institute IMDEA networks

technical report

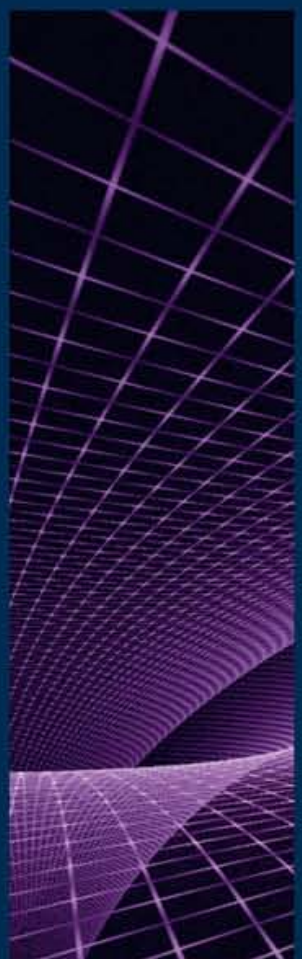
TR-IMDEA-Networks-2011-2

B-Neck: a distributed and quiescent max-min fair algorithm

Alberto Mozo

José Luis López-Presa

Antonio Fernández Anta



April 2011

B-Neck: a distributed and quiescent max-min fair algorithm

Alberto MOZO[‡] Jose Luis LÓPEZ-PRESA[⋆] Antonio FERNÁNDEZ ANTA[†]

[‡] Dpto. Arquitectura y Tecnología de Computadores, Universidad Politécnica de Madrid, Madrid, Spain

[⋆] DIATEL, Universidad Politécnica de Madrid, Madrid, Spain

[†] Institute IMDEA Networks, Madrid, Spain

amozo@eui.upm.es jllopez@diatel.upm.es antonio.fernandez@imdea.org

Abstract

The problem of fairly distributing a network capacity among a set of sessions has been widely studied. In this problem, each session connects via a single path a source and a destination, and its objective is to maximize its assigned transmission rate (i.e., its throughput). Since the links of the network have limited bandwidth, some form of criterion has to be defined to fairly distribute them among the sessions. A popular criterion is *max-min fairness* that, in short, guarantees that each session i gets a rate λ_i such that no session s can increase λ_s without causing another session s' to end up with a rate $\lambda_{s'} < \lambda_s$. Many max-min fair algorithms have been proposed, both centralized and distributed. However, to our knowledge, all proposed distributed algorithms require control packets being continuously transmitted to recompute the max-min fair rates when needed.

In this paper we propose B-Neck, a max-min fair distributed algorithm that 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. As far as we know, B-Neck is the first max-min fair distributed algorithm that does not require a continuous injection of control traffic to compute the rates. When changes occur, affected sessions are asynchronously informed of their new rate (i.e., sessions do not need to poll the network for changes). The correctness of B-Neck is formally proved, and extensive simulations are conducted. In them it is shown that B-Neck converges relatively fast and behaves nicely in presence of sessions arriving and departing.

1 Introduction

The fair distribution of network resources among a set of sessions is a recurring problem. In this problem, each session connects via a single communication path a source node and a destination node in the network, with the objective of maximizing the transmission rate (i.e., throughput) between them. Since the links of the network have limited capacity, the solution of the problem must use some criterion to fairly distribute the network resources among the sessions. A popular fairness criterion to share the available network capacity among a set of sessions without incurring in link overload is the, so called, *max-min fairness* [7, 20]. The basic idea behind the max-min fairness criterion is to first allocate equal bandwidth to all contending sessions at each link, and if a session can not utilize its bandwidth because of constraints elsewhere in its path, then the residual bandwidth is distributed among 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. Then, each session i is allocated a transmission rate λ_i so that no link is overloaded, and a session can only increase its rate at the expense of a session with the same or smaller rate. In other words, *max-min fairness* guarantees that each session i gets a rate λ_i, such that no session s can increase λ_s without causing another session s' to end up with a rate $\lambda_{s'} < \lambda_s$.

Many max-min fair algorithms have been proposed, both centralized and distributed (see the related work below). However, to our knowledge, all proposed distributed algorithms require packets being continuously transmitted to recompute the max-min fair rates when some change in the sessions happens (like new sessions arriving or sessions leaving). In this paper we propose a distributed algorithm, which we call B-Neck, to compute max-min fair rates that is *quiescent*, i.e., once the rates have been computed, the algorithm does not need, generate, nor assume any more traffic in the network. As far as we know, this is the first quiescent distributed algorithm that solves the max-min fairness problem.

1.1 Related Work

We are interested in computing the max-min fair rate allocation for single path sessions. The max-min fair rates of the sessions can be efficiently computed in a centralized way with the Water-Filling algorithm [7, 20]. Max-min fairness has usually been chosen as the target fairness criterion implemented by congestion control protocols to allocate the bandwidth of network links among the sessions that cross them. From a taxonomic point of view, centralized and distributed algorithms have been proposed. The latter have typically been implemented as the congestion control protocols. Another classification of max-min fair algorithms is based on considering if per-session state information is needed in routers, or if, otherwise, only constant information is used.

To our knowledge, the proposals of Gallager [12] and Katevenis [18] were the first to apply max-min fairness to share bandwidth among sessions in a packet switched network. They achieved fairness by implementing a round robin scheduler of packets in each router link. 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 is needed, and updated by every data packet, in each router link. Additionally, a continuous injection of traffic is needed to maintain the system in a stable state.

Later on, when ATM networks appeared, several distributed algorithms were proposed to calculate virtual circuit max-min fair rates in the Available Bit Rate (ABR) traffic mode [2, 6, 9, 8, 13, 24, 25]. These algorithms calculate the exact max-min fair rate assignment 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. Charny et al. [9] seem to have been the first to analytically prove the correctness of their proposed algorithm. Hou et al. [13] generalized the Charny algorithm to extend the max-min fairness criterion with minimum rate requests and peak rate constraints. A problem of the algorithm in [9] (when pseudo-saturated links appear) was identified and documented by Tsai and Kim [24]. 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. Finally, they proposed a distributed version of the algorithm, but no analytical properties nor experimental results for this distributed algorithm were presented.

It is worth noting that all the distributed algorithms mentioned above need per-session state information at the routers of the network. Distributed algorithms that only use constant state information in each router have also been proposed [2, 6, 10]. Several max-min fair algorithms have been proposed that compute an *approximation* of the rates [5, 4]. Unfortunately, it has been shown that max-min fair rates are sensitive to small changes, and hence an approximation with a small difference from the optimal allocation in one session can be drastically amplified at another session [1]. In any case, as already mentioned, all the algorithms proposed need packets to be continuously sent.

Recent research trends in explicit congestion control protocols (XCP [17], JetMax [27], RCP [11], PIQI-RCP [15]) implement efficient congestion controllers in routers, without the need to store and process state information for each session (in the case of RCP, even with a low per packet computational overhead), and guarantee that the max-min fair rate assignments are achieved when controllers are in steady state. In these proposals, the information returned to source nodes from the routers (e.g., an incremental window size or an explicit rate value) allows the sessions to know approximate values that converge eventually to their max-min fair rate assignments, as the system evolves to a steady state. Published results show formal analysis of stability of these protocols, but the experimental evaluations are done on simple topologies composed by a small number of links and sessions (far from real scenarios of large networks with transient dynamics). Additionally, none of them are quiescent, and so, traffic must be injected continuously in to the network in order to keep the system stable.

It is worth mentioning recent related works that explore different versions of the problem. Some of them compute the max-min fair rates with centralized algorithms that can be applied (a) to solve a generalized version of the problem when the bottlenecks are not easy to find (e.g. [21]), or (b) to find session routes that maximize the total bandwidth (e.g.,[3]). Other papers solve the problem in wireless networks, where new challenges appear because neighbors contend even if they use different wireless links. Huang and Bensaou [14] propose an algorithm that collects in the routers the session information of sections of the network, including many sessions that do not cross the router, and in the worst case, the whole set of sessions. Then, the problem is locally solved with a Water-Filling like algorithm. Tassiulas and Sarkar [23] propose a set of scheduling policies for providing fair allocation of bandwidth in the wireless context, but they do not compute max-min fair rates. Finally, some papers consider a multicast version of the problem [19, 22, 25].

1.2 Contributions

In this paper we propose B-Neck, a max-min fair distributed algorithm that is also quiescent. As far as we know, B-Neck is the first such algorithm. Instead of requiring a continuous injection of traffic to compute the max-min fair rates, B-Neck uses a limited number of control packets. In addition, each node only requires information of the sessions that traverse it. When changes in the sessions occur, B-Neck recomputes the new rates and asynchronously informs the affected sessions of their rate (i.e., sessions do not need to poll the network for changes). We have formalized the interaction between the (applications that create and use the) sessions and B-Neck, with a set of primitives. Then, primitives to start and end sessions have been defined (namely, $API.Join$ and $API.Leave$). A primitive that B-Neck uses to notify a session of a change in its rate is also defined (namely, $API.Rate$). Finally, the interface allows a session to fix the maximum rate that it requires both at the time it is created (with $API.Join$) and by using a fourth primitive, defined to change the requested maximum rate (namely, $API.Change$).

The properties of B-Neck are formally proved. This proof has two parts. Firstly, we show its *correctness*, i.e., if sessions do not change (for a time period large enough) B-Neck correctly finds the max-min fair rates of all the sessions, and notifies these rates to them. Secondly, we show *quiescence*, i.e., after computing the rates, eventually B-Neck stops injecting traffic into the network. We want to note that, once B-Neck is quiescent, changes in the sessions (new arrivals, departures, or changes in the requested maximum rates) reactivate it, so that, once the changes end, the new appropriate rates are found and notified, and eventually B-Neck becomes quiescent again.

The properties of B-Neck have been tested with extensive simulations. In these we have used networks of several sizes (with up to hundreds of thousands of nodes), with LAN and WAN characteristics, and with a wide range of session cardinalities (up to hundreds of thousands of sessions). To guarantee the correctness of our implementation of B-Neck, the max-min fair rates obtained have been compared with rates computed with a centralized algorithm (similar to the Water-Filling algorithm [7, 20]). B-Neck has always converged to the right set of max-min fair rates.

Our simulations have shown that B-Neck becomes quiescent very quickly, even in the presence of many interacting sessions. We have also stressed the algorithm by, once quiescent, causing large number of simultaneous departures and rate changes. In all cases B-Neck has shown to be robust and efficient, reaching convergence and quiescence again quickly. The control traffic caused in the network by the algorithm is also shown to be limited, and only for highly dynamic systems with many sessions has more than a few packets per session.

1.3 Structure of the Rest of the Paper

The rest of the paper is structured as follows. In Section 2 definitions and notations are provided. In Section 3 the max-min fair protocol B-Neck is presented, and in Section 4 its correctness is proved. Finally, in Section 5 experimental results are shown.

2 Definitions and Notation

In this section we describe the system model considered, and provide general definitions and notation. We have a network composed by routers and hosts which have directed links connecting them. The network can be modeled as a simple directed graph $G = (V, E)$. The network links may have different propagation delays and different bandwidths. Connected nodes have links in both directions, i.e., $(u, v) \in E \implies (v, u) \in E$. Some routers have hosts connected to them through dedicated links, so that each host is connected to only one router. Sessions follow a static path in the network. This path starts in a host which we call the *source node* of the session, and ends in another host which we call the *destination node*. The intermediate nodes in the path of a session are routers. Each host can only be the source node of one session. (This limitation is just for the sake of simplicity.) For every link $e \in E$, we use C_e to denote its bandwidth[1]. We use $\pi(s)$ to denote the path of a session s, which is a list of links from the source node to the destination node. As it will be described, some packets of a session s in the proposed protocol are sent across links of the path $\pi(s)$. These packets are said to be sent *downstream*. Other packets of session s are sent across links in the reverse path[2] of s. These packets are said to be sent *upstream*.

The problem we face is how to distribute the available network bandwidth among the sessions, assigning max-min fair rates to them. We allow the sessions to specify the maximum rate they need. (This maximum rate may be ∞.) Sessions are considered to be greedy in this context, i.e., they want to maximize their assigned bandwidth up to their maximum requested rate. We also allow the sessions to change their maximum rate request dynamically. The interface between the sessions and the protocol that implements a max-min fair rate assignment (with our additional capabilities) can be specified in terms of the following primitives:

- $API.Join(s, r)$: Used by session s to join the system and request a maximum rate of r.
- $API.Leave(s)$: Used by session s to signal its termination.
- $API.Change(s, r)$: Used by session s to request a new maximum rate of r.
- $API.Rate(s, \lambda)$: Used by the max-min fair algorithm to indicate to session s that its max-min-fair rate is λ.

A session s is *active* if it has invoked $API.Join(s, r)$, and it has not invoked $API.Leave(s)$. We assume that the primitives are used in a sensible way as follows. No active session invokes an $API.Join$ primitive, and only active sessions invoke $API.Leave$ and $API.Change$ primitives. In exchange, the max-min fair algorithm must guarantee that $API.Rate$ is always invoked on active sessions. If, during a period of time, there are no invocations to $API.Join$, $API.Leave$ or $API.Change$ primitives, then we say that the network is in a *steady state* in that period.

Let us consider a network in a steady state period. We denote with S the set of all active sessions in the system, and with S_e the set of sessions in S that cross link e. Let us denote, for each $s \in S$, the maximum rate requested by s as r_s. The max-min fair rates of the system can be computed in a modified system in which the maximum rate requested by each session is ∞, and the effective bandwidth of the first link e in the path of session s is $D_s = \min(C_e, r_s)$. For the sake of simplicity, in the informal descriptions that follow we use this modified system with the notation C_e instead of D_s (but the algorithm B-Neck correctly uses D_s).

[1] We assume that this is the bandwidth allocated to data traffic, and that the control traffic caused by the max-min fair algorithm does not consume any of this bandwidth.

[2] The reverse path of $\pi(s)$ traverses the same sequence of nodes in reverse order.

1 **for each** $e \in E$ **do** $R_e \leftarrow S_e$; $F_e \leftarrow \emptyset$ **end for**
2 $L \leftarrow \{e \in E : R_e \neq \emptyset\}$
3 **while** $L \neq \emptyset$ **do**
4 **for each** $e \in L$ **do** $B_e \leftarrow (C_e - \sum_{s \in F_e} \lambda^*_s)/|R_e|$ **end for**
5 $B \leftarrow \min_{e \in L}\{B_e\}$; $L' \leftarrow \{e \in L : B_e = B\}$; $X \leftarrow \bigcup_{e \in L'} R_e$
6 **for each** $s \in X$ **do** $\lambda^*_s \leftarrow B$ **end for**
7 **for each** $e \in L \setminus L'$ **do** $F_e \leftarrow F_e \cup (R_e \cap X)$; $R_e \leftarrow R_e \setminus F_e$ **end for**
8 $L \leftarrow \{e \in (L \setminus L') : R_e \neq \emptyset\}$
9 **end while**

Figure 1: Centralized B-Neck Algorithm.

Let us denote by λ^*_s the max-min fair rate of session $s \in S$.

Definition 1 *For any session s, a link $e \in \pi(s)$ is a* bottleneck *of s iff $\sum_{s' \in S_e} \lambda^*_{s'} = C_e$ and $\forall s' \in S_e, \lambda^*_{s'} \leq \lambda^*_s$.*

A link e is a *bottleneck* of the system if it is a bottleneck for every session in S_e. If a link e is a bottleneck of a session s, we say that s is *restricted* at e. Otherwise we say that s is *unrestricted* at e. In any max-min fair system, every session is restricted in at least one link, and hence has at least one bottleneck. It is also known that any max-min fair system has at least one bottleneck [7].

For each link e, the sets R^*_e and F^*_e are defined as $R^*_e = \{s : e \text{ is a bottleneck of } s\}$ and $F^*_e = S_e \setminus R^*_e$. Observe that all sessions in R^*_e have the same rate. We denote this rate as B^*_e and call it the *bottleneck rate* of link e. If $R^*_e \neq \emptyset$, then this rate can be computed as $B^*_e = (C_e - \sum_{s \in F^*_e} \lambda^*_s)/|R^*_e|$. Then, every session $s \in F^*_e$ has $\lambda^*_s < B^*_e$. Observe that, if $R^*_e = \emptyset$, then the bandwidth of link e is not fully assigned to the sessions (i.e., $\sum_{s \in S_e} \lambda^*_s < C_e$).

3 B-Neck Algorithm

In this section, we describe the algorithm B-Neck. We start by presenting a centralized algorithm that conveys most of the intuition of the logic of B-Neck. Then, we will describe how the logic of this centralized algorithm is translated into a distributed form in the final algorithm.

3.1 Centralized B-Neck

As we said, before tackling the problem of computing max-min fair rates with a distributed algorithm, we will introduce a simple centralized iterative solution that illustrates how these rates may be computed. This algorithm computes the max-min fair rates in a form that is similar to the Water-Filling algorithm [7]. The Centralized B-Neck algorithm is presented in Figure 1. This algorithm discovers bottlenecks iteratively, in increasing order of their bottleneck rates. To do so, it computes estimates of the *bottleneck rates* $B_e = (C_e - \sum_{s \in F_e} \lambda^*_s)/|R_e|$ for each link e such that $R_e \neq \emptyset$.

In the first iteration, the bottlenecks of the system are discovered. Since for these bottlenecks $F^*_e = \emptyset$ and, initially, the variable $F_e = \emptyset$, the rates $B_e = B^*_e$ are computed correctly. Then, all the sessions s that cross these links are assigned their rates $\lambda^*_s = B_e = B^*_e$, which is the bottleneck rate of these links. Once these sessions have got their rates assigned, a new network configuration is generated. First, all the sessions that have their rates assigned, are moved from R_e to F_e in every link e of their paths at which they are unrestricted (those links in $L \setminus L'$). Thus, their rates will be taken away in the computation of the following bottleneck rates. Then, all the bottleneck links discovered are removed from the system, together with those links e that have no session in R_e. The process continues until $L = \emptyset$. This procedure correctly computes the max-min fair rates [7], and guarantees that $R_e = R^*_e$ and $F_e = F^*_e$.

As we mentioned, B-Neck follows a logic similar to the one used in the Centralized B-Neck algorithm. From the point of view of sessions, the evolution of the algorithm can be seen as follows. For each session, the bottleneck rates of every link in the session's path are computed. The smallest such bottleneck rate (what we call the bottleneck rate of the session) is the rate assigned to the session. Doing this in increasing order of the bottleneck rate of the sessions yields the solution. However, in order to deal with the distributed and dynamic nature of the system, many problems have to be solved.

```
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 ((η = e ∧ λ = B_e) ∨ (η ≠ e ∧ λ ≤ B_e)) then
32              μ_s^e ← IDLE
33              λ_s^e ← λ
34          else // ((η = e ∧ λ < B_e) ∨ (λ > B_e))
35              τ ← UPDATE
36              μ_s^e ← WAITING_PROBE
37          end if
38          if ∀r ∈ R_e, λ_r^e = B_e ∧ μ_r^e = IDLE 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, λ_r^e = B_e ∧ μ_r^e = IDLE then
76          send downstream SetBottleneck (s, TRUE)
77      else if λ_s^e < B_e ∧ μ_s^e = IDLE 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
```

Figure 2: Task Router Link (RL).

3.2 Protocol packets

The B-Neck algorithm runs in every network link, and the source and destination nodes of each session. Sessions communicate with B-Neck through the API primitives already described, and the entities that run the B-Neck algorithm interact exchanging B-Neck packets. The B-Neck packets are:

- $Join(s, \lambda, \eta)$: Sent downstream along the path of session s to inform the links of the arrival of a new session. λ is the estimated bottleneck rate of the session, and η is the link with the smallest bottleneck rate found in the

```
1  task SourceNode (s, e)
2  // e is the first link of s
3  // and it is not shared with any other session
4
5  when API.Join(s, r) do
6      R_e ← {s}
7      D_s ← min(r, C_e)
8      upd_rcv_s ← FALSE
9      bneck_rcv_s ← FALSE
10     μ_s^e ← WAITING_RESPONSE
11     send downstream Join (s, D_s, e)
12 end when
13
14 when API.Leave(s) do
15     F_e ← ∅; R_e ← ∅
16     send downstream Leave (s)
17 end when
18
19 when API.Change(s, r) do
20     D_s ← min(r, C_e)
21     if μ_s^e = IDLE then
22         if s ∈ F_e then
23             F_e ← ∅; R_e ← {s}
24         end if
25         upd_rcv_s ← FALSE
26         bneck_rcv_s ← FALSE
27         μ_s^e ← WAITING_RESPONSE
28         send downstream Probe (s, D_s, e)
29     else
30         upd_rcv_s ← TRUE
31     end if
32 end when
33
34 when received Update (s) do
35     if μ_s^e = IDLE then
36         if s ∈ F_e then
37             F_e ← ∅; R_e ← {s}
38         end if
39         bneck_rcv_s ← FALSE
40         μ_s^e ← WAITING_RESPONSE
41         send downstream Probe (s, D_s, e)
42     else
43         upd_rcv_s ← TRUE
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)
51         if D_s > λ_s then
52             F_e ← {s}; R_e ← ∅
53         end if
54         send downstream SetBottleneck (s, D_s = λ_s)
55     end if
56 end when
57
58 when received Response (s, τ, λ, η) do
59     if τ = UPDATE ∨ upd_rcv_s then
60         upd_rcv_s ← FALSE
61         bneck_rcv_s ← FALSE
62         μ_s^e ← WAITING_RESPONSE
63         send downstream Probe (s, D_s, e)
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 then
70             F_e ← {s}; R_e ← ∅
71         end if
72         send downstream SetBottleneck (s, D_s = λ_s)
73     else // τ = RESPONSE
74         λ_s^e ← λ
75         μ_s^e ← IDLE
76         if D_s = λ_s then
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
```

Figure 3: Task Source Node (SN).

path.

- $Probe(s, \lambda, \eta)$: Like $Join$, but sent at any time when the rate for session s needs to be recomputed.
- $Response(s, \tau, \lambda, \eta)$: Sent upstream from the destination node to the source node, indicating the rate λ that can be assigned to s, which link η imposed the strongest rate restriction, and an indication τ of the next action to be performed.
- $Update(s)$: Sent upstream to the source node indicating that a new Probe cycle (see below) must be performed for session s.
- $Bottleneck(s)$: Sent upstream to the source node indicating that the current rate of session s has to be assumed to be its max-min fair rate.
- $SetBottleneck(s, \beta)$: Sent downstream from the source node, indicating that the current rate for session s is assumed to be its max-min fair rate, and the links e that do not restrict s must move it from R_e to F_e. Parameter β is used to check that there is at least one bottleneck for session s.
- $Leave(s)$: Sent downstream from the source, so that all the links in the path of session s may delete all data corresponding to this session, and the network is reconfigured.

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

Figure 4: Task Destination Node (DN).

3.3 A global perspective of B-Neck

In this section, we describe the distributed algorithm B-Neck, an event driven, distributed and quiescent algorithm which computes max-min fair rate allocations and notifies them to the sessions, in a network environment where sessions can asynchronously join and leave the network, or change their maximum desired rate.

Like other distributed algorithms that compute max-min fair rates, B-Neck needs to keep some per-session information at the links. It uses sets R_e and F_e (as in Centralized B-Neck) to store the sessions that are restricted at this link, and those that are restricted somewhere else, respectively, just like the centralized version. Besides, for each session s, it is necessary to store its state $\mu_s^e \in \{\text{IDLE}, \text{WAITING\_PROBE}, \text{WAITING\_RESPONSE}\}$ and its assigned rate λ_s^e, at each link. The assigned rate is meaningful only when $s \in F_e$ or $s \in R_e$ and $\mu_s^e = \text{IDLE}$. Whenever necessary, a link computes its bottleneck rate as $B_e = (C_e - \sum_{s \in F_e} \lambda_s^e)/|R_e|$. Note that this computation resembles the one performed in Centralized B-Neck, with the only difference that in the centralized version, the rates assigned to sessions in F_e are always the max-min fair rates λ_s^*, and in the distributed version the rates λ_s^e might (temporarily) not be the max-min fair rates. However, 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.

At the source nodes, the session's maximum desired rate D_s is kept in order to start new Probe cycles in the future (Probe cycles are described below). Additionally, two flags are used: $bneck\_rcv_s$ which indicates that a max-min-fair assignment has been made to session s, and upd_rcv_s which indicates that an Update packet has arrived during a Probe cycle, and a new Probe cycle must be started after the current one ends.

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

Whenever a session joins, changes its rate requirement or receives an Update packet from the network, it performs a *Probe cycle*. A Probe cycle is the basic procedure on which B-Neck relies to compute the max-min fair rates. A Probe cycle starts with the source node sending of a Probe packet (or Join when a session arrives) that, regenerated at each link, traverses the whole path of the session. At the destination node, a Response packet is generated and sent back to the source (regenerated at each link of the reverse path). A source node never starts a Probe cycle if there is one in course (this is guaranteed by flag upd_rcv_s). Note that, since new Probe cycles are only triggered by the network 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). When sessions stop joining, leaving or changing their rate requirement, the network will eventually get stable, and no more traffic will be generated by B-Neck until there is another change in the sessions configuration.

The Response packets that close a Probe cycle are used at the links to assign rates to the sessions, and to detect bottleneck conditions. A link identifies itself as a bottleneck when all the sessions, that are not restricted somewhere else, have completed a Probe cycle, are IDLE and have been assigned the same rate. In this case, all these sessions

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

are sent a Bottleneck packet to inform them that their rate is stable (and corresponds to the max-min fair rate with the present sessions configuration). The session that is performing the probe cycle receives the indication in the form of a Response packet with $\tau = \text{BOTTLENECK}$.

When a session joins the network, its source node sends a Join packet that traverses the path of this session, and serves two purposes: (1) it informs the links that a new session has arrived (and adds that session to their R_e sets), so they can recompute their bottleneck rates and send Update packets to the affected sessions that may have their rates reduced, and (2) it acts like a Probe packet gathering information of the rate that corresponds to this session. When a session leaves the network, its source node sends a Leave packet that traverses the path of this session, so the links may delete all the information related with the session, and send Update packets to the affected sessions that could increase their assigned rate.

When a session receives the Bottleneck communication, it sends a SetBottleneck packet to inform the links in its path that the rate is stable. Additionally, if a link is not a bottleneck for that session, i.e. the session's rate is lower than the links bottleneck rate B_e, the session is moved form R_e to F_e, so it is reconsidered in the computation of B_e. When a SetBottleneck packet reaches the destination node without having found a bottleneck for a session (what is controlled with the β field of the SetBottleneck packet), it means that there has been a change in the network. Then, the session is informed by the destination node with an Update packet, to trigger a new Probe cycle. Note that, during the Probe cycle, the session must come back to set R_e to recompute B_e at each link.

B-Neck discovers bottlenecks in a similar fashion to Centralized B-Neck. However, since B-Neck is a distributed algorithm, concurrently executed by all the network links, bottlenecks may be discovered in parallel, what speeds up convergence. However, sometimes, a bottleneck l might be incorrectly identified before a bottleneck l' on which it depends, what would never happen with the centralized algorithm. Nevertheless, when bottleneck l' is identified, the SetBottleneck packets that will be sent afterwards will generate the necessary Update packets to be sent to the affected sessions, and the first bottleneck l will eventually be correctly identified.

4 Proof of Correctness of B-Neck

In this section we prove the correctness of algorithm B-Neck. First, recall that Algorithm B-Neck uses for each link e, among others, the variables R_e, F_e, and for each $i \in R_e \cup F_e$ variables μ_i^e and λ_i^e. When $R_e \neq \emptyset$, let us define, $B_e = (C_e - \sum_{j \in F_e} \lambda_j^e)/|R_e|$. Observe that when $R_e = R_e^*$ and $F_e = F_e^*$, then $B_e = B_e^*$.

Definition 2 *Consider a network executing the B-Neck protocol, a link e is stable if $\forall i \in R_e \cup F_e, \mu_i^e = \text{IDLE}$, $\forall i \in R_e, \lambda_i^e = B_e$, and if $R_e \neq \emptyset$, then $\forall i \in F_e, \lambda_i^e < B_e$. The network is stable at a time t if all the links are stable and there is no packet of the B-Neck protocol in the network, neither in transit nor being processed at a link (i.e., no link is executing a* **when** *block).*

Observation 1 *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.*

Lemma 1 *Consider a network, which executes B-Neck, in a steady state. If an infinite number of packets of any kind is generated while in steady state, then some session generates an infinite number of Probe packets.*

Proof: If there is an infinite number of Bottleneck packets, there must be an infinite number of Response packets because a Bottleneck packet is only generated when processing a Response packet (Line RL42), and the processing of each Response packet only generates a finite number of Bottleneck packets (Lines RL41 – 43). If there is an infinite number of Response packets, there must be an infinite number of Probe packets because (if there are no more Join packets) a Response packet is only generated at the destination node of the session when a Probe packet is received (Line DN4). If there is an infinite number of Update packets, (since there are no more Join or Leave packets) there must be an infinite number of Probe packets (Lines RL53 and RL13) or SetBottleneck packets (Lines RL81 and DN13). If there is an infinite number of SetBottleneck packets, there must be an infinite number of Probe packets, because the use of $bneck\_rcv_s$ in Task SourceNode guarantees that, between two SetBottleneck packets generated for a session, at least one Probe packet is generated. Hence, if an infinite number of packets of any kind is generated, then an infinite

number of Probe packets is generated. Since the number of sessions is finite, then some session must generate an infinite number of Probe packets. ■

Lemma 2 *Consider a network, which executes B-Neck, in a steady state. If an infinite number of packets of any kind is generated while in steady state, then some session updates the λ variable of its source node an infinite number of times.*

Proof: Let us assume that an infinite number of packets is generated. From Lemma 1 some session s generates an infinite number of Probe packets. Then, the source node of s receives an infinite number of Response packets. If the condition at Line SN59 evaluates to FALSE infinite times, then the λ variable of the source node is updated an infinite number of times. Otherwise, there is a time after which all Response packets received at the source node of session s have $\tau = \text{UPDATE}$, or an Update packet was previously received (setting upd_rcv_s to TRUE).

In the first case, if a Response is received with $\tau = \text{UPDATE}$, then in the link e where τ was set to UPDATE (Lines RL34 – RL37), some session was moved from F_e to R_e (case $\lambda > B_e$) by the processing of a Probe packet, or from R_e to F_e (case $\eta = e \wedge \lambda < B_e$) by the processing of a SetBottleneck packet, between the processing of the Probe and the corresponding Response of s.

In the second case, since s does not generate SetBottleneck packets, Update packets can only be generated in a link e at Line RL13 when processing a Probe packet of a session $s' \in F_e$ that is moved to R_e, or at Line RL87 when processing a SetBottleneck of a session s' that is moved from R_e to F_e. Hence, some session s', at some edge e, is moved from F_e to R_e and vice versa an infinite number of times. The movement from R_e to F_e is only done when processing a SetBottleneck packet (Line RL83). Hence there are infinite SetBottleneck packets and this implies infinite updates of variable λ at the source node of s'. ■

Definition 3 *A session s causes a Probe packet of another session $s' \neq s$ if (a) an Update packet of session s' is generated in Lines RL13 or RL81 when processing a packet from session s, or (b) an Update packet of session s'' is generated in Lines RL13 or RL81 when processing a packet of session s, and s'' and s' share a bottleneck, what causes a SetBottleneck of s' to reach its destination node with $\beta = \text{FALSE}$. In this case, at the destination node, an Update packet is generated (Line DN13).*

Observation 2 *When Procedure $ProcessNewRestricted$ is called from Line RL53 when processing a Probe packet for session s', only sessions s such that $\lambda^e_{s'} < \lambda^e_s$ may be sent an Update packet.*

Lemma 3 *Consider a network, while executes B-Neck, in a steady state. Then, there is a time after which every Probe packet generated for session s at time t (after it updates its source node λ variable) is caused by the processing of a Probe or SetBottleneck packet of another session s' generated at time t' such that the λ variables of the source nodes of both sessions, namely λ_s and $\lambda_{s'}$, satisfy that $\lambda_{s'}$ at time t' is smaller than λ_s at time t.*

Proof: When no session joins nor leaves the network, or changes its rate requirement, a new Probe packet is generated at the source node when an Update packet is received, either generated immediately (Line SN41) or deferred to the reception of a Response packet (Line SN63) by setting $upd\_rcv_s = \text{TRUE}$ (Line SN43).

Update packets can be generated by (a) a call to procedure $ProcessNewRestricted$ when processing a Probe packet (Line RL53), (b) when processing a SetBottleneck packet at a router link (Line RL81), or (c) when processing a SetBottleneck packet at the destination node (Line DN13).

- Case (a): When Procedure $ProcessNewRestricted$ is called, from Observation 2, the session s' that generates the Update packets (and, hence, causes the subsequent Probe packets) only sends Update packets to sessions s such that $\lambda^e_{s'} < \lambda^e_s$. Since the sessions s are IDLE, the λ values at their source nodes, when the Update packet is received, will be λ^e_s.
- Case (b): If Line RL81 has been reached, then the condition of Line RL77 holds. Hence, $\lambda^e_{s'} < B_e$, and only sessions s such that $\mu^e_s = \text{IDLE}$ and $\lambda^e_s = B_e$ are sent an Update packet (Lines RL77 – RL82). Hence, the λ values at their source nodes, when the Update packet is received, will be λ^e_s.

- Case (c): The processing of a SetBottleneck packet at the destination node of session s generates an Update packet if the received flag $\beta = \text{FALSE}$. Let us consider the link e where the bottleneck was detected, which generated a Bottleneck packet for s (Line RL42) or set $\tau = \text{BOTTLENECK}$ in the Response packet of s (Line RL39). At the time when this happened, the bottleneck condition $\forall r \in R_e, \lambda_r^e = B_e \wedge \mu_r^e = \text{IDLE}$ was satisfied (Line RL38). If the corresponding SetBottleneck packet is received at the destination node of session s with $\beta = \text{FALSE}$ then, when the following SetBottleneck packet was processed at link e, the former bottleneck condition was not fulfilled (Line RL75). However, session s remains with $\mu_s^e = \text{IDLE}$. If this were not true, the SetBottleneck packet would not have reached the destination node, because it is a necessary condition to forward the SetBottleneck packet (Lines RL75, RL77 and RL85).
  If the bottleneck condition does not hold when the SetBottleneck packet of session s reached link e, then a session s' must have changed its state $\mu_{s'}^e$ or its assigned bandwidth $\lambda_{s'}^e$ at some time since the bottleneck condition was detected. If its $\lambda_{s'}^e$ has changed, then its state must have changed before. Let us consider the first session $s' \in R_e$ that changed its state after the bottleneck condition was detected.
  If s' was in F_e when the bottleneck condition was detected, and moved to R_e (what is only done when Join or Probe packets are processed), then all the sessions in R_e would have changed their state to WAITING_PROBE, among them session s (Line RL12 called from Line RL53), and this is impossible since μ_s^e remains IDLE. Hence, s' was in R_e and remains in R_e.
  As stated before, a Probe or Update packet for session s' reached link e. Therefore, the former three cases (a,b and c) may arise. In Case (c), when the SetBottleneck packet for s' crossed link e, the bottleneck condition must have hold because it was the first session that changed ts state after t'. Hence, Line RL76 must have been executed and so, β must have been set to TRUE when the SetBottleneck packet reached the destination node. Thus, the Update packet for session s' could not be generated at its destination node. Hence, the only possible cases are (a) and (b) previously discussed, so there must be a session s'' with its λ variable smaller than that of session s' (and hence smaller than that of s) that caused the Update packet of s', and transitively the Update packet of s.

Thus we conclude that only a session with a smaller λ may cause a probe of another session, either directly or transitively. ■

Observation 3 *Since the number of sessions in the network is finite and the possible contents of the sets F_e and R_e at each node e are finite, and the λ variables take the values of $B_e = (C_e - \sum_{s \in F_e} \lambda_s^e)/|R_e|$, then the number of different values of the λ variables is finite.*

Lemma 4 *Consider a network, executing the B-Neck, in a steady state. Then no session has an infinite number of packets.*

Proof: Let us assume that an infinite number of packets is generated. From Lemma 2 there are sessions that update the λ variable at their source node an infinite number of times. Since there are finitely many possible values that can be assigned to the λ variables, there are values that are assigned infinitely many times to the λ variables of the sessions' source nodes.

Consider a time t after which the values assigned finitely many times are not assigned anymore to any λ variable of a source node. However, (after time t) a session s may be assigned a value that is assigned an infinite number of times to some other session s', although it is only assigned a finite number of times to session s. Let x be the smallest value among those values assigned infinitely many times. Since the number of sessions is finite, some session s updates the λ variable of its source node, with value x, after time t. We will show that there is a time after which s does not generate new Probe packets, and hence its source node eventually stops receiving Response packets and updating its λ variable, what will contradict the claim that s updates its λ variable an infinite number of times.

From Lemma 3, eventually, every time the λ variable of the source node of s is assigned x, the next Probe packet is caused by another session whose λ variable at the source node has a value smaller than x.

Consider first a session that updates its λ variable at the source node an infinite number of times. After time t, its λ variable will eventually be assigned a value no smaller than x, from the definition of x. Then it will not be able to cause more Probe packets of s. Hence, it will not be able to cause an infinite number of Probe packets of s.

Consider now a session s' that updates its λ variable at the source node a finite number of times. We will show that, after time t, it can cause only a finite (small) number of Probe packets of s. We will consider two cases:

- Assume first that, after time t, session s' causes more than one Probe packet at Line RL81 when processing SetBottleneck packets of s' (generating either an $Update(s)$ or an $Update(s'')$, see cases (a) and (b) above). When a SetBottleneck is generated at the source node of s', the flag $bneck\_rcv_{s'}$ is set to TRUE. To send the next SetBottleneck packet, it has to be reset to FALSE at Lines SN39 or SN61, what forces $\mu_{s'}$ to be set to WAITING_RESPONSE. Then, it has to wait until $\mu_{s'}$ is set back to IDLE (Lines SN66 or SN75), what forces $\lambda_{s'}$ to be updated with a value no smaller than x. Then, from Lemma 3, it will not cause more Probe packets of s. Hence, it will not be able to cause an infinite number of Probe packets of s.
- Assume now that, after time t, session s' causes more than one Probe packet at Line RL13 when processing Probe packets of s' (generating either an $Update(s)$ or an $Update(s'')$, see cases (a) and (b) above). Every time this happens at any edge e, session s' is moved from F_e to R_e (Line RL52). The only way to move a session from R_e to F_e is with a SetBottleneck packet. Then, after a Probe packet is caused at Line RL13, a SetBottleneck packet has to be generated before another Probe packet can be caused this way. This reduces this case to the previous one, in which several SetBottleneck packets are generated (since $\lambda_{s'}$ will be updated with a value no smaller than x, from Lemma 3, it will not cause more Probe packets of s).

■

Lemma 5 *Consider a network, executing B-Neck, in a steady state. Then there is a finite time after which the network is permanently stable, i.e., after that time, the values of R_e, F_e, μ_i^e, and λ_i^e for all e and i do not change.*

Proof: From Lemma 4, if the network is in a steady state, then no session generates an infinite number of packets. Hence, there is a time at which the network becomes stable. Now, observe that after the network becomes stable, it is permanently stable and the values of R_e, F_e, μ_i^e, and λ_i^e for all e and i do not change. This follows from the fact that the links change their state or generate new packets only when they receive some packet. Then, since there are no packets in the network and no new external (Join nor Leave) packets will be received, the above claim follows. ■

Next observation follows from the management of join and leave packets. Note that the Join and Leave packets are always propagated to the destination nodes (Lines RL24 and RL101), and sessions are correctly added and removed from the corresponding sets (Line RL18 during Join, and Lines RL92 – RL96 during Leave). Note also that, when a session is extracted from a set F_e or R_e, it is immediately added to the other set (Lines RL9, RL52, and RL83). At the source node, sessions are registered the same way, with the difference that each source node has only one session. During Join, the session is stored in set R_e (Line SN6). During Leave, it is deleted (Line SN15). In any other case where a session leaves one set, it is stored in the other immediately (Lines SN23, SN37, SN52, and SN70).

Observation 4 *In stability the set of sessions at each link e is correct, i.e. $S_e = R_e \cup F_e$.*

Lemma 6 *Consider a network executing the B-Neck protocol, and a time at which the network is stable. Then, for each session i, for each two links e and e' in the path $\pi(i)$ of session i it holds that $\lambda_i^e = \lambda_i^{e'}$.*

Proof: In order for the network to reach stability, all sessions must send their SetBottleneck packets, since they are the last packets sent by the source nodes, and they have no response when there are no changes in the network. The source can only send a SetBottleneck packet when it receives (a) a Response packet with $\tau = \text{BOTTLENECK}$, or (2) when it receives a Bottleneck packet. In the former case (a) the same λ value is established at every link in the session's path (Line RL33) and at the source node (Line SN65). In the latter case (b) a Response packet was previously received at the source node, which established the same λ value at every link in the session's path (Line RL33) and at the source node (Line SN74). ■

Lemma 7 *Consider a network executing the B-Neck protocol, and a time at which the network is stable. Then, for each session i, there is some link e in the path $\pi(i)$ of session i such that $i \in R_e$.*

Proof: The source node of every session must send a SetBottleneck packet when it receives the Bottleneck indication, either in a Response packet (Line SN72) or with the Bottleneck packet (Line SN54), or when it identifies itself as the bottleneck for the session (Lines SN79). In all cases, if the first link $e \in \pi(i)$ is a bottleneck link for the session, it remains in R_e, while, if it is not, the session is moved to F_e (Lines SN51 – SN53, and SN69 – SN71). At any other link, the only way for a session to get to F_e at some link is when processing a SetBottleneck packet (Line RL83). The SetBottleneck packet must reach the destination node with $\beta = \text{TRUE}$. Otherwise it would cause an Update packet (Line DN13). Hence, either at the first link $\beta = \text{TRUE}$ because $D_s = \lambda_s$ and the session remained in R_e, or at some other link e, Line RL76 must have been executed for session i and, hence, Lines RL77 – RL84 were not executed at link e, so session i was not moved from R_e to F_e, and remains in R_e. ■

We will denote by λ_i the value λ_i^e of every link $e \in \pi(i)$ of session i when the network is stable. λ_i is well defined from Lemma 6.

Lemma 8 *Consider a network executing the B-Neck protocol, and a time at which the network is stable. Then, for each session i, every link e in the path $\pi(i)$ of session i has $\lambda_i^e = \lambda_i^*$.*

Proof: Let us assume, in order to reach a contradiction, that there is (at least) a session i such that $\lambda_i \neq \lambda_i^*$. Let us consider, among the sessions that satisfy $\lambda_i \neq \lambda_i^*$, a session s that has minimal λ_s^*. In case of a tie, the session s with smallest λ_s is chosen (breaking ties arbitrarily). Let e be a bottleneck of session s, as in Definition 1. Consider the sets R_e^* and F_e^* as introduced after Definition 1. Observe that $s \in R^*$ and $B_e^* = \lambda_s^*$. The values of R_e, F_e, and λ_s^e are those computed by Protocol Bneck at link e. We study different cases.

(1) $F_e = \emptyset$. From Observation 4, $R_e = S_e$. From the definition of B_e and $F_e = \emptyset$, $\lambda_s^e = B_e = C_e/|S_e|$. If $F_e^* = \emptyset$, then $R_e^* = S_e$, and from the definition of B_e^* we obtain that $\lambda_s^* = B_e^* = C_e/|S_e| = \lambda_s^e$. This is a contradiction, since we assumed initially that $\lambda_s^e \neq \lambda_s^*$.

On the other hand, assume $F_e^* \neq \emptyset$. Since e is bottleneck for s, $\lambda_s^* = B_e^*$. Consider the session $s' \in F_e^*$ with the smallest value $\lambda_{s'}^*$ (breaking ties arbitrarily). Then, we have that $\lambda_{s'}^* < C_e/|S_e| < B_e^* = \lambda_s^*$. From the definition of B_e and $F_e = \emptyset$, $\lambda_{s'}^e = C_e/|S_e|$. Therefore, $\lambda_{s'}^e = \lambda_{s'} \neq \lambda_{s'}^*$ and $\lambda_{s'}^* < \lambda_s^*$, which yields a contradiction from the minimality of λ_s^*.

(2) $F_e \neq \emptyset$. We consider several cases.

(2.a) $s \in R_e$. Let us consider first that $\lambda_s < \lambda_s^*$. We show that all the sessions i in R_e^* have a smaller rate λ_i assigned by the protocol than the max-min rate λ_i^* and all the capacity C_e is consumed in both cases. Then, some session in F_e^* has a larger rate with the protocol, which leads to a contradiction. Recall that $s \in R_e^*$. Then, every session $i \in R_e^*$ has $\lambda_i^* = \lambda_s^*$ and, since $s \in R_e$, $\lambda_i \leq \lambda_s < \lambda_i^*$ (independently on whether i is in R_e or F_e). Since $R_e \neq \emptyset$ and $R_e^* \neq \emptyset$, all the capacity C_e is used. Hence, there is some session $s' \in F_e^*$ that has $\lambda_{s'} > \lambda_{s'}^*$. Furthermore, $\lambda_{s'}^* < \lambda_s^*$ given that $s \in R_e^*$ and $s' \in F_e^*$. This contradicts the assumption that λ_s^* is minimal.

Let us now consider that $\lambda_s > \lambda_s^*$. Since $s \in R_e^*$, no session has a max-min fair rate larger than λ_s^* (i.e., $\forall i \in S_e : \lambda_i^* \leq \lambda_s^*$) and the whole capacity C_e is assigned (since $R_e^* \neq \emptyset$). Then, some session s' must have a rate assigned by the protocol smaller than its max-min fair rate, $\lambda_{s'} < \lambda_{s'}^*$. Since $\lambda_{s'}^* \leq \lambda_s^*$, the existence of s' contradicts the assumption that λ_s^* is minimal breaking ties with λ_s.

(2.b) $s \in F_e$. If $\lambda_s > \lambda_s^*$, the same argument of the previous proof (case (2.a) with $\lambda_s > \lambda_s^*$) applies and we reach a contradiction.

If $\lambda_s < \lambda_s^*$, from Lemma 7, there is some link e' in the path of s such that $s \in R_{e'}$. We consider two cases. If $s \in R_{e'}^*$, then from the analysis of the case (2.a) with $\lambda_s < \lambda_s^*$ applied to edge e' we reach a contradiction.

Otherwise, we have that $\lambda_s < \lambda_s^*$, $s \in R_{e'}$, and $s \in F_{e'}^*$. Since $R_{e'} \neq \emptyset$, the protocol has assigned the whole capacity $C_{e'}$ to the sessions. Then, the protocol has assigned some session s' a rate $\lambda_{s'}$ larger than its max-min rate $\lambda_{s'}^*$. Since $s \in R_{e'}$, no session has been assigned by the protocol a rate larger than λ_s. Hence, $\lambda_s^* > \lambda_s \geq \lambda_{s'} > \lambda_{s'}^*$. In this case session s' has a smaller max-min fair rate than s and has $\lambda_{s'} \neq \lambda_{s'}^*$. This contradicts the minimality of λ_s.

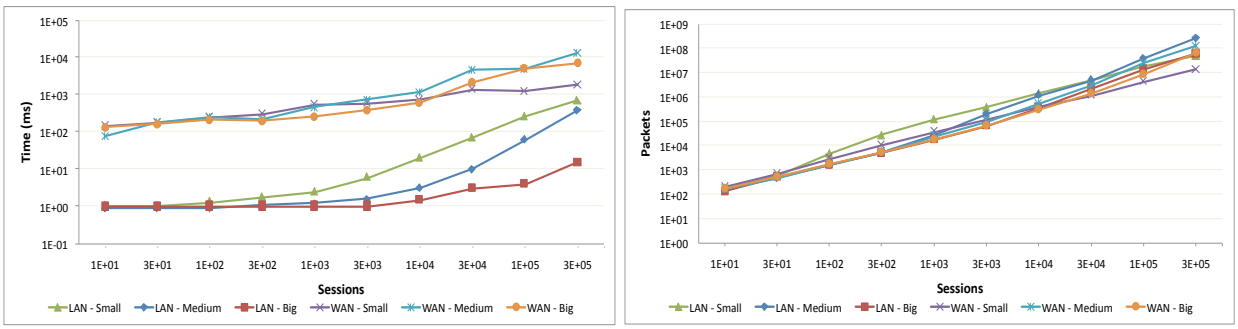


Figure 5: Time until quiescence and number of packets observed in Experiment 1.

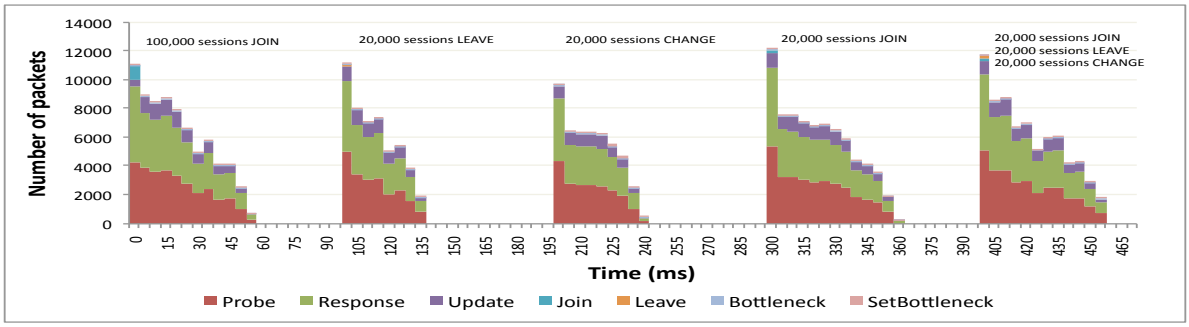


Figure 6: Traffic details of B-Neck in Experiment 2.

■

Theorem 1 *Consider a network, executing the B-Neck protocol, in a steady state. Then, eventually, the network becomes permanently stable, and all sessions are assigned their max-min fair rate.*

Proof: Follows directly from Lemma 5 and Lemma 8. ■

5 Experimental evaluation

In order to get realistic evaluation results, we have coded the B-Neck algorithm in Java and run it on top of a discrete event simulator. We have modified a version of Peersim [16] to be able to run B-Neck with thousands of routers and up to a million hosts and sessions. Peersim has been adapted and optimized into our modified version, *Peersim+*, which 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 [26], and (c) modelling several network parameters, like processing time in routers, and transmission and propagation times in the network links.

The simulations have been run on three network topologies of different sizes, formed by 110 routers (*Small* network), 1100 routers (*Medium* network) and 11,000 routers (*Big* network), respectively, and up to 600,000 hosts. These topologies have been generated using the *gt-itm* graph generator configured with a typical Internet transit-stub model [26]. The physical link bandwidths have been configured to 100 Mbps in the links between hosts and stub routers, 200 Mbps in the links between stub routers, and 500 Mbps in the transit routers' links. Propagation times in network links have been modeled in two ways to evaluate B-Neck with two different scenarios. Firstly, in what we call *LAN scenario*, the propagation time has been fixed to 1 microsecond in every link, as in 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, in what we call *WAN scenario*, all links except host to router links have been assigned a propagation time generated uniformly at random in the range of 1 to 10 milliseconds. All the links between hosts and routers are assigned 1 microsecond of propagation time. This second scenario has a resemblance with an Internet topology where the propagation times in the internal network links are in the range of a typical WAN link. In this kind of networks, Probe cycles are completed more slowly and more interactions with packets from other sessions are potentially produced than in the LAN scenario. In the experiments, sessions have been created by choosing a source and a destination node, uniformly at random among all the network hosts. A session path is a shortest path from its source to its destination node. In order to check the correctness of the results obtained when executing B-Neck, we have programmed Centralized B-Neck (Figure 1), and so, every B-Neck execution result (the max min fair rate assignment to each session) has been successfully validated against the result obtained when executing the centralized version with the same input data. We have designed three different experiments, that we call *Experiment 1*, *Experiment 2* and *Experiment 3*.

In Experiment 1 we evaluate the behavior of B-Neck when many sessions arrive simultaneously. In this experiment a different number of sessions (from 10 to 300,000 sessions) join the network during the first millisecond of the simulation. The joining time of a session has been also chosen uniformly at random in the first millisecond of the simulation. We have run simulations using Small, Medium and Big networks configured in both LAN and WAN scenarios. In this experiment, we are interested in the time B-Neck requires to complete (i.e. to become quiescent) and the traffic it generates.

Figure 5 (left side) presents the time needed to reach quiescence in Experiment 1 (Observe that both axes are in logarithmic scale). It can be observed, in the LAN scenarios, that with a relatively small number of sessions (up to 100 for the Small network and up to 1,000 in Medium and Big networks) the time that B-Neck needs to calculate the max-min fair assignment for all sessions is almost negligible. This is because a small number of sessions in the network cause limited mutual interaction, and therefore, the B-Neck calculation process is simple. Hence, when the first millisecond completes, most of the sessions already have their max-min fair rate assigned. Only sessions that joined the network near the end of the first millisecond still have to be assigned a max-min fair rate, which can be done a few hundreds of microseconds later. However, when the number of sessions is large enough, the interaction among them is much higher, and so, the time B-Neck requires to complete grows roughly linearly with the number of participant sessions. In the WAN scenarios, the times to quiescence shown in Figure 5 are more linear than in the LAN scenarios. This is due to the predominance of larger propagation times, that produce round trip times of 40 milliseconds (on average) on each Probe cycle. In this case even a small dependence among sessions can delay the quiescence of B-Neck significantly. E.g., only 3 Probe cycles in a session delay the quiescence of B-Neck in 120 milliseconds on average. We have observed that the long round trip time of Probe cycles in the WAN scenarios reduces the number of Probe cycles required (because many Update packets from other sessions are received in the middle of a Probe cycle, and so, they are deferred until the Probe cycle finish).

In Figure 5 (right side) we observe that the number of packets transmitted in the network grows roughly linearly with the number of sessions. In this figure every packet sent across a link is accounted for, i.e., a Probe cycle of session s generates a number of packets that is twice the length of s's path. The size of the network has no uniform impact in the number of packets as the number of sessions changes. Conversely, each LAN scenario systematically produces more packets than the equivalent WAN scenario, caused by the smaller number of Probe cycles of the latter. However, the differences observed are less than one order of magnitude.

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. For instance, in the LAN scenarios, even with 300,000 sessions it completes in less than 1 second, while in the WAN scenarios it completes in the worst case in around 10 seconds. Regarding the number of transmitted packets, the average number of packets per session is less than 1,000 packets.

In Experiment 2, we explore the stability of B-Neck in front of a highly dynamic system. We consider LAN scenarios and a Medium network where sessions join, leave, and change their rates. The results are presented in Figure 6, which shows the packets of each type transmitted aggregated in time intervals of 5 milliseconds. The experiment starts with a first join phase in which 100,000 sessions join uniformly at random in 1 ms. It can be seen that B-Neck becomes quiescent in 55 ms. Then in a second phase, 20,000 sessions leave the system during the first millisecond

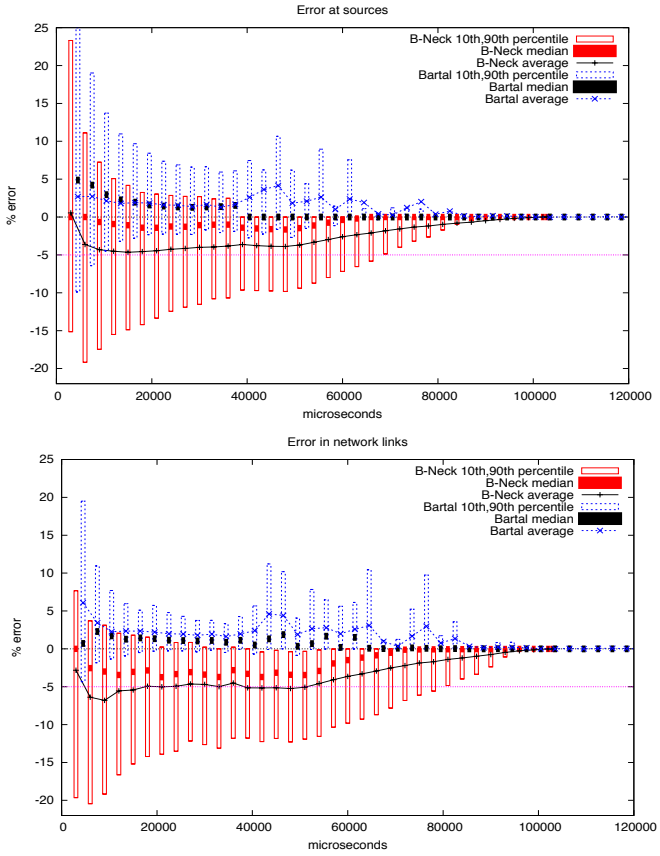


Figure 7: Error distribution in Experiment 3.

of the phase. B-Neck becomes quiescent in 35 ms. later. After that, in the third phase 20,000 sessions change their maximum rate during the first millisecond of the phase. B-Neck converges 40 ms. later. Next, in a fourth phase, 20,000 sessions join the network during the first millisecond of the phase. In this phase, B-Neck completes in 60 ms. (this value is bigger than the values of the second and third phases because B-Neck must calculate again the max-min fair rates of 100,000 sessions instead of 80,000). Finally, in a fifth phase, 20,000 sessions join, 20,000 sessions leave, and 20,000 sessions change their rates, during the first millisecond of the phase. B-Neck becomes quiescent in this phase after 55 ms.

In view of these results we can conclude that B-Neck performance in terms of time to quiescence, is nearly independent of the kind of session dynamics (joins, leaves and rate changes) and the temporal order when changes are produced.

In Experiment 3, we try to compare B-Neck performance against three representatives of non-quiescent protocols: BFYZ [6] representing the family of algorithms that need per-session information at each router, CG [10] as an algorithm that only uses constant state at each router, and RCP [11] as an efficient representative of modern congestion controllers without the need of store and process state information for each session. We consider a LAN scenario in the *Medium network*, where 100,000 sessions join the network and 10,000 leave it during the first five milliseconds. We only represent results from BFYZ, because we observed in our simulations that the other two protocols did not converge to the solution in the time allocated when more than 500 sessions were considered. At fixed intervals (3 milliseconds) we evaluate the accuracy of the max-min fair rates assigned by the protocol to each session. In Figure 7 (left side), we present at the end of each interval the distribution of the relative error between the assigned rate value (a) and the max-min fair rate (x) of the sessions as $e = 100\frac{a-x}{x}$. Positive values of e imply that the session has been assigned a value greater than the max-min fair rate, and negative values of e means that a lower value than the

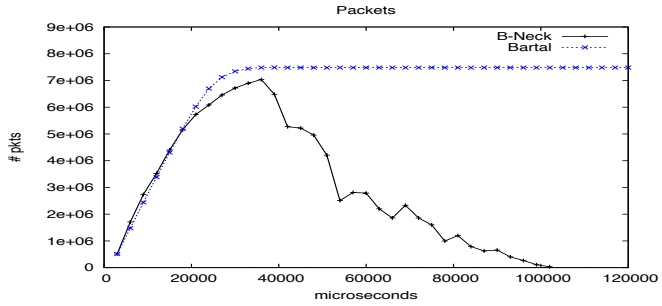


Figure 8: Packets transmitted in Experiment 3.

exact one has been assigned to the session. So, this error gives an idea of the variability that an application using this protocol is going to suffer. In Figure 7 (right side), we present the distribution of the relative error between the sum of the assigned rates (sa) of all the sessions crossing bottleneck links, and the sum of the max-min fair rates (sx) of these sessions as $e = 100\frac{sa-sx}{sx}$. In this case, the error gives an idea of the stress that the bottleneck links are going to suffer. The two figures show that B-Neck always behaves in a conservative way, while BFYZ tends to overload the network, assigning transient rates greater than the max-min fair rates. Hence B-Neck is more *network friendly* than BFYZ. Additionally, the two figures show that B-Neck converges faster than BFYZ (110 milliseconds vs 230 milliseconds), although, in a practical sense, BFYZ reaches rates that are nearly the max-min fair rates in a similar time.

Finally in Figure 8 we plot the number of packets transmitted in each interval by B-Neck and BFYZ. It can be observed that B-Neck, in the worst case (around the 20th millisecond, when sessions have not converged yet to the max-min fair rate), injects to the network the same number of packets than BFYZ. However, as soon as sessions converge to their max-min fair rates, they stop injecting packets to the network and the total traffic generated by B-Neck decreases dramatically. Finally, no packet is injected when all the sessions have converged (around the 110th millisecond). However, BFYZ keeps injecting the same number of packets (around $7 \cdot 10^6$ packets in each interval) even when convergence is reached (because BFYZ does not know that sessions are on the convergence point).

In this experiment we can conclude that (a) B-Neck has an application and network friendly behavior, and (b) it converges strictly faster than BFYZ.

References

[1] Yehuda Afek, Yishay Mansour, and Zvi Ostfeld. Convergence complexity of optimistic rate-based flow-control algorithms. *J. Algorithms*, 30(1):106–143, 1999.

[2] Yehuda Afek, Yishay Mansour, and Zvi Ostfeld. Phantom: a simple and effective flow control scheme. *Computer Networks*, 32(3):277–305, 2000.

[3] Miriam Allalouf and Yuval Shavitt. Centralized and distributed algorithms for routing and weighted max-min fair bandwidth allocation. *IEEE/ACM Trans. Netw.*, 16(5):1015–1024, 2008.

[4] Baruch Awerbuch and Yuval Shavitt. Converging to approximated max-min flow fairness in logarithmic time. In *INFOCOM*, pages 1350–1357, 1998.

[5] Yair Bartal, John W. Byers, and Danny Raz. Global optimization using local information with applications to flow control. In *FOCS*, pages 303–312, 1997.

[6] Yair Bartal, Martin Farach-Colton, Shibu Yooseph, and Lisa Zhang. Fast, fair and frugal bandwidth allocation in atm networks. *Algorithmica*, 33(3):272–286, 2002.

[7] Dimitri Bertsekas and Robert G. Gallager. *Data Networks (2nd Edition)*. Prentice Hall, 1992.

[8] Zhiruo Cao and Ellen W. Zegura. Utility max-min: An application-oriented bandwidth allocation scheme. In *INFOCOM*, pages 793–801, 1999.

[9] Anna Charny, David Clark, and Raj Jain. Congestion control with explicit rate indication. In *International Conference on Communications, ICC'95, vol. 3*, pages 1954 – 1963, 1995.

[10] Jorge Arturo Cobb and Mohamed G. Gouda. Stabilization of max-min fair networks without per-flow state. In Sandeep S. Kulkarni and André Schiper, editors, *SSS*, volume 5340 of *Lecture Notes in Computer Science*, pages 156–172. Springer, 2008.

[11] Nandita Dukkipati, Masayoshi Kobayashi, Rui Zhang-Shen, and Nick McKeown. Processor sharing flows in the internet. In Hermann de Meer and Nina T. Bhatti, editors, *IWQoS*, volume 3552 of *Lecture Notes in Computer Science*, pages 271–285. Springer, 2005.

[12] Ellen L. Hahne and Robert G. Gallager. Round robin scheduling for fair flow control in data communication networks. In *IEEE International Conference in Communications, ICC'86*, pages 103–107, 1986.

[13] Yiwei Thomas Hou, Henry H.-Y. Tzeng, and Shivendra S. Panwar. A generalized max-min rate allocation policy and its distributed implementation using abr flow control mechanism. In *INFOCOM*, pages 1366–1375, 1998.

[14] Xiao Long Huang and Brahim Bensaou. On max-min fairness and scheduling in wireless ad-hoc networks: analytical framework and implementation. In *Proceedings of the 2nd ACM international symposium on Mobile ad hoc networking & computing*, MobiHoc '01, pages 221–231, New York, NY, USA, 2001. ACM.

[15] S. Jain and D. Loguinov. Piqi-rcp: Design and analysis of rate-based explicit congestion control. In *Fifteenth IEEE International Workshop on Quality of Service, IWQoS 2007*, pages 10 –20, June 2007.

[16] Márk Jelasity, Alberto Montresor, Gian Paolo Jesi, and Spyros Voulgaris. The Peersim simulator. `http://peersim.sf.net`.

[17] Dina Katabi, Mark Handley, and Charles E. Rohrs. Congestion control for high bandwidth-delay product networks. In *SIGCOMM*, pages 89–102. ACM, 2002.

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

[19] Hyang-Won Lee, Jeong woo Cho, and Song Chong. Distributed max-min flow control for multi-rate overlay multicast. *Computer Networks*, 54(11):1727–1738, 2010.

[20] Dritan Nace and Michal Pióro. 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, 2008.

[21] Bozidar Radunovic and Jean-Yves Le Boudec. A unified framework for max-min and min-max fairness with applications. *IEEE/ACM Trans. Netw.*, 15(5):1073–1083, 2007.

[22] Dan Rubenstein, Jim Kurose, and Don Towsley. The impact of multicast layering on network fairness. *IEEE/ACM Trans. Netw.*, 10:169–182, April 2002.

[23] Leandros Tassiulas and Saswati Sarkar. Maxmin fair scheduling in wireless networks. In *INFOCOM*, 2002.

[24] Wei Kang Tsai and Yuseok Kim. Re-examining maxmin protocols: A fundamental study on convergence, complexity, variations, and performance. In *INFOCOM*, pages 811–818, 1999.

[25] Hong-Yi Tzeng and Kai-Yeung Sin. On max-min fair congestion control for multicast abr service in atm. *IEEE Journal on Selected Areas in Communications*, 15(3):545 –556, April 1997.

[26] Ellen W. Zegura, Kenneth L. Calvert, and Samrat Bhattacharjee. How to model an internetwork. In *INFOCOM*, pages 594–602, 1996.

[27] Yueping Zhang, Derek Leonard, and Dmitri Loguinov. Jetmax: Scalable max-min congestion control for high-speed heterogeneous networks. *Computer Networks*, 52(6):1193–1219, 2008.