B-Neck: A Distributed and Quiescent Max-min Fair Algorithm

Alberto Mozo
Dpto. Arquitectura y Tecnología de Computadores
U. Politécnica de Madrid
Madrid, Spain
Email: amozo@eui.upm.es

José Luis López-Presa
DIATEL
U. Politécnica de Madrid
Madrid, Spain
Email: jllopez@diatel.upm.es

Antonio Fernández Anta
Institute IMDEA Networks
Madrid, Spain
Email: antonio.fernandez@imdea.org

***Abstract*—The problem of fairly distributing the capacity of a network 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 goal is to maximize its assigned transmission rate (i.e., its throughput). Since the links of the network have limited bandwidths, some criterion has to be defined to fairly distribute their capacity 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 data being continuously transmitted to recompute the max-min fair rates when needed (because none of them has mechanisms to detect convergence to the max-min fair rates).**

In this paper we propose B-Neck, a distributed max-min fair 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. Quiescence is a key design concept of B-Neck, because B-Neck routers are capable of detecting and notifying changes in the convergence conditions of max-min fair rates. As far as we know, B-Neck is the first distributed max-min fair algorithm that does not require a continuous injection of control traffic to compute the rates. 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.

***Index Terms*—Max-mix fairness, distributed algorithm, quiescence.**

I. 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, some criterion to fairly distribute the network resources among the sessions must be used. A popular fairness criterion to share the available network capacity among a set of sessions, without overloading the links, is the, so called, *max-min fairness* [6], [18]. The 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 all sessions are guaranteed a certain minimum quality of service. 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, both centralized and distributed, have been proposed (see Section I-A). However, to our knowledge, all the distributed algorithms proposed require control data being continuously transmitted to recompute the max-min fair rates even if the set of sessions do not change (like sessions arriving or leaving). In this paper we propose a distributed algorithm, which we call B-Neck, that is *quiescent*, i.e., once the max-min fair rates have been computed, B-Neck 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.

A. Related Work

We focus on computing the max-min fair rate allocation for single path sessions. These max-min fair rates can be efficiently computed in a centralized way with the Water-Filling algorithm [6], [18]. Max-min fairness has usually been chosen as the target fairness criterion implemented by congestion control protocols to divide 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 congestion control protocols. Another classification of max-min fair algorithms is based on considering whether per-session state information is needed in routers or, otherwise, only a constant amount of information is used.

To our knowledge, the proposals of Gallager [11] and Katevenis [16] were the first to apply max-min fairness to share bandwidth among sessions in a packet switched network. They achieved fairness by allocating, at each router link,

978-0-7695-4489-2/11 $26.00 © 2011 IEEE
DOI 10.1109/NCA.2011.10

one queue per session, and using a round robin scheduler. However, no max-min fair rate was explicitly calculated. 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], [5], [8], [7], [12], [19], [20]. These algorithms calculate the max-min fair rates using the ATM special Resource Management (RM) cells, and so, router links are in charge of executing the max-min fair algorithm. Charny et al. [8] seem to have been the first to analytically prove the correctness of their proposed algorithm. Hou et al. [12] generalized this algorithm to extend the max-min fairness criterion with minimum rate requests and peak rate constraints. A problem of the algorithm in [8] (when pseudo-saturated links appear) was identified and documented by Tsai and Kim [19]. It is worth noting that all the distributed algorithms mentioned above need per-session state information at the routers. A distributed algorithm that only uses constant state information in each router to exactly compute the max-min fair rates has been proposed in [9], but it requires strong synchronous behavior. Several max-min fair algorithms have been proposed that compute an *approximation* of the rates [4], [3], [2]. Unfortunately, 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].

Recent research trends in explicit congestion control protocols (XCP [15], RCP [10], PIQI-RCP [13]) implement efficient congestion controllers in routers. These controllers do not need storing and processing state information for each session, and guarantee that the max-min fair rate assignments are achieved when controllers are in steady state.

In any case, none of the former algorithms are quiescent, and so, control data must be continuously injected into the network to keep the system stable. It is not straightforward to transform any of these algorithms to achieve quiescence.

B. Contributions

In this paper we propose B-Neck, the first max-min fair distributed algorithm that is also quiescent. Instead of requiring a continuous injection of control data to compute the max-min fair rates, B-Neck uses a limited number of control packets. Quiescence is a key design feature of B-Neck. Routers are provided with the capability of detecting changes in the convergence conditions of session rates (from instability to stability and vice-versa), so that they can notify the affected sessions of these changes. In case of session stability, the session informs the routers in its path of this fact and becomes quiescent. Otherwise, the session restarts the computation of its fair rate. This behavior is not present in any of the non-quiescent published algorithms and, as mentioned, making any of these algorithms quiescent is not a trivial task.

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 limit the maximum rate that it requires, both at the time it is created (with $API.Join$) and at any other time by using a specific primitive (namely, $API.Change$).

The properties of B-Neck have been formally proved. This proof has two parts. Firstly, its *correctness* is shown, 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, *quiescence* is shown, i.e., after computing the rates, B-Neck eventually 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 B-Neck eventually becomes quiescent again.

B-Neck has been tested with extensive simulations. In them, we have used networks of several sizes (with up to hundreds of thousands of nodes), with LAN and WAN properties, and with a wide range of session cardinalities (up to hundreds of thousands). 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 [6], [18]). B-Neck has always converged to the correct set of max-min fair rates. Our simulations have shown that B-Neck converges very quickly, even in the presence of many interacting sessions. We have also stressed the algorithm by, once quiescent, causing a large number of simultaneous departures and rate changes. In all cases, B-Neck has shown to be robust and efficient, quickly reaching convergence and quiescence again. The control traffic caused in the network by the algorithm is limited, and only for highly dynamic systems, with many sessions, has more than a few packets per session. Finally, comparing B-Neck with some non-quiescent protocols, we have observed that it converges faster and, unlike these other protocols, until convergence, B-Neck assigns transient rates, to the sessions, that are smaller than the max-min fair rates. Hence, due to these conservative transient rate assignments, it is expected that the network links will not suffer from packet overloading before convergence.

C. Structure of the Rest of the Paper

The rest of the paper is structured as follows. In Section II definitions and notations are provided. In Section III the max-min fair protocol B-Neck is presented, and its correctness is sketched. Finally, Section IV presents experimental results.

II. 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, hosts, and 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, called the *source node* of the session, and ends in another host, called 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 of s (path of $\pi(s)$ that traverses the same sequence of nodes in reverse order). 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) is 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, i.e., 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. S denotes the set of all active sessions in the system, and S_e the set of sessions in S that cross link e. For each $s \in S$, r_s denotes the maximum rate requested by s. The max-min fair rates 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 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.

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

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 [6].

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., $\sum_{s \in S_e} \lambda^*_s < C_e$).

III. 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.

A. Centralized B-Neck

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 [6], and guarantees that $R_e = R^*_e$ and $F_e = F^*_e$.

As we mentioned, B-Neck follows a logic similar to that of the Centralized B-Neck algorithm. From the point of view of sessions, the algorithm works as follows. For each session, the bottleneck rates of every link in the session's path are computed. The smallest such rate (called the bottleneck rate of the session) is assigned to the session. Doing this in increasing order of the bottleneck rate of the sessions yields the solution.

```
1  task RouterLink (e)
2  var R_e ← ∅; F_e ← ∅
3
4  procedure ProcessNewRestricted()
5      while ∃s ∈ F_e : λ_s^e ≥ B_e do
6          λ_m ← max_{s∈F_e}{λ_s^e}
7          R' ← {r ∈ F_e : λ_r^e = λ_m}
8          F_e ← F_e \ R'; R_e ← R_e ∪ R'
9      foreach s ∈ R_e : μ_s^e = IDLE ∧ λ_s^e > B_e do
10         μ_s^e ← WAITING_PROBE; send upstream Update (s)
11
12 when received Join (s, λ, η) do
13     R_e ← R_e ∪ {s}; μ_s^e ← WAITING_RESPONSE
14     ProcessNewRestricted()
15     if λ > B_e then λ ← B_e; η ← e
16     send downstream Join (s, λ, η)
17
18 when received Response (s, τ, λ, η) do
19     if τ = UPDATE then μ_s^e ← WAITING_PROBE
20     else
21         if ((η = e ∧ λ = B_e) ∨ (η ≠ e ∧ λ ≤ B_e)) then
22             μ_s^e ← IDLE; λ_s^e ← λ
23         else // ((η = e ∧ λ < B_e) ∨ (λ > B_e))
24             τ ← UPDATE ; μ_s^e ← WAITING_PROBE
25         if ∀r ∈ R_e, λ_r^e = B_e ∧ μ_r^e = IDLE then
26             τ ← BOTTLENECK;  η ← e
27             foreach r ∈ R_e \ {s} do send upstream Bottleneck (r)
28     send upstream Response (s, τ, λ, η)
29
30 when received Probe (s, λ, η) do
31     μ_s^r ← WAITING_RESPONSE
32     if s ∈ F_e then
33         F_e ← F_e \ {s}; R_e ← R_e ∪ {s}
34         ProcessNewRestricted()
35     if λ > B_e then λ ← B_e; η ← e
36     send downstream Probe (s, λ, η)
37
38 when received Update (s) do
39     if μ_s^e = IDLE then
40         μ_s^e ← WAITING_PROBE; send upstream Update (s)
41
42 when received Bottleneck (s) do
43     if μ_s^e = IDLE ∧ s ∈ R_e then send upstream Bottleneck (s)
44
45 when received SetBottleneck (s, β) do
46     if ∀r ∈ R_e, λ_r^e = B_e ∧ μ_r^e = IDLE then
47         send downstream SetBottleneck (s, TRUE)
48     else if λ_s^e < B_e ∧ μ_s^e = IDLE then
49         R' ← {r ∈ R_e : μ_r^e = IDLE ∧ λ_r^e = B_e}
50         foreach r ∈ R' do
51             μ_r^e ← WAITING_PROBE;  send upstream Update (r)
52         R_e ← R_e \ {s}; F_e ← F_e ∪ {s}
53         send downstream SetBottleneck (s, β)
54     else if μ_s^e = IDLE ∧ λ_s^e = B_e then
55         send downstream SetBottleneck (s, β)
56
57 when received Leave (s)
58     R' ← {r ∈ R_e \ {s} : μ_r^e = IDLE ∧ λ_r^e = B_e}
59     if s ∈ F_e then F_e ← F_e \ {s} else R_e ← R_e \ {s}
60     foreach r ∈ R' do
61         μ_r^e ← WAITING_PROBE; send upstream Update (r)
62     send downstream Leave (s)
```

Figure 2. Task Router Link (RL).

B. 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

```
1  task SourceNode (s, e)
2
3  when API.Join(s, r) do
4      R_e ← {s}; D_s ← min(r, C_e); μ_s^e ← WAITING_RESPONSE
5      upd_rcv_s ← FALSE; bneck_rcv_s ← FALSE
6          send downstream Join (s, D_s, e)
7
8  when API.Leave(s) do
9      F_e ← ∅; R_e ← ∅; send downstream Leave (s)
10
11 when API.Change(s, r) do
12     D_s ← min(r, C_e)
13     if μ_s^e = IDLE then
14         if s ∈ F_e then F_e ← ∅; R_e ← {s}
15         upd_rcv_s ← FALSE; bneck_rcv_s ← FALSE
16         μ_s^e ← WAITING_RESPONSE
17         send downstream Probe (s, D_s, e)
18     else  upd_rcv_s ← TRUE
19
20 when received Update (s) do
21     if μ_s^e = IDLE then
22         if s ∈ F_e then F_e ← ∅; R_e ← {s}
23         bneck_rcv_s ← FALSE; μ_s^e ← WAITING_RESPONSE
24         send downstream Probe (s, D_s, e)
25     else  upd_rcv_s ← TRUE
26
27 when received Bottleneck (s) do
28     if μ_s^e = IDLE ∧ ¬bneck_rcv_s then
29         bneck_rcv_s ← TRUE;  API.Rate (s, λ_s)
30         if D_s > λ_s then F_e ← {s}; R_e ← ∅
31         send downstream SetBottleneck (s, D_s = λ_s)
32
33 when received Response (s, τ, λ, η) do
34     if τ = UPDATE ∨ upd_rcv_s then
35         upd_rcv_s ← FALSE; bneck_rcv_s ← FALSE
36         μ_s^e ← WAITING_RESPONSE
37         send downstream Probe (s, D_s, e)
38     else if τ = BOTTLENECK then
39         λ_s^e ← λ ;  μ_s^e ← IDLE; bneck_rcv_s ← TRUE
40         API.Rate (s, λ_s^e)
41         if D_s > λ_s then F_e ← {s}; R_e ← ∅
42         send downstream SetBottleneck (s, D_s = λ_s)
43     else // τ = RESPONSE
44         λ_s^e ← λ;  μ_s^e ← IDLE
45         if D_s = λ_s then
46             bneck_rcv_s ← TRUE;  API.Rate (s, λ_s^e)
47             send downstream SetBottleneck (s, TRUE)
```

Figure 3. Task Source Node (SN).

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

Figure 4. Task Destination Node (DN).

estimated bottleneck rate of the session, and η is the link with the smallest bottleneck rate found in the 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 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.

C. A global perspective of B-Neck

B-Neck is formally specified as three asynchronous tasks 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.

Like most max-min fair distributed algorithms, B-Neck keeps per-session information at each link e. 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. Besides, for each session s, it is stored its state $\mu_s^e \in \{\text{IDLE, WAITING\_PROBE, WAITING\_RESPONSE}\}$ and its assigned rate λ_s^e. 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.

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.

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 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). When sessions stop joining, leaving or changing their rate requirement, the network will eventually get stable, and no more traffic will be generated until there is another change in the sessions configuration.

The Response packets that close Probe cycles are used, at the links, to detect bottleneck conditions, and to assign rates to the sessions. 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. Then, all these sessions 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 source node receives the Bottleneck communication, it sends a SetBottleneck packet to inform the links in its session's 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. 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, bottlenecks may be discovered in parallel, which speeds up convergence. However, sometimes, due to this parallelism, 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 in the affected sessions, and the first bottleneck l will eventually be correctly identified.

D. Correctness of B-Neck

The details of the correctness proof are omitted due to space limitations. The complete proof can be found in [17]. However, we give a rough idea of the proof structure. Let us first define network stability.

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).

The key of the proof resides mainly in proving the following two lemmas. Observe that permanent stability implies quiescence.

Lemma 1: 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.

Lemma 2: Consider a network executing the B-Neck protocol, and a time at which the network is stable. Then, for each session i, every link $e \in \pi(i)$ has $\lambda_i^e = \lambda_i^*$.

Then, the main theorem derives directly from these lemmas.

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.

IV. 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 (Peersim [14]), modified to be able to run B-Neck with thousands of routers and up to a million hosts and sessions, and to model 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 [21]. 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. 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. 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. 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.

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.

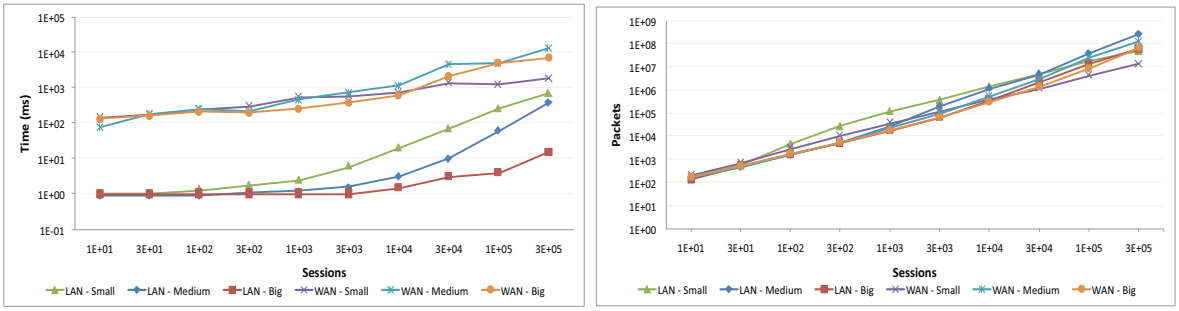

Figure 5. Time until quiescence and number of packets observed in Experiment 1.

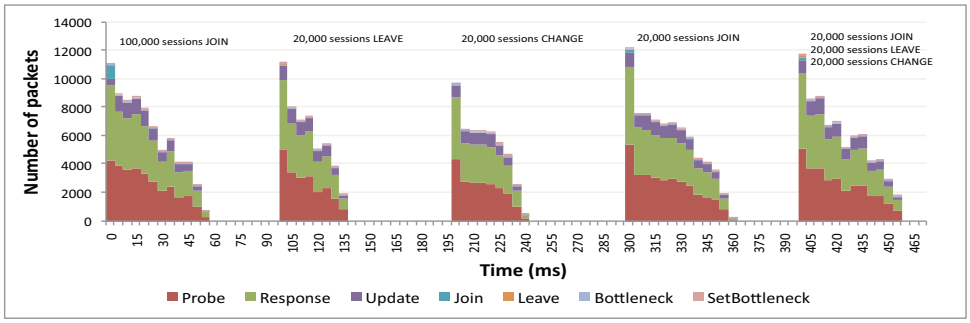

Figure 6. Traffic details of B-Neck in Experiment 2.

In this experiment we can conclude that B-Neck behaves rather efficiently both in terms of time to quiescence, and in terms of average number of packets per session.

In *Experiment 2*, we explore the stability of B-Neck against 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 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 of the phase. B-Neck becomes quiescent 35 ms. later. In a third phase $20,000$ sessions change their maximum rate during the first millisecond. B-Neck converges 40 ms. later. Next, in a fourth phase, $20,000$ sessions join the network during the first millisecond. In this phase, B-Neck completes in 60 ms. This phase takes longer to converge because B-Neck must recompute 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. 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 [5] representing the family of algorithms that need per-session information at each router, CG [9] as an algorithm that only uses constant state at each router, and RCP [10] 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 (a) and the max-min fair rate (x) of the sessions as $e = 100\frac{a-x}{x}$. Positive values of e show that the session's rate has been overestimated, and negative values show that the session's rate has been underestimated. Hence, 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, overestimating 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

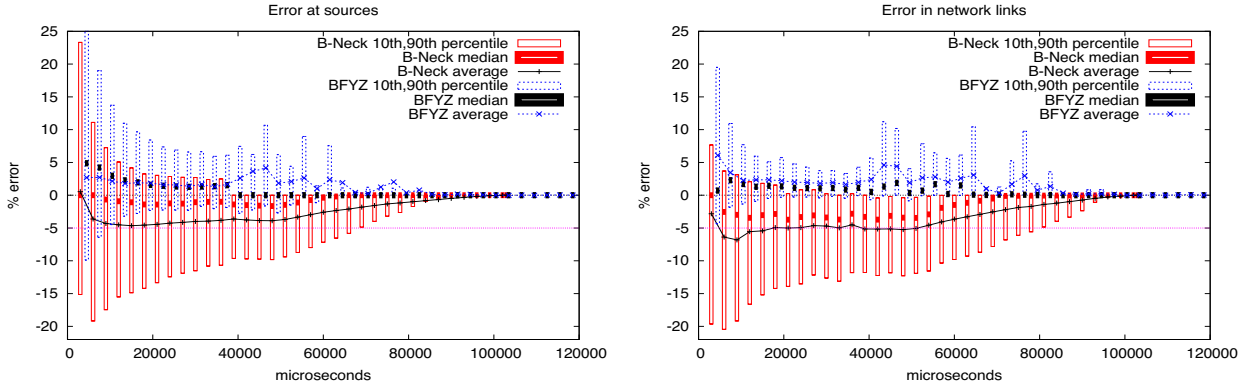

Figure 7. Error distribution in Experiment 3.

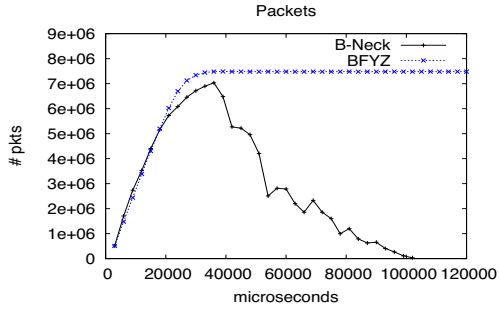

Figure 8. Packets transmitted in Experiment 3.

that B-Neck, in the worst case (around the 20th millisecond, when sessions have not converged yet to the max-min fair rate), injects the as many packets as BFYZ. However, as soon as sessions converge to their max-min fair rates, they become quiescent, 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 per interval) even when convergence is reached (since BFYZ can not detect convergence).

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.

Acknowledgements

This research was partially funded by the Comunidad de Madrid grant S2009TIC-1692 and Spanish MICINN grant TIN2008–06735-C02-01. The authors would like to thank Araceli Lorenzo, Andrés Sevilla, and Pilar Manzano for useful discussions, and Miguel Ángel Hernández for helping with the simulations.

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

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

[5] 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.

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

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

[16] 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.

[17] Alberto Mozo, Jose Luis López-Presa, and Antonio Fernández Anta. B-Neck: A distributed and quiescent max-min fair algorithm. Technical Report TR-IMDEA-Networks-2011-2, Institute IMDEA Networks, Madrid, Spain, April 2011.

[18] 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.

[19] 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.

[20] 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.

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