambda_{s'}^\eta)/|R_\eta|$. If the value of B_η is increased, it must be because the value of $\sum_{s' \in F_\eta} \lambda_{s'}^\eta$ is increased, or because the value of $|R_\eta|$ is decreased. As it can be observed in the algorithm specification, the value of $\lambda_{s'}^\eta$ is not changed for any session in F_η. Hence, either a session x such that $\lambda_x^\eta < B_\eta$ has been moved from R_η to F_η, or a session has left the network. In the first case, this happens at Lines RL83–84. Thus, since $\lambda_s^\eta = B_\eta$ and s is idle, at Line RL80, μ_s^η must have been set to WAITING_PROBE. Hence, it is not idle, what contradicts the initial assumption. In the second case, since $\lambda_s^\eta = B_\eta$ and s is idle, at Line RL98, μ_s^η must have been set to WAITING_PROBE, what again contradicts the initial assumption. □

The following theorem and its corollary show that, if a session is idle, then its estimated rate must be correct under the current sessions configuration, what proves that Distributed B-Neck satisfies Property 1. Indirectly, this means that sessions do not generate control traffic when it is not necessary.

Theorem 1. *If there is a session s such that, for some link* $e \in \pi(s)$, $\lambda_s^e \neq \min_{e' \in \pi(s)} B_{e'}$, *then s is not idle.*

Proof. If session s is such that $\lambda_s^e \neq \min_{e' \in \pi(s)} B_{e'}$ for some link $e \in \pi(s)$, then either $\lambda_s^e < \min_{e' \in \pi(s)} B_{e'}$ for some link $e \in \pi(s)$, or there is some $e \in \pi(s)$ such that $\lambda_s^e > B_e$. In the first case, from Lemma 2, s is not idle. In the second case, $s \in R_e$ since otherwise ($s \in F_e$), from Claim 5, $\lambda_s^e < B_e$ what contradicts the initial assumption that $\lambda_s^e > B_e$. Hence, from Claim 8, s is not idle. □

Corollary 1. *If a session s is idle, then for all* $e \in \pi(s)$, $\lambda_s^e = \min_{e' \in \pi(s)} B_{e'}$.

The next two claims prove basic properties of B-Neck which help reducing control traffic, only relaying packets when necessary, and avoiding the sending of more than one *SetBottleneck* packet for the same estimated rate.

Claim 12. *Bottleneck packets for a session s are relayed upstream unless a SetBottleneck packet for that session has already been sent downstream or session s is not idle.*

Proof. *Bottleneck* packets are relayed at Line RL70, if $\mu_s^e =$ IDLE $\wedge$ $s \in R_e$. If $\mu_s^e \neq$ IDLE, then s is not idle. If $s \notin R_e$, then a *SetBottleneck*

packet for that session has already been sent, since s is moved to F only at Line RL83. □

Claim 13. *If a Bottleneck packet (or a Response packet with $\tau =$ BOTTLENECK) is received at the source, a SetBottleneck packet is sent unless the session is not idle or a previous SetBottleneck packet has already been sent with the current bandwidth assignment.*

Proof. In the case of a *Bottleneck* packet, if $\mu_s^e \neq$ IDLE (what implies that the session is not idle), no *SetBottleneck* packet is sent (see Line SN48). If $bneck\_rcv_s$ (Line SN48 again), then a previous *SetBottleneck* packet must have been sent. See that every time $bneck\_rcv_s$ is set to TRUE (Lines SN43 and SN77), a *SetBottleneck* packet is sent (Lines SN54 and SN79 respectively) and, when a *Probe* cycle is started, $bneck\_rcv_s$ is set to FALSE (Line SN8). In the case of the *Response* packet with $\tau =$ BOTTLENECK, it is not necessary to make these tests, because μ_s^e has just been set to IDLE (Line SN66) and no previous *Bottleneck* packet may have overtaken the *Response* packet being processed. □

Let us focus now on what happens at each core bottleneck e from $SSP(e)$ on.

To start with, we prove three lemmas which are essential to prove Theorem 2.

Lemma 3. *Let e be a core bottleneck. Then, for each $s \in S_e^*$, from $SSP(s)$ on, for all $e' \in \pi(s)$, $C_{e'}/|S_{e'}| \geq C_e/|S_e|$.*

Proof. By the way of contradiction, let us assume that there is some $s \in S_e^*$, such that, from $SSP(s)$ on, there is some link $e' \in \pi(s)$, such that $C_{e'}/|S_{e'}| < C_e/|S_e|$. From the definition of $SSP(s)$, from $SSP(s)$ on, $S_e = S_e^*$ and $S_{e'} = S_{e'}^*$. Then, $\sum_{s' \in S_{e'}} \lambda_{s'}^* \leq C_{e'}$. Besides, since e is a core bottleneck, $\lambda_s^* = B_e^* = C_e/|S_e| > C_{e'}/|S_{e'}|$. Hence, since $s \in S_{e'}$, there must be another session $s'' \in S_{e'}$ such that $\lambda_{s''}^* < C_{e'}/|S_{e'}|$ to compensate the sum. This implies that s depends on s'' and s'' does not depend on s, what is not possible since e is a core bottleneck and $s \in S_e^*$. □

Lemma 4. *Let e be a core bottleneck. Then, for each $s \in S_e^*$, from $SSP(s)$ on, for all $e' \in \pi(s)$, $B_{e'} \geq C_e/|S_e|$.*

Proof. From Claim 6, $B_{e'} \geq C_{e'}/|S_{e'}|$. Besides, from Lemma 3, $C_{e'}/|S_{e'}| \geq C_e/|S_e|$. Hence, $B_{e'} \geq C_e/|S_e|$. □

Corollary 2. *Let e be a core bottleneck. After $SSP(e)$, for all e' such that $S_e \cap S_{e'} \neq \emptyset$, $B_{e'} \geq C_e/|S_e|$.*

Lemma 5. *Let e be a core bottleneck. By time $SSP(e) + RTT$, F_e is permanently empty.*

Proof. By the way of contradiction, assume that F_e is not empty by time $SSP(e) + RTT$. Then, from Claim 7, there is some session $s \in F_e$ such that $\lambda_s^e < C_e/|S_e| = C_e/|S_e^*|$.

A session is moved to F_e, only, when a *SetBottleneck* packet is sent at the source node (Lines SN52 and SN70), or when a *SetBottleneck* packet is processed at a router link (Line RL83). Besides, a *SetBottleneck* packet is sent, only, after a *Response* packet with $\tau \neq$ UPDATE is received at the source node, that is, when μ_s^e is set to IDLE (Lines SN66 and SN75) which is a necessary condition for the sending of a *SetBottleneck* packet (see Lines SN48–55 and Lines SN73–81). In this case, at every router link in $\pi(s)$, μ_s^e is set to IDLE and λ_s^e is set to λ at Lines RL35–36 (otherwise τ is set to UPDATE at Line RL32). Recall also that, from Claim 9, when a *Probe* packet of a session s reaches the destination node, $\lambda = \min_{e \in \pi(s)} B_e = B_\eta$ for some $\eta \in \pi(s)$ (considering the values of B_e and B_η when the *Probe* packet was processed at each link e). Thus, $\lambda_s^\eta < C_e/|S_e^*|$ and μ_s^η was set to IDLE when the *Response* packet was processed at link η.

However, from Lemma 3, from $SSP(s)$ on, $C_\eta/|S_\eta| \geq C_e/|S_e| = C_e/|S_e^*|$. Besides, from Claim 6, $B_\eta \geq C_\eta/|S_\eta|$. Hence, by time $SSP(s)$, $\lambda_s^\eta < B_\eta$, while they were the same when λ_s^η was set. Hence, B_η was increased afterwards, but not later than $SSP(s)$. Note now that B_η could only be increased when a *SetBottleneck* packet or a *Leave* packet was processed. In the first case, as a consequence of a session with an assigned bandwidth which is smaller than B_η being moved from R_η to F_η, in which case all the sessions $r \in R_\eta$ such that $\mu_r^e =$ IDLE $\wedge\ \lambda_r^e = B_e\}$ (s among them) were sent an *Update* packet (see Lines RL77–84). In the second case, a session leaving the network increased B_η and again all the sessions $r \in R_\eta$ such that $\mu_r^e =$ IDLE $\wedge\ \lambda_r^e = B_e$ (s among them) were sent an *Update* packet (see Lines RL91–100). Recall that these *Update* packets are sent by time $SSP(s)$.

From Claim 11, *Update* packets for a session s are relayed unless a previous *Update* packet (or a *Response* packet with $\tau =$ UPDATE) for session s is in its way to the source, or a *Probe* cycle for session s is already in course. Since session s was idle, the *Update* packet was relayed, so it reached the source node by at most $SSP(s) + RTT(s)/2$.

When an *Update* packet is received at the source node, if the session is idle (which is the case of s as previously stated), a new *Probe* cycle is started (see Lines SN41–45). Thus, by time $SSP(s) + RTT(s)$, the *Probe* packet was processed at link e, so s was moved to R_e (see Lines RL51–54) by time $SSP(s) + RTT(s)$, what contradicts the initial assumption. Hence, F_e must be empty by time $SSP(s) + RTT$. □

Based on the concept of dependency, we will define the concept of stability of sessions, stability of links and stability of the network. To do so, we first define the concept of quiescence formally. Recall that we consider that the processing of the packets in the network is atomic, and we only consider the state of the network when no packet is being processed.

Definition 6. A session s is *quiescent* if there is no B-Neck packet of session s in the network. A link e is *quiescent* if for all $s \in S_e$, s is quiescent. The network is *quiescent* if for all $e \in E$, e is quiescent, i.e. there is no B-Neck packet in the network.

Now we define the concept of stability. A session s is stable if it is quiescent, all the sessions that affect s are also stable, its estimated rate corresponds to its max-min fair rate at all the nodes in its path and, if s is restricted at a link e, then s is in R_e and otherwise it is in F_e. A link e is stable if all the sessions whose paths traverse link e are stable. Finally, the network is stable if all its links are stable. More formally:

Definition 7. A session s is *stable* if (1) all the sessions that affect s are stable, (2) s is quiescent, (3) for all $e \in \pi(s)$, $\lambda_s^e = \lambda_s^*$, and (4) for all $e \in \pi(s)$, if $\lambda_s^e = B_e^*$, then $s \in R_e$, and $s \in F_e$ otherwise. A link e is *stable* if for all $s \in S_e^*$, s is stable. The network is *stable up to level* n if every link e such that $BL(e) \leq n$ is stable. The network is *stable* if all its links are stable. If a session/link/network is not stable, it is said to be *unstable*.

Now we prove that *Update* packets for each session r can only be generated when processing a *SetBottleneck* packet for a session with a smaller estimated bandwidth, or when processing a *Probe* packet for a session whose estimated bandwidth exceeds the estimated bottleneck rate of a link in its path. This property of B-Neck is needed to prove Theorem 2.

Claim 14. *After $SSP(r)$, Update packets for a session r can only be generated at a link e when (1) $r \in R_e$ and, when processing a SetBottleneck packet for a session $s \in S_e$, $\lambda_s^e < \lambda_r^e$, or (2) when processing a Probe packet for a session $s \in S_e$, $\lambda_r^e > B_e$.*

Proof. After $SSP(r)$, for all $e \in \pi(r)$, $S_e = S_e^*$ permanently. Thus *Update* packets can only be generated at Lines RL13 (when processing *SetBottleneck* packet) and RL81 (when processing a *Probe* packet).

In the first the case (that of Line RL81), only sessions $r \in R'$ are sent an *Update* packet, and these sessions have $\lambda_r^e = B_e$ (tested at Line RL78), while $\lambda_s^e < B_e$ (tested at Line RL77). In the case (that of Line RL13), only sessions r such that $\lambda_r^e > B_e$ will be sent an *Update* packet (tested at Line RL11). □

The following theorem proves that every core bottleneck e stabilizes by time $SSP(e) + 4RTT$. It will be used as the base case in the proof of the main result of this section.

Theorem 2. *Let e be a core bottleneck. By time $SSP(e) + 4RTT$, e is permanently stable.*

Proof. From Lemma 5, by time $SSP(e) + RTT$, F_e is permanently empty. Hence, $B_e = C_e/|S_e|$. Besides, from Corollary 2 after $SSP(e)$, for all e' such that $S_e \cap S_{e'} \neq \emptyset$, $B_{e'} \geq C_e/|S_e|$. Recall also that, from Claim 9, when a *Probe* packet of a session s reaches the destination node, $\lambda = \min_{e' \in \pi(s)} B_{e'} = B_\eta$ for some $\eta \in \pi(s)$ (considering the values of $B_{e'}$ and B_η when the *Probe* packet was processed at each link e'). Hence, every *Probe* cycle started for a session $s \in S_e$, after time $SSP(e) + RTT$, will end with $\lambda = \min_{e' \in \pi(s)} B_{e'} = C_e/|S_e| = B_e^*$. Recall that, if e is a core bottleneck, then for all $s \in S_e$, $\lambda^* = B_e^*$.

Consider the different states in which a session $s \in S_e$ might be at time $SSP(e) + RTT$:

- If it is idle, then, from Corollary 1, for all $e' \in \pi(s)$, $\lambda_s^{e'} = \min_{e'' \in \pi(s)} B_{e''} = C_e/|S_e|$.
- If an *Update* packet is in its way to the source node, then a *Probe* cycle will be started by time $SSP(e) + (3/2)RTT$. Recall that, from Claim 11, *Update* packets are relayed unless a previous *Update* packet (or a *Response* packet with $\tau =$ UPDATE) for session s is in its way to the source, or a *Probe* cycle for session s is already in course. This *Probe* cycle will end with $\lambda = C_e/|S_e|$, as previously stated, by time $SSP(e) + (5/2)RTT$.
- If a *Probe* packet is in its way to the destination node, which has not been processed at link e yet, then it will end with $\lambda = C_e/|S_e|$. Recall that, from Claim 3, every *Probe* cycle is always completed, and two *Probe* cycles of the same session never overlap. Besides, since the *Probe* packet will be processed at link e not before time $SSP(e) + RTT$, then, at link e, λ will be set to $C_e/|S_e|$ (see Lines RL55–57). Recall that, from Lemma 5, F_e is permanently empty by time $SSP(e) + RTT$. Thus this *Probe* cycle ends by time $SSP(e) + 2RTT$ with $\lambda = C_e/|S_e|$.
- If a *Probe* packet is in its way to the destination node, which has already been processed at link e, then the corresponding *Response* packet will be processed at link e after $SSP(e) + RTT$. Hence, either τ will be set to UPDATE if $\lambda > C_e/|S_e|$ (see Lines RL31–33), or $\lambda = C_e/|S_e|$. Recall again that, from Lemma 5, F_e is permanently empty after time $SSP(e) + RTT$. Thus, either the current *Probe* cycle ends with $\lambda = C_e/|S_e|$ by time $SSP(e) + 2RTT$, or a new *Probe* cycle is started immediately after the current one ends. Recall that, from Claim 3, every *Probe* cycle is always completed, and upon reception of a *Response* packet with $\tau = \lambda$, a new *Probe* cycle is started (see Lines SN62–63). Hence, this *Probe* cycle ends by time $SSP(e) + 3RTT$ with $\lambda = C_e/|S_e|$.
- If a *Response* packet is in its way to the source node and it has not been processed at link e yet, the case is analogous to the previous one with the only difference that the *Probe* cycles start (and end) $(1/2)RTT$ earlier. Hence, the last *Probe* cycle ends by time $SSP(e) + (5/2)RTT$ with $\lambda = C_e/|S_e|$.
- If a *Response* packet is in its way to the source node and it has been already processed at link e, then, if $B_e = C_e/|S_e|$ when the *Response* packet was processed at link e, then the case is analogous to the previous one. Otherwise ($B_e > C_e/|S_e|$ when the *Response* packet was processed at link e), by time $SSP(e) + RTT$, F_e becomes empty. In this case, an *Update* packet is sent to the source, unless a previous one has crossed link e after the *Response* packet was processed, but before F_e became empty. Note that, after $SSP(e)$, $S_e = S_e^*$, so no session may have left the network in that time interval. Hence, a session must have been moved from F_e to R_e. This is done only when a *Probe* packet is processed (Line RL52), in which case procedure *ProcessNewRestricted*() is executed (Line RL53). Then, an *Update* packet is sent to all sessions that have $\lambda_s^e > B_e$ and $\mu_s^e =$ IDLE (see Lines RL11–14). Recall also that, by time $SSP(e) + RTT$, $B_e = C_e/|S_e|$. Hence, an *Update* packet is sent to session s by time $SSP(e) + RTT$ following the *Response* packet, which is the second case considered.

Thus, we conclude that every session in S_e^* completes a *Probe* cycle with $\lambda = C_e/|S_e|$ and $\tau \neq$ UPDATE, by time $SSP(e) + 3RTT$. Consider the last such session whose *Response* packet was processed at link e. Since *Probe* cycles are always completed, after that *Response* packet was processed, for all other session $s' \in S_e$, $\lambda_{s'}^e = C_e/|S_e|$ and $\mu_{s'}^e =$ IDLE. Hence, the condition of Line RL38 holds. Thus, τ is set to BOTTLENECK and a *Bottleneck* packet is sent to each other session $s' \in S_e$ (see Lines RL38–44). These *Bottleneck* packets reach their source nodes by time $SSP(e) + (7/2)RTT$. Recall that, from Claim 12, *Bottleneck* packets are always relayed upstream unless a *SetBottleneck* packet for that session has already been sent downstream or the corresponding session not idle.

When a *Response* packet for session s is processed at its source node f, from Claim 13, a *SetBottleneck* packet is sent downstream unless a *SetBottleneck* packet has already been sent or $\mu_s^f \neq$ IDLE. Again from Claim 13, when a *Response* packet with $\tau =$ BOTTLENECK is received at the source node, a *SetBottleneck* packet is sent downstream. Thus, by time $SSP(e) + 4RTT$, all these *SetBottleneck* packets have reached their destination nodes and have been discarded. Note that *SetBottleneck* packets are always relayed unless the session is not idle (see Lines RL75–87). When the *SetBottleneck* packet is processed at a link e', where for all session $r \in R_{e'}$, $\mu_r^{e'} =$ IDLE and $\lambda_r^{e'} = B_{e'}$, then β is set to TRUE (see Lines RL75–76). This happens at link e, as previously stated, so when these *SetBottleneck* packet reach their destination nodes, they do with $\beta =$ TRUE. Thus no subsequent *Update* packet is generated at the destination nodes (see Lines DN4–5). Furthermore, when the *SetBottleneck* packet is processed at a link e', where $\lambda_r^{e'} < B_{e'}$, then the session is moved to $F_{e'}$, while it is kept in $R_{e'}$ otherwise (see Lines RL75–87).

Finally, recall that, from Claim 14, after $SSP(e)$, the sessions $s \in S_e$ can only be sent an *Update* packet at some link $e' \in \pi(s)$, when (1) $s \in R_{e'}$ and a *SetBottleneck* packet is processed for a session $s' \in S_{e'}$ with $\lambda_{s'}^{e'} < \lambda_s^{e'}$, or (2) processing a *Probe* packet for a session $s' \in S_{e'}$, $\lambda_s^{e'} > B_{e'}$. However, once a session $s \in S_e$ has completed the *Probe* cycle described, their assigned rate is $S_e/|S_e|$ and, from Corollary 2, after $SSP(e)$, for all e' such that $S_e \cap S_{e'} \neq \emptyset$, $B_{e'} \geq C_e/|S_e|$. Hence, the previous condition does not hold, so these sessions will never again be sent an *Update* packet. Thus, they will remain permanently stable by time $SSP(e) + 4RTT$. □

Let e be a bottleneck. All the sessions in R_e^* depend on each other since they have the same max-min fair rate and they share link e. Hence, all of them become stable at the same time. Similarly, all the links that depend among each other have the same bottleneck rate (and bottleneck level) and become stable at the same time.

Lemma 6. *After SSP, if the network is stable up to level n, then for all bottleneck e with $BL(e) = n + 1$, for each link e' which depends on e, $B_{e'} \geq B_e^*$.*

Proof. Let $S^n = \bigcup_{e'':BL(e'') \leq n} R_{e''}^*$. Let $G = F_{e'}^* \cap S^n$, $H = F_{e'}^* \setminus G$, and $P = F_{e'} \setminus G$. Note that, since the network is stable up to n, $G \subseteq F_{e'}$. Hence, $G \cap H = \emptyset$, $G \cap P = \emptyset$, $F_{e'}^* = G \cup H$ and $F_{e'} = G \cup P$. By defini-

tion,

$$B_{e'} = \frac{C_{e'} - \sum_{s\in F_{e'}} \lambda_s^{e'}}{|R_{e'}|} = \frac{C_{e'} - \sum_{s\in G} \lambda_s^{e'} - \sum_{s\in P} \lambda_s^{e'}}{|R_{e'}|}$$

Since the network is stable up to n and $G \subseteq S^n$, all the sessions in G are stable. Hence, for all $s \in G$, $\lambda_s^{e'} = \lambda_s^*$. Thus,

$$B_{e'} = \frac{C_{e'} - \sum_{s\in G} \lambda_s^* - \sum_{s\in P} \lambda_s^{e'}}{|R_{e'}|}$$

From Claim 4, the set of sessions at link e is correct, i.e. $S_{e'} = R_{e'} \cup F_{e'}$. Hence, after *SSP*, $S_{e'} = R_{e'} \cup F_{e'} = S_{e'}^* = R_{e'}^* \cup F_{e'}^*$. Besides, $R_{e'} \cup F_{e'} = R_{e'} \cup G \cup P$ and $R_{e'}^* \cup F_{e'}^* = R_{e'}^* \cup G \cup H$. Thus, $R_{e'} \cup G \cup P = R_{e'}^* \cup G \cup H$, so $R_{e'} \cup P = R_{e'}^* \cup H$. Then, $R_{e'} = (R_{e'}^* \cup H) \setminus P$. Since $H \subseteq F_{e'}^*$ and $F_{e'}^* \cap R_{e'}^* = \emptyset$, $|(R_{e'}^* \cup H)| = |R_{e'}^*| + |H|$. Thus, since $P \subseteq (R_{e'}^* \cup H)$, $|R_{e'}| = |R_{e'}^*| + |H| - |P|$. Hence,

$$B_{e'} = \frac{C_{e'} - \sum_{s\in G} \lambda_s^* - \sum_{s\in P} \lambda_s^{e'}}{|R_{e'}^*| + |H| - |P|}$$

Thus, $B_{e'}(|R_{e'}^*| + |H|) - B_{e'}|P| = C_{e'} - \sum_{s\in G} \lambda_s^* - \sum_{s\in P} \lambda_s^{e'}$. From Claim 5, for all $s \in P \subseteq F_{e'}$, $\lambda_s^{e'} < B_{e'}$. Besides, $|P| \geq 0$. Hence, $B_{e'}|P| \geq \sum_{s\in P} \lambda_s^{e'}$. Then, $B_{e'}(|R_{e'}^*| + |H|) \geq C_{e'} - \sum_{s\in G} \lambda_s^*$. Thus,

$$B_{e'} \geq \frac{C_{e'} - \sum_{s\in G} \lambda_s^*}{|R_{e'}^*| + |H|}$$

Let us focus now on $B_{e'}^*$. By definition,

$$B_{e'}^* = \frac{C_{e'} - \sum_{s\in F_{e'}^*} \lambda_s^*}{|R_{e'}^*|} = \frac{C_{e'} - \sum_{s\in G} \lambda_s^* - \sum_{s\in H} \lambda_s^*}{|R_{e'}^*|}$$

Thus, $B_{e'}^*|R_{e'}^*| = C_{e'} - \sum_{s\in G} \lambda_s^* - \sum_{s\in H} \lambda_s^*$. Then, $B_{e'}^*|R_{e'}^*| + \sum_{s\in H} \lambda_s^* = C_{e'} - \sum_{s\in G} \lambda_s^*$. Since e' depends on e, from Observation 2, $B_{e'}^* \geq B_e^*$. Hence, $B_e^*|R_{e'}^*| + \sum_{s\in H} \lambda_s^* \leq C_{e'} - \sum_{s\in G} \lambda_s^*$. Since $H \cap S^n = \emptyset$, the sessions in H are restricted at bottlenecks with bottleneck levels bigger or equal than that of e. Then, $\lambda_s^* \geq B_e^*$ for all $s \in H$. Otherwise, there would be a link e'' on which e depends but which does not depend on e, so $BL(e'')$ would be smaller than $BL(e)$, what is not possible. Hence, $B_e^*|R_{e'}^*| + B_e^*|H| \leq C_{e'} - \sum_{s\in G} \lambda_s^*$. Thus,

$$B_e^* \leq \frac{C_{e'} - \sum_{s\in G} \lambda_s^*}{|R_{e'}^*| + |H|} \leq B_{e'}$$

□

Lemma 7. *After SSP, if the network is stable up to level n, then no link e such that $BL(e) \leq n$ can be destabilized while the network remains in a steady state.*

Proof. Since e is stable, every session $s \in S_e^*$ is stable and it will not start a *Probe* cycle unless it receives an *Update* packet. However, from Claim 14, after *SSP*(s), *Update* packets for a session s can only be generated at a link e' if (1) $s \in R_{e'}$ and, when processing a *SetBottleneck* packet for a session $r \in S_{e'}$, $\lambda_r^{e'} < \lambda_s^{e'}$, or (2) when processing a *Probe* packet for a session $r \in S_{e'}$, $\lambda_s^{e'} > B_{e'}$.

Consider any such session $s \in S_e^*$. Since s is stable, from the definition of stability, for all $e' \in \pi(s)$, if $\lambda_s^{e'} = B_{e'}^*$, then $s \in R_{e'}$, and $s \in F_{e'}$ otherwise ($\lambda_s^{e'} < B_{e'}^*$). Thus, in case $s \in R_{e'}$, then all the sessions in $S_{e'}$ must also be stable, since s depends on all of them. Hence they cannot have B-Neck traffic, so they cannot destabilize session s. In case $s \in F_{e'}$, then only the move of a session r from $F_{e'}$ to $R_{e'}$ may cause the sending of an *Update* packet to session s (see Lines RL51–54), and this happens only if $\lambda_s^{e'} > B_{e'}$.

Note that stable bottlenecks have no B-Neck traffic. Hence, e' must be unstable and, since it shares session s with e, e' depends on e but e does not depend on e'. Hence, $BL(e) \leq n < BL(e')$. Hence, from Observation 2, $B_{e'}^* > B_e^*$. Besides, from Lemma 6, for all link e'' which depends on e', $B_{e''} \geq B_{e'}^*$. In particular, $B_{e'} \geq B_{e'}^*$. However, $\lambda_s^{e'} = \lambda_s^* \leq B_e^* < B_{e'}^* \leq B_{e'}$. Hence, the condition of Line RL11 ($\lambda_s^{e'} > B_{e'}$) does not hold, so no *Update* packet may be sent to session s in this case either. □

Claim 15. *After SSP, consider a network which is stable up to level n. Let e be a bottleneck such that $BL(e) = n + 1$ and $F_e \neq F_e^*$. Then there is a session $s \in G = F_e \setminus F_e^*$ such that $\lambda_s^e < B_e^*$.*

Proof. By definition, $B_e = (C_e - \sum_{s\in F_e} \lambda_s^e)/|R_e|$ and $B_e^* = (C_e - \sum_{s\in F_e^*} \lambda_s^*)/|R_e^*|$. From Claim 4, $S_e = R_e \cup F_e$. Besides, after *SSP*, $S_e = S_e^* = R_e^* \cup F_e^*$. Hence, $R_e = R_e^* \setminus G$, $G \subseteq R_e^*$ and $G \neq \emptyset$. Recall that, from the definition of stability, all the stable sessions $s \in S_e$ are in F_e and have $\lambda_s^e = \lambda_s^*$. Thus,

$$B_e = \frac{C_e - \sum_{s\in F_e} \lambda_s^e}{|R_e|} = \frac{C_e - \sum_{s\in F_e^*} \lambda_s^* - \sum_{s\in G} \lambda_s^e}{|R_e^*| - |G|}$$

By the way of contradiction, assume that, for all $s \in G$, $\lambda_s^e \geq B_e^*$. Then,

$$B_e < \frac{C_e - \sum_{s\in F_e^*} \lambda_s^* - |G|B_e^*}{|R_e^*| - |G|}$$

$$B_e(|R_e^*| - |G|) < C_e - \sum_{s\in F_e^*} \lambda_s^* - |G|B_e^*$$

$$C_e - \sum_{s\in F_e^*} \lambda_s^* - |G|B_e^* = C_e - \sum_{s\in F_e^*} \lambda_s^* - |G|\frac{C_e - \sum_{s\in F_e^*} \lambda_s^*}{|R_e^*|}$$

$$B_e(|R_e^*| - |G|) < \frac{|R_e^*|(C_e - \sum_{s\in F_e^*} \lambda_s^*) - |G|(C_e - \sum_{s\in F_e^*} \lambda_s^*)}{|R_e^*|}$$

$$B_e(|R_e^*| - |G|) < \frac{(|R_e^*| - |G|)(C_e - \sum_{s\in F_e^*} \lambda_s^*)}{|R_e^*|}$$

$$B_e < \frac{C_e - \sum_{s\in F_e^*} \lambda_s^*}{|R_e^*|} = B_e^*$$

However, from Lemma 6, after *SSP*, if the network is stable up to level n, then for all bottleneck e with $BL(e) = n + 1$, for each link e' which depends on e, $B_{e'} \geq B_e^*$. In particular, $B_e \geq B_e^*$, so we have reached a contradiction. □

Lemma 8. *After SSP, if the network is stable up to level n by time t, then for all bottleneck e, such that $BL(e) = n + 1$, by time $t + RTT$, $F_e = F_e^*$.*

Proof. Recall that, from the definition of stability, if a session s is stable, then for all $e \in \pi(s)$ such that $\lambda_s^* \neq B_e^*$, $s \in F_e$. Hence, $F_e^* \subseteq F_e$. By the way of contradiction, assume that, by time $t + RTT$, $F_e \neq F_e^*$. Then, from Claim 15, there is a session $s \in G = F_e \setminus F_e^*$ such that $\lambda_s^e < B_e^*$. Besides, from Lemma 6, if the network is stable up to level n, then for all bottleneck e with $BL(e) = n + 1$, for each link e' which depends on e, $B_{e'} \geq B_e^*$. Hence, $\lambda_s^e < B_{e'}$ for all $e' \in \pi(s)$. Then, from Lemma 2, s could not remain idle when the network became stable up to level n. Hence, by time t, either a *Probe* cycle for session s was in progress, or an *Update* packet was in its way to the source node. Recall that, from Claim 11, *Update* packets for a session s are relayed unless a previous *Update* packet (or a *Response* packet with τ = UPDATE) for session s is in its way to the source, or a *Probe* cycle for session s is already in course. Thus, a *Probe* cycle was in course by time t or a *Probe* cycle was started by time $t + RTT/2$ (see Lines SN41–45). Thus, by time $t + RTT$, the *Probe* packet was processed at link e, so s was moved to R_e (see Lines RL51–54) by time $t + RTT(s)$, what contradicts the initial assumption. Hence, $F_e = F_e^*$ by time $t + RTT$. □

Next theorem uses previous results to prove that the bottlenecks of the network stabilize following the order determined by

their bottleneck level. This result will be used as the induction step in the proof of the main result of this section.

Theorem 3. *After SSP, if the network is stable up to level n by time t, then it will be stable up to level $n+1$ by time $t+4RTT$.*

Proof. Note first that, from Lemma 8, after *SSP*, if the network is stable up to level n by time t, then for all bottleneck e, such that $BL(e)=n+1$, by time $t+RTT$, $F_e=F_e^*$. Besides, for all $s\in F_e^*$, $\lambda_s^e=\lambda_s^*$. Hence, $B_e=(C_e-\sum_{s\in F_e}\lambda_s^e)/|R_e|=(C_e-\sum_{s\in F_e^*}\lambda_s^*)/|R_e^*|=B_e^*$. Recall also that, from Claim 9, when a *Probe* packet of a session s reaches the destination node, $\lambda=\min_{e'\in\pi(s)}B_{e'}=B_\eta$ for some $\eta\in\pi(s)$ (considering the values of $B_{e'}$ and B_η when the *Probe* packet was processed at each link e'). Hence, every *Probe* cycle started for a session $s\in S_e$, after time $t+RTT$, will end with $\lambda=\min_{e'\in\pi(s)}B_{e'}=B_e^*$.

Consider the different states in which a session $s\in R_e$ might be at time $t+RTT$:

- If it is idle, then, from Corollary 1, for all $e'\in\pi(s)$, $\lambda_s^{e'}=\min_{e''\in\pi(s)}B_{e''}=B_e^*$.
- If an *Update* packet is in its way to the source node, then a *Probe* cycle will start by time $t+(3/2)RTT$. Recall that, from Claim 11, *Update* packets are relayed unless a previous *Update* packet (or a *Response* packet with $\tau=$ UPDATE) for session s is in its way to the source, or a *Probe* cycle for session s is already in course. This *Probe* cycle will end with $\lambda=B_e^*$, as previously stated, by time $t+(5/2)RTT$.
- If a *Probe* packet is in its way to the destination node, which has not been processed at link e yet, then it will end with $\lambda=B_e^*$. Recall that, from Claim 3, every *Probe* cycle is always completed, and two *Probe* cycles of the same session never overlap. Besides, since the *Probe* packet will be processed at link e not before time $t+RTT$, then, at link e, λ will be set to B_e^* (see Lines RL55–57). Recall that, from Lemma 8, after *SSP*, if the network is stable up to level n by time t, then for all bottleneck e, such that $BL(e)=n+1$, by time $t+RTT$, $F_e=F_e^*$. Thus this *Probe* cycle ends by time $t+2RTT$ with $\lambda=B_e^*$.
- If a *Response* packet is in its way to the source node and it has not been processed at link e yet, the case is analogous to the previous one with the only difference that the *Probe* cycles start (and end) $(1/2)RTT$ earlier. Hence, the last *Probe* cycle ends by time $t+(5/2)RTT$ with $\lambda=B_e^*$.
- If a *Response* packet is in its way to the source node and it has been already processed at link e, then, if $B_e=B_e^*$ when the *Response* packet was processed at link e, then the case is analogous to the previous one. Otherwise ($B_e>B_e^*$ when the *Response* packet was processed at link e), by time $t+RTT$, $F_e=F_e^*$. In this case, an *Update* packet is sent to the source, unless a previous one has crossed link e after the *Response* packet was processed, but before F_e became equal to F_e^*. Note that, after *SSP*, $S_e=S_e^*$, so no session may have left the network in that time interval. Hence, a session must have been moved from F_e to R_e. This is done only when a *Probe* packet is processed (Line RL52), in which case procedure *ProcessNewRestricted*() is executed (Line RL53). Then, an *Update* packet is sent to all sessions that have $\lambda_s^e>B_e$ and $\mu_s^e=$ IDLE (see Lines RL11–14). Recall also that, by time $t+RTT$, $B_e=B_e^*$. Hence, an *Update* packet is sent to session s by time $t+RTT$ following the *Response* packet, which is the second case considered.

Thus, we conclude that every session in R_e^* completes a *Probe* cycle with $\lambda=B_e^*$ and $\tau\neq$ UPDATE, by time $t+3RTT$. Consider the last such session whose *Response* packet was processed at link e. When that *Response* packet was processed, for all other session $s'\in R_e$, $\lambda_{s'}^e=B_e^*$ and $\mu_{s'}^e=$ IDLE, since they completed their *Probe* cycles. Hence, the condition of Line RL38 holds. Thus, τ is set to BOTTLENECK and a *Bottleneck* packet is sent to each other session $s'\in R_e$ (see Lines RL38–44). These *Bottleneck* packets reach their source nodes by time $t+(7/2)RTT$. Recall that, from Claim 12, *Bottleneck* packets are relayed upstream unless a *SetBottleneck* packet for that session has already been sent downstream or the corresponding session not idle.

When a *Response* packet for session s is processed at its source node f, from Claim 13, a *SetBottleneck* packet is sent downstream unless a *SetBottleneck* packet has already been sent or $\mu_s^f\neq$ IDLE. Again from Claim 13, when a *Response* packet with $\tau=$ BOTTLENECK is received at the source node, a *SetBottleneck* packet is sent downstream. Thus, by time $t+4RTT$, all these *SetBottleneck* packets have reached their destination nodes and have been discarded. Note that *SetBottleneck* packets are always relayed unless the session is not idle (see Lines RL75–87). When the *SetBottleneck* packet is processed at a link e', where for all session $r\in R_{e'}$, $\mu_r^{e'}=$ IDLE and $\lambda_r^{e'}=B_{e'}$, then β is set to TRUE (see Lines RL75–76). This happens at link e, as previously stated, so when these *SetBottleneck* packet reach their destination nodes, they do with $\beta=$ TRUE. Thus no subsequent *Update* packet is generated at the destination nodes (see Lines DN4–5). Furthermore, when the *SetBottleneck* packet is processed at a link e', where $\lambda_r^{e'}<B_{e'}$, then the session is moved to $F_{e'}$, while it is kept in $R_{e'}$ otherwise (see Lines RL75–87). Hence, the network is stable up to level $n+1$ by time $t+4RTT$.

Finally, from Lemma 7, after *SSP*, if the network is stable up to level n, then no link e such that $BL(e)\leq n$ can be destabilized while the network remains in a steady state. Hence, no link with bottleneck level $n+1$ or less may be destabilized after time $t+4RTT$. □

Now we present the main result of this section. We prove, by induction on the bottleneck level of the network, that once the network comes to a steady state, by time $SSP+BL\cdot 4RTT$, the network becomes stable, what implies that B-Neck becomes quiescent.

Theorem 4. *The network stabilizes by time $SSP+BL\cdot 4RTT$.*

Proof. This is proved by induction on the bottleneck level of the network BL as follows:

- Base case: The network stabilizes up to level 1 by time $SSP+4RTT$. This was proved as Theorem 2.
- Induction hypothesis: The network is stable up to level n by time t.
- Induction step: If the network is stable up to level n by time t, then it is stable up to level $n+1$ by time $t+4RTT$. This was proved as Theorem 3.

Hence, The network stabilizes by time $SSP+BL\cdot 4RTT$. □

Thus we have proved that B-Neck becomes quiescent by time $SSP+BL\cdot 4RTT$, and it computes the max-min fair rates correctly.

5. Experimental evaluation

We coded the B-Neck algorithm in Java and run it on top of a discrete event simulator to obtain realistic evaluation results. The event simulator is a home made version of *Peersim* (Montresor & Jelasity (2009)) that is able to run B-Neck with thousands of routers and up to a million hosts and sessions. This *Peersim* adaptation allows (a) running B-Neck simulations with a large number of routers, hosts and sessions, (b) importing Internet-like topologies generated with the Georgia Tech *gt-itm* tool (Zegura, Calvert, & Bhattacharjee (1996)), and (c) modelling several network parameters, like transmission and propagation times in the network links, processing time in routers and limited size packet queues.

In the conducted simulations we use topologies that are similar to the Internet, generated with the *gt-itm* network generator, configured with a typical Internet transit-stub model (Zegura et al. (1996)). The bandwidths of the physical links connecting hosts and stub routers have been set to 100 Mbps, those connecting stub routers have been set to 1 Gbps, and those connecting transit routers have been set to 5 Gbps. We use networks of three different sizes, to span various cases: a *Small* network (with 110 routers), a *Medium* network (with 1100 routers) and a *Big* network (with 11,000 routers). The number of hosts in the network is limited to 600,000. The links of the networks have been assigned two different propagation delays, to evaluate B-Neck with two different typical scenarios. Firstly, the propagation delay has been set to 1 microsecond in every link, in what we call the *LAN scenario*. This scenario models a typical LAN network, where *Probe* cycles are completed nearly instantly and the interactions of *Probe* and *Response* packets with packets from other sessions are only produced when a large number of sessions are present in the network. Secondly, the propagation time in links between hosts and routers was set to 1 microsecond, while it was set uniformly at random in the range of 1 to 10 milliseconds for the other links, in what we call the *WAN scenario*. This second scenario has a resemblance with an actual Internet topology where the propagation times in the internal network links are in the range of a typical WAN link. In these networks, *Probe* cycles take longer to complete, and they potentially interact more with other sessions than in the LAN scenario.

The sessions used in the experiments are created by choosing source and destination nodes uniformly at random, and connecting them with a shortest path. These paths are statically computed in advance. We have also coded the Centralized B-Neck algorithm (Fig. 2). Then, we have used this algorithm to validate that (the distributed) B-Neck in fact always find the correct max-min fair rates.

We have run two different experiments. *Experiment 1* analyzes the convergence time of B-Neck, and the number of B-Neck packets exchanged until quiescence, with different network sizes and different numbers of sessions. *Experiment 2* compares the performance of B-Neck with several well-known max-min fair distributed protocols, which were designed to be used as part of congestion control mechanisms.

5.1. Experiment 1

In our first experiment (Experiment 1) we force multiple sessions to arrive simultaneously initially, and evaluate the performance of B-Neck until convergence. Specifically, we focus on the time B-Neck requires to converge (i.e. to become quiescent) and the amount of traffic it generates. The number of sessions varies from 100 to 300,000 sessions. We have each session joining at a time chosen uniformly at random in the first five milliseconds of the simulation. We explore all the six different configurations: Small, Medium and Big network sizes, with LAN and WAN scenarios.

From a theoretical point of view, recall that the time B-Neck requires to become quiescent after no session joins or leaves the network (i.e. when the Steadiness Starting Point is reached) is upper bounded by $BL \cdot 4RTT$. Note that *RTT* (the biggest round-trip time in the network) depends on the sessions configuration, the network topology, and the bandwidth and propagation time of the links, while *BL* (the bottleneck level of the network) does not depend on the bandwidth and propagation time of the links. Thus, for a certain network topology, LAN and WAN scenarios will have different (and possibly variable) *RTT*, while they will have the same *BL*.

Regarding *BL*, when the number of sessions is small compared to the number of links in the network, their routes will hardly share links. Hence, the bottleneck level *BL* will remain almost constant (and close to 1), until the number of sessions reaches a certain threshold which depends on the size and topology of the network. Once this threshold is exceeded, the dependencies among sessions start to increase with the number of sessions, what increases the bottleneck level of the network accordingly. This increment is upper bounded by the number of links in the network (e.g. a set of sessions in a network in which each link is a bottleneck and depends on another link). *BL* behavior can be observed in Fig. 7. In the Small topology (110 routers), *BL* is 1 when the number of sessions is up to 1,000. When the number of sessions exceeds 1,000, *BL* starts growing up to a value of 100 for 30,000 sessions. Finally, no significant increment is observed in *BL* when more sessions than 30,000 are considered because no more bottleneck dependencies can appear in this topology. The figure also shows that *BL* is 1 while up to 10,000 and 100,000 sessions are considered respectively in the Medium and Big topologies, and it starts growing when more sessions are considered.

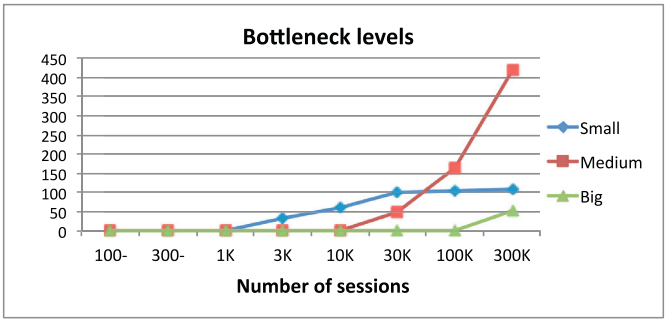

Fig. 7. Average bottleneck levels observed in Experiment 1 with different network topologies.

RTT behaviour in LAN and WAN scenarios is shown in Fig. 8. Recall that we define $RTT = \max_{s \in S} RTT(s)$ (i.e. the maximum value of session *RTT*s) in an execution. In order to show the central tendency of this value we do not plot the *RTT* but the average value of the *RTT*s obtained at 1 millisecond intervals during an execution. Session *RTT* is the time a packet takes to go from the source node of the session to its destination and back. We can approximate the *RTT* of session s as $2 \cdot \sum_{e \in \pi(s)} (T_q(e) + T_x(e) + T_p(e))$, where T_q is the time that a packet has to wait in the link queue before it is transmitted, T_x is the transmission time, and T_p is the propagation time. Both T_x and T_p are constant values for a link depending linearly on its transmission and propagation speeds respectively. However, T_q depends on the congestion level of the link. This value will be zero only if the queue is empty and no packet is being transmitted in the link, and so, we will observe increasing $T_q(e)$ and *RTT* values if the network link suffers increasing degrees of congestion. Recall that sessions are constantly performing *Probe* cycles before they become quiescent and a new *Probe* cycle is not started before the current one ends. In addition, a *Probe* cycle generates a *Probe* packet that is transmitted from the source node of the session to its destination, and a *Response* packet that is transmitted back. Since a *Probe* cycle is completed after one *RTT*, the smaller the session *RTT* is, the bigger the number of *Probe* cycles a session performs per second, and the bigger the number of packets (*Probe* and *Response*) injected per second into the network. When the number of sessions in the network is big enough, the total number of *Probe* cycles and injected packets start saturating network links and some of these packets have to wait in link queues generating increasing *RTT*s (because $T_q(e)$ increases). This situation can be observed in the Small topology when the LAN scenario is considered. Before reaching 10,000 sessions, the packets generated by the *Probe* cycles have not saturated any network link yet. When

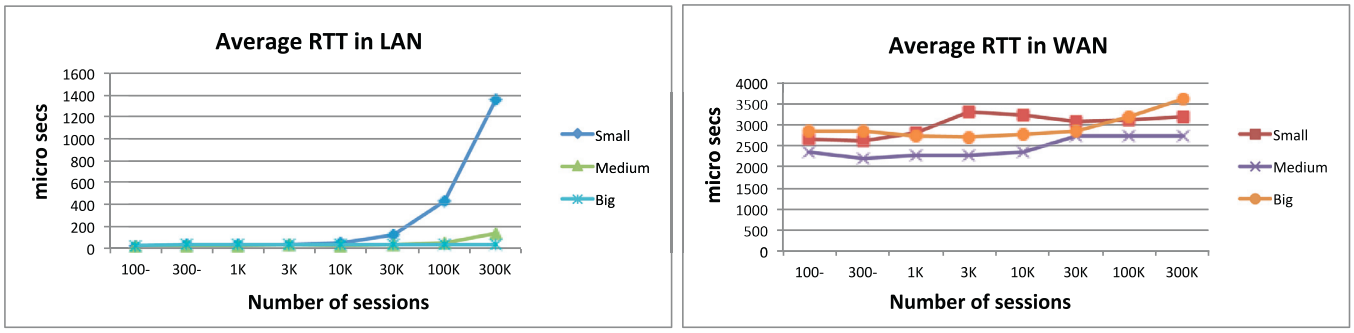

Fig. 8. Average *RTT*s observed in Experiment 1 with LAN (left) and WAN (right) topologies.

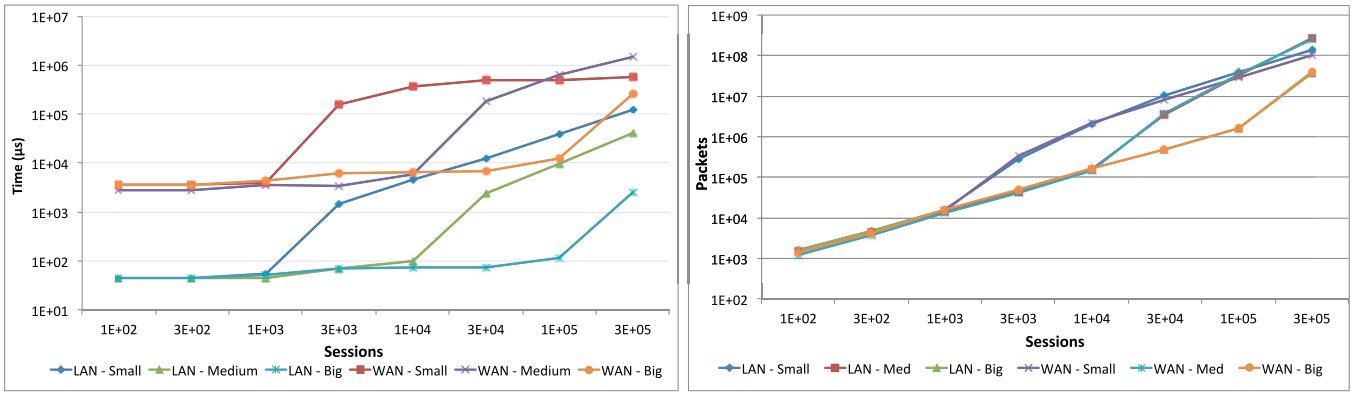

Fig. 9. (left) Time until quiescence (right) and number of packets observed in Experiment 1.

the number of sessions is greater than 10,000, an increasing number of packets have to wait in link queues generating bigger *RTT*s. In the Medium topology this behaviour does not appear until the number of sessions reaches 100,000. In the Big topology we do not observe this increment in the *RTT*s because 300,000 sessions are not enough to saturate the network links. Note that, as the *RTT* increases, the number of *Probe* cycles and injected packets decreases, and hence, the Small curve in LAN will not increment indefinitely and will tend to stabilize around a *RTT* balance point not shown in the figure.

A different situation appears when a WAN scenario is considered. In this scenario, even in the Small topology, the *RTT* curve is nearly a constant because the value that dominates session *RTT*s is not $T_q(e)$ but $T_p(e)$. In other words, WAN propagation times in links, which are in the range of milliseconds, determine a greater duration of *Probe* cycles, so a smaller number of *Probe* cycles is performed per time unit. In this context, the number of sessions used in our experiment produces a number of *Probe* cycles per second that does not inject enough packets to saturate network links. Therefore, $T_q(e)$ will be nearly negligible in all links, not affecting significantly *RTT* values. Slight *RTT* increases are observed when 1,000, 10,000 and 30,000 sessions are considered in Small, Medium and Big topologies respectively, but their balance point is only roughly a 30% greater than the rest of *RTT* values.

Regarding the convergence time of B-Neck, Fig. 9 (left) shows the time needed to reach quiescence in Experiment 1 (observe that both axes are in logarithmic scale). First, we focus in the case of the LAN scenario. As it can be observed in the plot, while the number of sessions does not exceed the previously commented threshold (1,000 for the Small, 10,000 for the Medium and 100,000 for the Big network), *BL* is 1 and the average *RTT* is around 30 microseconds. Therefore, the observed convergence time is almost constant and in the order of 1.5 *RTT*. Note that the theoretical convergence upper bound is $BL \cdot 4RTT$ and in this situation, since *BL* is 1, the upper bound is $4RTT$. When the number of sessions is greater than this threshold, the number of bottleneck levels starts increasing (Fig. 7), and so does the convergence time. Note that in the Small network, as previously commented, when 30,000 or more sessions join the network, *BL* stops increasing and remains constant, but *RTT* increments gradually when 10,000 sessions or more join the network due to the emergence of saturated links. Therefore, the convergence time being the product of both values continues increasing until the *RTT* balance point is reached (this point is not shown in the figure).

In the WAN scenario, as average *RTT* is roughly constant independently of the number of sessions, the convergence time increases linearly depending upon *BL*. Like in the LAN scenario, while the number of sessions does not exceed the previously described threshold (1,000 for the Small, 10,000 for the Medium and 100,000 for the Big network), *BL* is 1 and the observed convergence time is almost constant and in the order of 1.5 *RTT* (the average *RTT* is around 2500 microseconds). When the number of sessions is greater than the threshold, the number of bottleneck levels starts increasing (Fig. 7), and so does the convergence time depending linearly upon *BL*. Note that, in the Small network, *BL* remains constant when more than 30,000 sessions are considered, and so does the convergence time.

In Fig. 9 (right) we show the total number of packets transmitted in the network during each simulation. In this experiment, every packet sent across a link is accounted for, i.e. a *Probe* cycle of a session *s* generates a number of packets that is twice the length of the path of *s*. Regarding the theoretical convergence upper bound previously proved, the number of *Probe* cycles a session needs to stabilize and become quiescent depends upon *BL* (in the worst case, a session can require $4BL$ *Probe* cycles to stabilize). Therefore, the total the number of *Probe* cycles generated depend upon *BL* and the number of sessions. While the number of sessions does not exceed the threshold (1,000 for the Small, 10,000

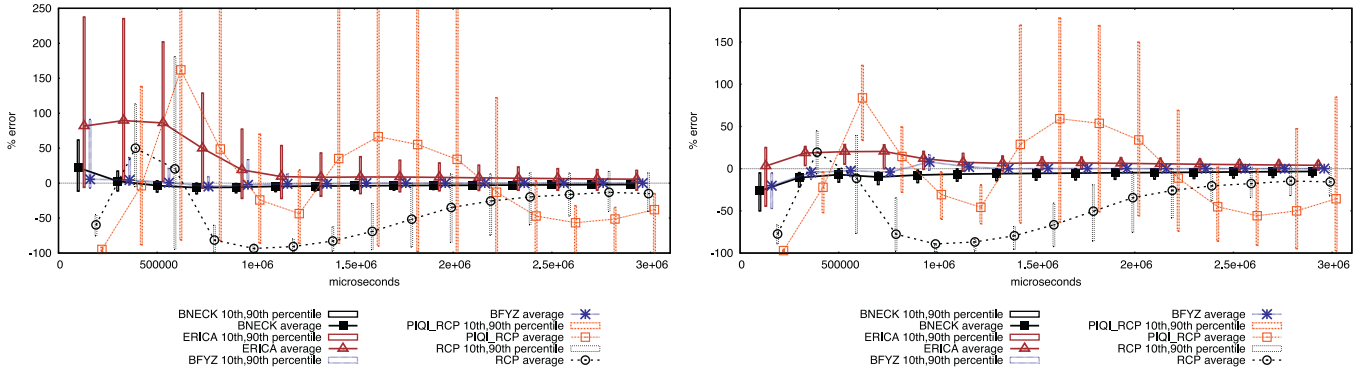

Fig. 10. Experiment 2a. Error distribution at sources (left) and bottlenecks (right).

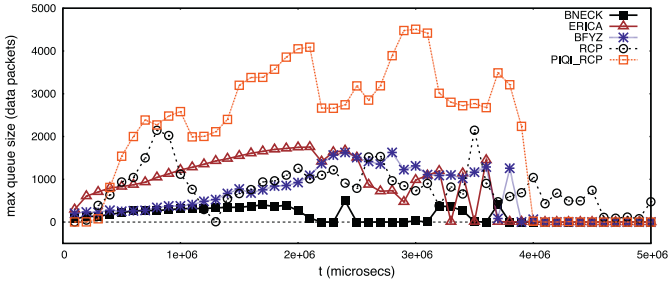

Fig. 11. Experiment 2b. Maximum queue sizes.

for the Medium and 100,000 for the Big network), *BL* is 1, and so, the number of packets transmitted and *Probe* cycles are linearly dependent only upon the number of sessions (1.5 *Probe* cycles per session on average). When the number of sessions is greater than this threshold, the number of packets transmitted and *Probe* cycles depends upon *BL* and the number of sessions, and so, an increase in the number of packets can be observed in the figure due to greater values of *BL*. When considering LAN and WAN scenarios with the same network topology, both curves overlap and the differences observed between them are nearly negligible. As previously commented, in the worst case, the total the number of *Probe* cycles depends upon *BL* and the number of sessions, being independent of WAN or LAN *RTT* values. Nevertheless, our simulations show limited differences when both scenarios are compared. As WAN *RTT* values are greater than LAN ones, more interactions can happen during a WAN *Probe* cycle, and so, from a practical perspective, a slightly smaller number of *Probe* cycles is required to stabilize each session on average.

We can conclude the analysis of this experiment observing that B-Neck behaves rather efficiently both in terms of time to quiescence, and in terms of average number of packets per session. Note that, when the number of sessions is small, dependency among them is also small and, then, the number of bottleneck levels is small, so a high degree of parallelism in the stabilization of the bottlenecks is achieved obtaining convergence times significantly smaller than the theoretical upper bound.

5.2. Experiment 2

In Experiment 2, we compare B-Neck with other well-known max-min fair distributed algorithms used to implement control congestion mechanisms. We selected RCP (Dukkipati et al. (2005)) and PIQI-RCP (Jain & Loguinov (2007)) as representatives of the closed-loop-control approach, and ERICA (Kalyanaraman et al. (2000)) and BYZF (Bartal et al. (2002)) because they use per-session state as B-Neck.

First, we set up a WAN scenario using a *Medium* network and run two experiment variations (Experiments 2a and 2b). We analyze the stability of each algorithm with respect to wrong rate assignments and specifically with those that are greater than the max-min fair ones as they are more likely to produce congestion problems. In addition, we study the stress induced on the link queues in order to identify potential link overshoot problems when high session churn appears in the network. Finally, in Experiment 2c, we set up a WAN scenario using a *Big* network to compare the B-Neck with the best representative of the other max-min fair algorithms, in order to show their transient behaviour when exposed to aggressive session churn patterns.

Experiment 2a. We analyze the accuracy of bandwidth assignments in the sources and at the links in order to compare the stability of B-Neck with these algorithms. We show that B-Neck, being as scalable as closed-loop-control proposals, exhibits a better transient behavior, and in the same range of the best per-session state proposals.

We measure the accuracy of bandwidth assignments at the sources, plotting the distribution of the relative error of the rate assignments at the sources with respect to their max-min fair values, obtained with a centralized max-min fair algorithm. This error reflects the accuracy of the algorithm experienced by the sessions at each point in time. Let $\lambda_s(t)$ be the rate assigned to session s at time t, and $\lambda_s^*(t)$ the max-min fair assignment given the set of sessions at time t, the error of session s at time t is computed as $E_s(t) = \frac{\lambda_s(t)-\lambda_s^*(t)}{\lambda_s^*(t)} \times 100$. In addition, we study the relative error of the rates assigned to the sessions that cross a link with respect to the link capacity in order to measure the accuracy of the bandwidth assignments at bottlenecks, This error allows to evaluate the overutilization (e.g, overshoots) in the bottlenecks, with respect to their maximum capacity, and so, it gives an idea of the stress that the bottleneck links are suffering. Let, $S_e(t)$ be the set of sessions that cross link e at time t, and C_e the bandwidth of link e. Then, the error of a bottleneck e at time t is $E_e(t) = \frac{\sum_{s\in S_e(t)} \lambda_s(t) - C_e}{C_e} \times 100$.

In this scenario 50,000 sessions are started during the first 500 milliseconds, then 10,000 sessions leave the network and 10,000 new sessions join it during the following 200 milliseconds. The parameters used in RCP experiments are $\alpha = 0.1$ and $\beta = 1$ as suggested in Dukkipati et al. (2005). In PIQI-RCP, the parameters used are $T = 5$ ms and the upper bound $\kappa^* = \frac{T}{2(T+RTT)}$ as suggested in Eq. 35 in Jain and Loguinov (2007). Fig. 10 shows the accuracy of the rate assignments for all the algorithms considered, at the sources and links. Percentile bars are used to show the deviation from the average. Perceptibly, B-Neck, BYZF and ERICA show the better accuracy, with small deviations from the average at sources

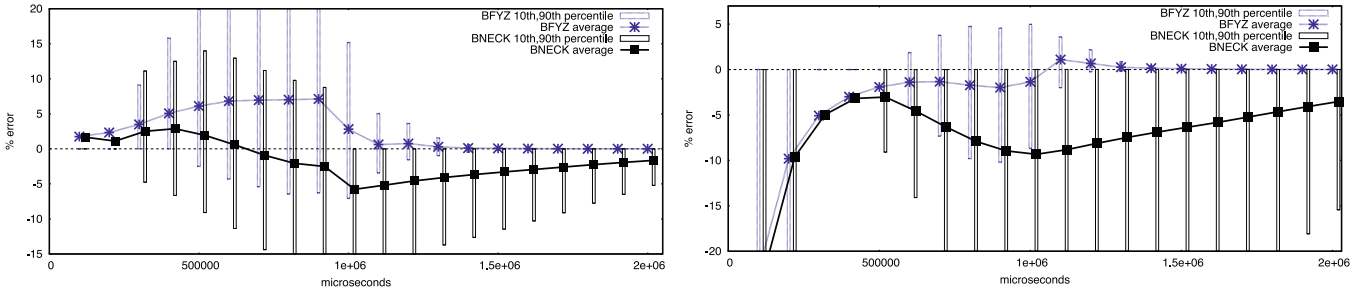

Fig. 12. Experiment 2c. Error distribution at (left) sources and (right) bottlenecks.

and links, although ERICA tends to make rate assignments a little bit over the exact rate.

Experiment 2b. During 500 milliseconds 50,000 sessions are started and they are assigned fixed amounts of data to be transmitted, following a Pareto distribution with $mean = 300$ packets of 2000 bytes each, and $shape = 4$. In Fig. 11 we show the evolution of the size of queues at the links of the network, to state the stress induced on the links. It can be observed that only B-Neck is able to keep the maximum size of the queues bounded during the whole experiment what is a clear indication that no congestion problems are created at links and hence to the network. *Experiment 2c*. We study the B-Neck transient behavior when it is exposed to aggressive session churn patterns. In this case we compare B-Neck only with BFYZ, since the previous experiments showed that BFYZ performance was the best among the rest of algorithms. In this experiment variation, 180,000 sessions are injected during the first second. The results of this experiment are shown in Fig. 12 where we can observe that B-Neck performs as well as BFYZ. Both algorithms quickly reach the max-min fair rates, what implies almost full utilization of the links but never overshooting them significantly. However, B-Neck is more conservative than BFYZ while BFYZ tends to slightly overload the network, assigning transient rates greater than the max-min fair rates. Hence B-Neck is more *network friendly* than BFYZ, but BFYZ converges slightly faster than B-Neck, although, in a practical sense, B-Neck reaches rates that are nearly the max-min fair rates in a similar time.

Finally, we plot the number of packets transmitted in each interval by B-Neck and BFYZ in this experiment variation (Fig. 13). It can be observed that B-Neck, in the worst case (when sessions have not converged yet to their max-min fair rate), always injects to the network a number of packets smaller than BFYZ. As soon as sessions converge to their max-min fair rates, B-Neck stops injecting packets to the network and the total traffic generated by B-Neck decreases dramatically. Eventually, no packet is injected when all the sessions have converged and B-Neck has reached quiescence. However, BFYZ keeps injecting the same number of packets even when convergence is reached (because BFYZ does not know that sessions are on the convergence point). In light of these results, we conclude that B-Neck offers, due to its quiescence, an efficient alternative to the rest of distributed max-min fair algorithms that need to transmit packets periodically into the network to recompute the max-min fair rates.

We conclude this section observing that B-Neck convergence to the max-min fair rates is realized (i) independently of the churn session pattern, with error distributions at sources and links that are nearly constant during transient periods, (ii) achieving a nearly full utilization of links but never overshooting them on average (the maximum of queue sizes is upper bounded by 3% of the link capacity) and (iii) more efficiently that the rest of distributed max-min fair algorithms.

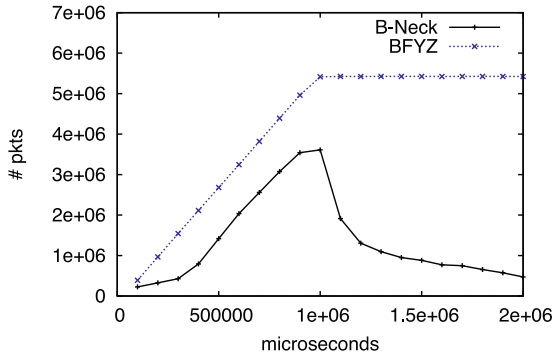

Fig. 13. Packets transmitted in Experiment 2c.

6. Conclusions and future work

In this paper we present B-Neck, a distributed algorithm that can be used as the basic building block for the new generation of proactive and anticipatory congestion control protocols.

B-Neck computes proactively the optimal sessions' rates independently of congestion signals applying a max-min fairness criterion. B-Neck iterates rapidly until converging to the optimal solution and is also quiescent. This means that, in absence of changes (i.e., session arrivals or departures), once the max-min rates have been computed, B-Neck stops generating network traffic (i.e., packets). When changes in the environment occur, B-Neck reacts and the affected sessions are asynchronously informed of their new rate (i.e., sessions do not need to poll the network for changes). As far as we know, B-Neck is the first max-min fair distributed algorithm that does not require a continuous injection of packets to compute the rates. This property can be advantageous in practical deployments in which energy efficiency needs to be considered.

From a theoretical perspective, the main contribution of this paper is that we formally prove the correctness and convergence of B-Neck. Only a few of the existing max-min fair optimization algorithms formally prove their correctness and convergence and, to the best of our knowledge, none of the reactive proposals does it. In addition, we obtained an upper bound for B-Neck convergence speed. After no session joins or leave the network, B-Neck becomes quiescent by time $BL \cdot 4RTT$, where BL is the number of bottleneck levels and RTT is the maximum round trip time in the network. This upper bound, based on bottleneck levels, is more accurate than the existing proposals that are based on the number of bottleneck links. (Note that several bottleneck links could share the same bottleneck level.) Considering these theoretical properties and being scalable, fully distributed and quiescent, we conclude that (a) B-Neck is a good candidate to be utilized as the basic building block for proactive congestion control mechanisms, and (b) due to its simple way of propagate local link information by

means of Probe cycles, the integration of a distributed predictive module that can provide anticipatory capabilities should be done seamlessly.

From an experimental point of view, the main contribution of this paper is an in-depth experimental analysis of the B-Neck algorithm. Extensive simulations were conducted to validate the theoretical results and compare B-Neck with the most representative competitors. Firstly, we made an experimental validation of the previously obtained theoretical upper bound for convergence time, which depends on the number of bottleneck levels (*BL*) and the round trip time (*RTT*). The main conclusion is that when the number of sessions is small, dependency among them is also small and so, the number of bottleneck levels (*BL*) is small. Therefore, a high degree of parallelism in the stabilization of the bottlenecks is achieved obtaining experimental convergence times significantly smaller than the theoretical upper bound. Additionally, we can conclude this analysis observing that (a) B-Neck behaves rather efficiently both in terms of time to quiescence, and in terms of average number of packets per session, (b) B-Neck convergence to the max-min fair rates is realized independently of the churn session pattern, with error distributions at sources and links that are nearly constant during transient periods, and achieving a nearly full utilization of links but never overshooting them on average, (c) as soon as sessions converge to their max-min fair rates, B-Neck stops injecting packets to the network and the total traffic generated by B-Neck decreases dramatically, and (d) B-Neck scalability has been experimentally demonstrated in Internet-like topologies with up to 11,000 routers and 180,000 of sessions.

Secondly, when compared with the main representatives of proactive and reactive approaches, (a) B-Neck convergence time is in the same range as that of the fastest (non-quiescent) distributed max-min fair algorithms (Bartal et al. (2002)) and faster that any of the reactive closed-control-loops proposals, some of which were observed to not converge many times, and (b) error distributions at sources and links are in the same range as the best competitors (Bartal et al. (2002)). These characteristics encourage the application of B-Neck to explicitly compute sending rates in a proactive congestion control protocol, being a more efficient alternative than the rest of current distributed max-min fair algorithms due to its quiescence.

In the near future, the Internet will have to meet a high demanding scenario in network bandwidth and speed. Therefore, effective congestion control mechanisms will have to anticipate decisions in order to avoid or at least mitigate congestion problems. In this context, we plan in the future to take additional steps to integrate predictive and forecasting capabilities in the B-Neck algorithm. Machine learning techniques, and in particular, prediction and forecasting models can be applied to proactive algorithms in order to obtain future estimations of the session's rates in nearly real-time. However, the selection of the right features to be input to the machine learning predictors is a compelling research question. Should we add multiresolution context information to the time series to be forecasted? In order to decrease the error for each prediction, do we need to include exogenous variables (e.g. network topology, expected patterns of sessions joining and leaving the network) to the time series representing the session rate assignment at each time interval?

Nowadays, one of the most promising lines of research in computer vision and speech recognition are deep neural networks. In these fields, deep networks are replacing and outperforming the manual feature extraction procedures based on hand-crafted filters or signal transforms. We plan to evaluate the accuracy of session's rate predictions training different deep models (e.g. recurrent or convolutional neural networks) against traditional predictors and time series forecasting techniques (e.g. ARIMA, GARCH) in the context of congestion avoidance. Given that traditional approaches for time series analysis and forecasting rely on a set of preprocessing techniques, such as identification of stationarity and seasonality, detrending, etc. the research question is: Can deep models replace these methods and successfully analyse raw time series data for forecasting congestion problems?

Acknowledgments

The research leading to these results has received funding from the Spanish Ministry of Economy and Competitiveness under the grant agreement TEC2014-55713-R, the Spanish Regional Government of Madrid (CM) under the grant agreement S2013/ICE-2894 (project Cloud4BigData), the National Science Foundation of China under the grant agreement n. 61520106005 and the European Union under the FP7 grant agreement n. 619633 (project ONTIC, http://ict-ontic.eu) and the H2020 grant agreement n. 671625 (project CogNet, http://www.cognet.5g-ppp.eu/).

Supplementary material

Supplementary material associated with this article can be found, in the online version, at 10.1016/j.eswa.2017.09.015.

References

Afek, Y., Mansour, Y., & Ostfeld, Z. (1999). Convergence complexity of optimistic rate-based flow-control algorithms. *Journal of Algorithms, 30*(1), 106–143.

Afek, Y., Mansour, Y., & Ostfeld, Z. (2000). Phantom: A simple and effective flow control scheme. *Computer Networks, 32*(3), 277–305.

Awerbuch, B., & Shavitt, Y. (1998). Converging to approximated max-min flow fairness in logarithmic time. In *INFOCOM* (pp. 1350–1357).

Bartal, Y., Farach-Colton, M., Yooseph, S., & Zhang, L. (2002). Fast, fair and frugal bandwidth allocation in ATM networks. *Algorithmica, 33*(3), 272–286.

Bertsekas, D., & Gallager, R. G. (1992). *Data networks (2nd edition)*. Prentice Hall.

Bui, N., Cesana, M., Hosseini, S. A., Liao, Q., Malanchini, I., & Widmer, J. (2017). A survey of anticipatory mobile networking: context-based classification, prediction methodologies, and optimization techniques. *IEEE Communications Surveys & Tutorials, PP*(99).

Cao, Z., & Zegura, E. W. (1999). Utility max-min: An application-oriented bandwidth allocation scheme. In *INFOCOM* (pp. 793–801).

Charny, A., Clark, D., & Jain, R. (1995). Congestion control with explicit rate indication. In *International conference on communications, ICC'95, vol. 3* (pp. 1954–1963).

Chen, C.-K., Kuo, H.-H., Yan, J.-J., & Liao, T.-L. (2009). Ga-based PID active queue management control design for a class of TCP communication networks. *Expert Systems with Applications, 36*(2), 1903–1913.

Cobb, J. A., & Gouda, M. G. (2008). Stabilization of max-min fair networks without per-flow state. In S. S. Kulkarni, & A. Schiper (Eds.), *SSS*. In *Lecture Notes in Computer Science: 5340* (pp. 156–172). Springer.

Dukkipati, N., Kobayashi, M., Zhang-Shen, R., & McKeown, N. (2005). Processor sharing flows in the internet. In H. de Meer, & N. T. Bhatti (Eds.), *IWQoS*. In *Lecture Notes in Computer Science: 3552* (pp. 271–285). Springer.

Hahne, E. L., & Gallager, R. G. (1986). Round robin scheduling for fair flow control in data communication networks.. In *IEEE international conference in communications, ICC'86* (pp. 103–107).

Hou, Y. T., Tzeng, H. H.-Y., & Panwar, S. S. (1998). A generalized max-min rate allocation policy and its distributed implementation using ABR flow control mechanism. In *INFOCOM* (pp. 1366–1375).

Jain, S., Kumar, A., Mandal, S., Ong, J., Poutievski, L., Singh, A., et al. (2013). B4: Experience with a globally-deployed software defined wan. *ACM SIGCOMM Computer Communication Review, 43*(4), 3–14.

Jain, S., & Loguinov, D. (2007). PIQI-RCP: Design and analysis of rate-based explicit congestion control. In *Fifteenth IEEE international workshop on quality of service, IWQoS 2007* (pp. 10–20). doi:10.1109/IWQOS.2007.376543.

Jose, L., Yan, L., Alizadeh, M., Varghese, G., McKeown, N., & Katti, S. (2015). High speed networks need proactive congestion control. In *Proceedings of the 14th ACM workshop on hot topics in networks* (p. 14). ACM.

Kalyanaraman, S., Jain, R., Fahmy, S., Goyal, R., & Vandalore, B. (2000). The ERICA switch algorithm for ABR traffic management in ATM networks. *IEEE/ACM Transactions on Networking, 8*(1), 87–98. doi:10.1109/90.836480.

Katabi, D., Handley, M., & Rohrs, C. E. (2002). Congestion control for high bandwidth-delay product networks. In *SIGCOMM* (pp. 89–102). ACM.

Katevenis, M. (1987). Fast switching and fair control of congested flow in broadband networks. *IEEE Journal on Selected Areas in Communications, SAC-5*(8), 1315–1326.

Kaur, J., & Vin, H. (2003). Core-stateless guaranteed throughput networks. In *INFOCOM 2003. twenty-second annual joint conference of the IEEE computer and communications. IEEE societies: 3* (pp. 2155–2165). IEEE.

Kushwaha, V., & Gupta, R. (2014). Congestion control for high-speed wired network: A systematic literature review. *Journal of Network and Computer Applications, 45*, 62–78.

Montresor, A., & Jelasity, M. (2009). PeerSim: A scalable p2p simulator. . In H. Schulzrinne, K. Aberer, & A. Datta (Eds.), *Peer-to-peer computing* (pp. 99–100). IEEE.

Mozo, A., López-Presa, J. L., & Fernández Anta, A. (2011). B-Neck: A distributed and quiescent max-min fair algorithm. In *Network computing and applications (NCA), 2011 10th IEEE international symposium on* (pp. 17–24). doi:10.1109/NCA.2011.10.

Mozo, A., López-Presa, J. L., & Fernández Anta, A. (2012). SLBN: A scalable max-min fair algorithm for rate-based explicit congestion control. In *Network computing and applications (NCA), 2012 11th IEEE international symposium on* (pp. 212–219). doi:10.1109/NCA.2012.40.

Nace, D., & Pióro, M. (2008). Max-min fairness and its applications to routing and load-balancing in communication networks: A tutorial. *IEEE Communications Surveys and Tutorials, 10*(1–4), 5–17.

Ros, J., & Tsai, W. K. (2001). A theory of convergence order of max-min rate allocation and an optimal protocol. In *INFOCOM 2001. Twentieth annual joint conference of the IEEE computer and communications societies. Proceedings IEEE: 2* (pp. 717–726). IEEE.

Tsai, W. K., & Iyer, M. (2000). Constraint precedence in max-min fair rate allocation. In *Communications, 2000. ICC 2000. 2000 IEEE international conference on: 1* (pp. 490–494). IEEE.

Tsai, W. K., & Kim, Y. (1999). Re-examining max-min protocols: A fundamental study on convergence, complexity, variations, and performance. In *INFOCOM* (pp. 811–818).

Tzeng, H.-Y., & Sin, K.-Y. (1997). On max-min fair congestion control for multicast ABR service in ATM. *IEEE Journal on Selected Areas in Communications, 15*(3), 545–556. doi:10.1109/49.564148.

Yang, Y., Wu, G., Dai, W., & Chen, J. (2011). Joint congestion control and processor allocation for task scheduling in grid over OBS networks. *Expert Systems with Applications, 38*(7), 8913–8920.

Zegura, E. W., Calvert, K. L., & Bhattacharjee, S. (1996). How to model an internetwork. In *INFOCOM* (pp. 594–602).

Zhang, Z.-L., Duan, Z., Gao, L., & Hou, Y. T. (2000). Decoupling QOS control from core routers: A novel bandwidth broker architecture for scalable support of guaranteed services. In *Proceedings of the conference on applications, technologies, architectures, and protocols for computer communication*. In *SIGCOMM '00* (pp. 71–83). New York, NY, USA: ACM. doi:10.1145/347059.347403.