Theoretical Computer Science 412 (2011) 5562–5579

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

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

Stabilization of max–min fair networks without per-flow state

Jorge A. Cobb [a,*], Mohamed G. Gouda [b]

[a] Department of Computer Science, The University of Texas at Dallas, Richardson, TX 75083-0688, United States

[b] Department of Computer Science, The University of Texas at Austin, Austin, TX 78712-0233, United States

ARTICLE INFO

Keywords:
Stabilization
Max–min fairness
Quality of service
Computer networks

ABSTRACT

Let a *flow* be a sequence of packets that are sent from a source computer to a destination computer. In this paper, we consider the fair allocation of bandwidth to each flow in a computer network. We focus on max–min fairness, which assigns to each flow the largest possible bandwidth that avoids affecting other flows. What distinguishes our approach is that routers only maintain a constant amount of state, i.e., no per-flow state is maintained. This is consistent with trends in the Internet (such as the proposed Differentiated Services Internet architecture). In addition, to provide a high degree of fault-tolerance, we ensure our approach is self-stabilizing, that is, it returns to a normal operating state after a finite sequence of faults.

1. Introduction

To provide best-effort service, routers in the Internet do not need to maintain any state information about the flows of packets that traverse them. On the other hand, to provide more advanced services, maintaining state for each individual flow simplifies the design of protocols that support these services. However, as the Internet grows, scalability at the core of the Internet has become a concern. Thus, the Differentiated Services architecture [1,2], which maintains only a constant amount of state per router, is favored over the Integrated Services architecture [3,4], where each router maintains state for each individual flow.

We focus on providing fair bandwidth allocation among the flows in a core network. Many different notions of fairness exist [5], and each of these leads to a different optimization objective. We choose the notion of *max–min fairness* [6], which, intuitively, can be described as follows. An allocation of bandwidth to each flow is max–min fair if increasing the bandwidth of any flow can only be done at the expense of decreasing the bandwidth of another flow with equal or lesser bandwidth.[1]

Max–min fairness satisfies many intuitive fairness properties, and it has been studied extensively [7–10]. However, all of these proposed algorithms need per-flow state.

In this paper, we present a fault-tolerant and distributed algorithm for computing the max–min bandwidth allocation in a computer network. Our algorithm only requires a constant amount of state information at each router. Although constant-state algorithms have been presented earlier [11,12], they have disregarded fault-tolerance altogether. Our algorithm is presented formally and is shown to be stabilizing, i.e., it is resilient against a wide variety of transient faults.

The organization of this paper is as follows. Section 2 presents related work. Section 3 describes our notation and defines stabilization. Our network model and the definition of max–min fairness are given in Section 4. Then, in Section 5, we present a general signaling protocol that is stabilizing and provides constant state at each router. Our stabilizing max–min signaling protocol is then presented in Section 6. Finally, concluding remarks are given in Section 7.

* Corresponding author. Tel.: +1 972 883 2479; fax: +1 972 883 2349.
E-mail addresses: cobb@utdallas.edu (J.A. Cobb), gouda@cs.utexas.edu (M.G. Gouda).

[1] See Section 4 for a detailed definition.

0304-3975/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2010.11.042

2. Related work

Max–min fairness captures what most users would intuitively consider to be a fair distribution of bandwidth. However, sometimes fairness is achieved by sacrificing throughput; i.e., a maximum throughput allocation is not always max–min fair [6].[2] Nonetheless, max–min fairness is preferred over maximum throughput since it provides a greater degree of user satisfaction. Other methods, such as proportional fairness and balanced fairness [5], attempt to provide a balance between user satisfaction and overall network throughput.

Max–min fairness has been studied extensively in the literature. For the case of general-purpose networks, distributed algorithms to compute max–min fairness have been presented in [8,13]. In addition, max–min fairness has been studied in a variety of network conditions, such as links having variable rate [7], when there are restrictions on the minimum and maximum bandwidth of each flow [9], and in a multicast environment [10].

In additional to general-purpose networks, max–min fairness has also been applied to many classes of networks. This includes optical networks [14,15], ad hoc networks [16,17], sensor networks [18,19], network-on-chip architectures [20], and aggregated links for future Internet architectures [21].

All the algorithms above maintain per-flow state at each switch or router. As mentioned above, our focus is on maintaining as little state as possible in the router, in particular, a constant amount of state. In addition, we require our protocol to be stabilizing, and thus to be resilient to all forms of transient faults. Other constant-state algorithms have been presented earlier [11,12]; however, fault-tolerance was not addressed.

3. Notation and stabilization

A *system* consists of a set of processes, and a set of communication channels between these processes. The *topology* of the system consists of a connected undirected graph, where each node represents one process in the system, and each edge between two nodes p and q indicates that processes p and q are neighbors in the system. Neighboring processes are joined by a pair of communication channels allowing them to exchange messages.

Each *process* in a system is specified by finite sets of constants, variables, and actions. To distinguish variables with the same name but in different processes, we refer to variable v in process x as $x.v$.

Each process is assumed to have access to a real-time clock. Clock values need not be synchronized between processes. The only requirement is that clocks of different processes advance at (approximately) the same rate.

Each action of a process p is of the form

$$\langle \text{guard} \rangle \rightarrow \langle \text{assignment} \rangle$$

where ⟨guard⟩ is a boolean expression in one of three forms: (i) local, (ii) receiving, or (iii) timeout, as follows.

A local guard is a boolean expression over the constants and variables of process p. A receiving guard of the form **rcv** m evaluates to true if there is a message of type m in one of the incoming channels of p. Finally, a timeout action is a boolean expression involving the clock of the process.

In the above action, ⟨assignment⟩ is a sequence of assignment statements, each of which is of the form

$$x := \mathrm{E}(y, \ldots) \textbf{ if } P$$

where x is a variable in process p, E is an expression of the same type as variable x, and y is a either a constant or a variable in process p. Executing this assignment statement assigns the value of expression E to variable x provided predicate P is true. Otherwise, the value of x is left unchanged.

A *state* of a system S is specified by one value for each variable, taken from the domain of values of that variable, in each process in S, and the contents of each communication channel in S.

A *transition* of a system S is a triple of the form

$$(s, ac, s')$$

where s and s' are two states of system S, and ac is an action in some process in S such that the following two conditions hold.

i. *Enablement:* The guard of action ac is true at state s.
ii. *Execution:* Executing the assignment of action ac, when system S is in state s, yields system S in state s'.

A *computation* of a system S is a sequence of the form

$$(s_0, ac_0, s_1),\ (s_1, ac_1, s_2),\ \ldots$$

where each element $(s_i, ac_i, s_{(i+1)})$ is a transition of S such that the following two conditions hold.

[2] An example is given in Section 4.

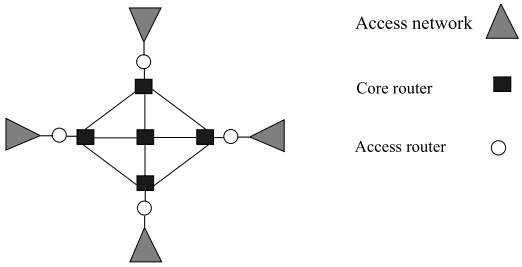

Fig. 1. Core network.

i. *Maximality:* Either the sequence is infinite, or its last element $(s_{(z-1)}, ac_{(z-1)}, s_z)$ is such that the guard of every action in system S is false at state s_z, and timeout actions cannot evaluate to true by increasing the value of the clocks in the system.
ii. *Fairness:* If the sequence has an element $(s_i, ac_i, s_{(i+1)})$ and the guard of some action ac is true at state $s_{(i+1)}$, then the sequence has a later element $(s_k, ac_k, s_{(k+1)})$ where ac_k is ac or the guard of ac is false at state $s_{(k+1)}$.

A *predicate* P of a system S is a boolean expression over the variables in all processes in system S and the contents of the channels in S.

A system S is called *P-stabilizing* iff every computation of S has a suffix where P is true at every state of the suffix [22–24].

Stabilization is a strong form of fault-tolerance. Normal behavior of the system is defined by predicate P. If a fault causes the system to a reach an abnormal state, i.e., a state where P is false, then the system will converge to a normal state where P is true, and remain in the set of normal states as long as the execution remains fault-free.

4. Network model

In this section, we present a description of the various components in a core computer network, and show how these components map to processes in our formal model. We then present the definition of max–min fairness, and conclude with an example.

4.1. Network components

Consider a computer network as depicted in Fig. 1. It consists of a set of core routers surrounded by access networks. Access routers serve as intermediate points between the core network and the access networks.

Consider a computer in an access network that generates data packets that must cross the core network to reach their destination at a different access network. We denote this sequence of packets as a *flow*.

As it is commonly assumed [25–28], access routers maintain information about each individual flow, while core routers, for scalability purposes, do not. In our case, core routers will maintain only a constant amount of information regarding the flows that traverse them.

We model this by having three types of processes in our system: source processes, router processes, and destination processes. Each source process corresponds to the actions that an access router must perform for an individual flow. Thus, there are multiple source processes per access router, and each source process is associated with a single destination process at a different access router.

Routers have multiple processes, one per output channel, as shown in Fig. 2(a). Therefore, the path traversed by a flow is abstracted as shown in Fig. 2(b). That is, data begins at a source process, it traverses multiple router processes, and ends at a destination process.

The path across the core network between a source and destination is assumed to be constant, which may be implemented with mechanisms such as MPLS [29]. Route changes across a core network are rare, and thus they are viewed as faults in our system.

We assume sources are greedy, and will use as much bandwidth as the network allows them. In the concluding remarks, we discuss fixed-rate sources and sources with a fixed upper bound on their bandwidth.

Each source probes the network to determine how much bandwidth it is allowed to use. Routers only keep an aggregate (and hence constant) amount of information regarding the flows that traverse them and the bandwidth they consume. Through signaling messages, the sources are able to modify this aggregate information in order to maintain its accuracy and to achieve fairness.

To ensure correct synchronization of values between sources and routers, we require some bounds on the delay of signaling messages. Routers must give signaling messages high priority, ensuring that the end-to-end delay does not exceed ε seconds. Messages exceeding this bound are discarded. This can be accomplished in a variety of ways, including timestamping each message with its inception time, or with the accumulated queuing delay that the packet has encountered along its path. We thus incorporate this assumption on end-to-end delays into our system model.

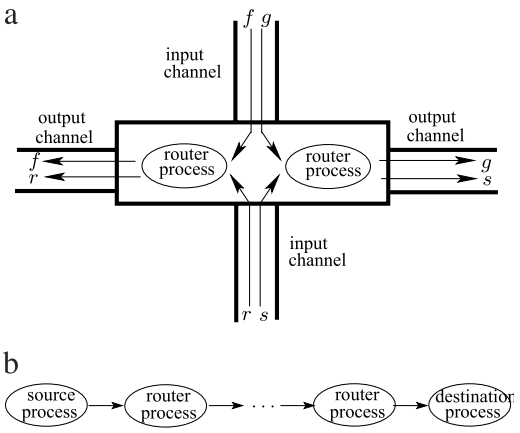

Fig. 2. Processes and flows in a core router.

4.2. Max–min fairness

We conclude by defining our fairness objective. We will consider max–min fairness [6], which is intuitively defined as follows: bandwidth is allocated to each flow f so that any increase of the bandwidth allocated to f must be done at the expense of decreasing the bandwidth of another flow g, where the bandwidth allocated to g is smaller than or equal to that of f.

The max–min bandwidth allocation $f.w$ of each flow f can be defined iteratively as follows. Let:

- $C(p, q)$ be the bandwidth capacity of channel (p, q),
- $T(p, q)$ be the set of flows traversing channel (p, q),
- $f.w$ be zero initially, i.e., no bandwidth is allocated to f.
- For simplicity,

$$G(p, q) = \{f \mid f \in T(p, q) \wedge f.w > 0\}$$
$$U(p, q) = \{f \mid f \in T(p, q) \wedge f.w = 0\}.$$

That is, $G(p, q)$ contains those flows traversing (p, q) whose bandwidth has been assigned, and $U(p, q)$ contains those flows traversing (p, q) whose bandwidth has not been assigned.

The iteration steps are given below. The iterations end when all flows have bandwidth allocated to them.

1. Let (p, q) be a channel such that $U(p, q) \neq \emptyset$ (if no such channel, then exit) and the following expression is minimized:

$$\frac{C(p, q) - \sum_{f \in G(p,q)} f.w}{|U(p, q)|}. \tag{1}$$

2. For every flow f, where $f \in U(p, q)$, assign (1) to $f.w$.
3. Return to step 1.

It is easy to show that each edge is chosen at most once in the above iterations. Also, if m_i is the bandwidth assigned in the ith iteration, then the sequence $m_1, m_2, \ldots,$ is non-decreasing. Finally, if flow f is assigned bandwidth during the iteration for edge (p, q), we say that f is *bottled*, or limited by, edge (p, q).

As a simple example, consider Fig. 3, where we have five routers and five flows. Flow f traverses the entire network, while the remaining flows traverse only a single hop. Assume all links have equal capacity C, except for the link (R_3, R_4), which has capacity $C/2$.

To maximize the throughput of the system, each of flows g_1, g_2 and g_4 must be assigned a bandwidth of C, g_3 must be assigned a bandwidth of $C/2$, while flow f must be assigned a bandwidth of zero, which of course is unfair to f.

Under max–min fairness, at each link, we divide the bandwidth by its number of flows, and find the minimum of these values. This occurs at link (R_3, R_4), with a value of $((C/2)/2) = C/4$, while all other links have a value of $C/2$. Thus, f and g_3 are assigned a bandwidth of $C/4$ each. Also, since f traverses the other three links, their bandwidth is reduced by $C/4$.

We thus have a bandwidth of $3 \cdot C/4$ left at each of the remaining three links. Since each of these has only one flow, then g_1, g_2, and g_4 are assigned a bandwidth of $3 \cdot C/4$.

Finally, throughout the paper, we use the terms bandwidth and data rate interchangeably.

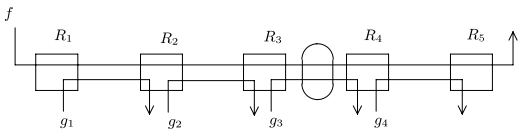

Fig. 3. Max–min fairness example.

5. Signaling

In this section, we present a signaling protocol whose purpose is to allow each router to aggregate information about each of the flows that traverses it. The type of information that is aggregated at the routers is left undetermined. This allows the protocol to be used in a wide variety of contexts, not just for max–min fairness. Its specific application to max–min fairness is presented in Section 6.

Our protocol is an abstraction of the protocol we presented in [30,31]. In addition to being more general, it is strengthened to become stabilizing.

We make the assumption that the set of flows in the network is fixed, or the elapsed time between the addition or removal of flows is large. Otherwise, a flow's fair share of the network bandwidth varies quickly over time and, hence, the system never converges to a fixed bandwidth assignment.

Since the set of flows is fixed, we do not address the steps required to setup/tear-down a flow, and focus only on refreshing/correcting information at the routers. We discuss setup/tear-down of flows in the concluding remarks.

5.1. Shadow state

As mentioned earlier, routers only maintain an aggregate of the values provided by each flow. We assume that the source of flow f has a value f_v, and that there exists an associative and commutative operator, $\oplus$, on the domain of this value. Thus, each router process R maintains the following value:

$$\langle \oplus f \ : \ f \text{ traverses } R \ : \ f_v \rangle.$$

For example, in Section 6, each router process maintains the sum of the bandwidths of the flows that traverse the channel.

The signaling protocol must ensure that the above aggregate is correct, and must recompute the correct value after faults occur. For example, the aggregate has to be recomputed if source processes die, or the path between a source and its destination changes.

To eliminate stale information that was caused by a fault, the router process maintains two variables: the aggregate A, and its "shadow copy" $\widehat{A}$. The router also maintains a boolean bit s, which we refer to as the "shadow bit". Every T seconds, where T is a predefined constant, the router updates its state in the following way:

$$s, A, \widehat{A} := \neg s, \widehat{A}, 0. \tag{2}$$

That is, bit s is flipped, the shadow copy $\widehat{A}$ is assigned to A, and the shadow copy $\widehat{A}$ is reset to the zero value of the $\oplus$ operator.

By performing the above, any value that fails to be added to $\widehat{A}$ within the update period will no longer be part of the aggregate A. Thus, values from terminated (faulty) sources will be removed from the aggregate within $2 \cdot T$ seconds.

However, in order to maintain the information in aggregate A up-to-date, each flow f that traverses the router must add f_v into $\widehat{A}$ *once and only once* each time the update in (2) is performed. In this manner, the router at all times will contain f_v in its aggregate A. Below, we discuss how the source of f can accomplish this.

5.2. Aggregate messages

To keep the information accurate along its path, the source of flow f periodically sends an *Aggregate* signaling message containing f_v. As mentioned in Section 4, the message is sent along the path of f with high priority and bounded round-trip time.

Adding the value of each flow to $\widehat{A}$ exactly once is accomplished as follows. The *Aggregate* message contains a bit vector, $\vec{s}$, with one bit for each router along the path of the flow. The vector contains the values (as known to the source) of the s bits of the routers along the path. The value of the flow is added to the shadow variable $\widehat{A}$ only if the state of the router has been updated (and thus s has changed) from the time of the previous *Aggregate* message of the flow.

In summary, the following two steps are performed at the ith router along the path of f whenever it receives an *Aggregate*$(f, f_v, \vec{s})$ message.

- if $\vec{s}_i \neq s$, then assign $\widehat{A} \oplus f_v$ to $\widehat{A}$, and assign s to $\vec{s}_i$.
- forward the *Aggregate*$(f, f_v, \vec{s})$ message along the next hop to the destination of f.

When the destination receives the *Aggregate* message, it returns an *Ack* message back to the source, containing the updated vector $\vec{s}$. The source will not generate a new *Aggregate* message until the acknowledgment is received for the previous one.

5.3. Timing of aggregate messages

We next address how often the source of a flow should send an *Aggregate* message. As mentioned earlier, we assume a bound, ε, on the time for a signaling message to traverse the network. A signaling message created at time t is discarded by a router if it is received at a time greater than $t + \varepsilon$. State updates of different routers are not required to be synchronized. The only assumption is that each router process performs updates at least T seconds apart.

Consider Fig. 4, and consider a router along the path of flow f. A state update occurs in the router at time t_0, and another at time t_2. At time t_1, the source of f transmits an *Aggregate* message, which arrives at the router in the interval (t_0, t_2). Thus, at least one *Aggregate* message from f must arrive at the router in the interval (t_2, t_3). In the worst case, t_1 is almost equal to t_2, which implies that the next *Aggregate* message must arrive at the router no later than $t_1 + T$, i.e., it must be sent no later than $t_1 + T - \varepsilon$. Furthermore, the next *Aggregate* cannot be sent until an *Ack* is received for the first *Aggregate*, which at the latest will occur at time $t_1 + 2 \cdot \varepsilon$. Thus, we require

$$t_1 + 2 \cdot \varepsilon < t_1 + T - \varepsilon.$$

That is, $3 \cdot \varepsilon < T$, and the interval between successive transmissions of *Aggregate* messages should be at most $T - \varepsilon$.

5.4. Specification of the signaling protocol

To adapt the signaling protocol to the max–min fairness protocol in Section 6, we add the following two generalizations:

(i) The source of a flow f may change its value f_v over time.
(ii) The router may arbitrarily choose whether or not to add the flow to its aggregate.

Consider requirement (i). Because the source of f may change its value f_v, the contribution of f to the aggregate A in a router must be updated to reflect the new value of f_v. Hence, each *Aggregate* message must contain both the old and new values of f_v.

Consider next requirement (ii). Because routers do not maintain per-flow information, they are unaware if a specific flow f is included in their aggregate or not. This requires the source to store this information in the form of a bit vector $\vec{b}$, where $\vec{b}_i$ is true if f_v is included in the aggregate of the i^{th} router along the path of f.

In summary, each *Aggregate* message contains the following fields:

- x : source node id of flow f.
- y : destination node id of flow f (the pair (x, y) uniquely identifies f).
- v : the current value of f_v that has been included in the aggregate at the routers.
- v' : the new value chosen by the source of f.
- $\vec{b}$: bit vector indicating if v is included in the aggregate at each hop.
- $\vec{s}$: bit vector with a copy of the shadow bits of the routers along the path.

We are now ready to present the specification of the source, router, and destination processes. The source process is specified as follows.

```
process source x
const
  y        :  process id     {destination}
  ε        :  integer        {min. inter-packet time}
var
  s⃗        :  bit vector     {s⃗_R = shadow-bit at router R along the path}
  b⃗        :  bit vector     {b⃗_R = true if v is aggregated at router R}
  v        :  data           {data value aggregated at each router}
  v'       :  data           {new data value to be aggregated at each router}
  t        :  integer        {time last msg was sent}
begin
  rcv Ack(x, y, v, v', b⃗, s⃗)  →
        skip;
□
  timeout clock ∈ [t + 2·ε, t + T − ε]  →
        v := v';
        v' := any;
        send Aggregate(x, y, v, v', b⃗, s⃗) to y;
        t := clock;
□
  t + T − ε < clock < t  →  t := clock;
end
```

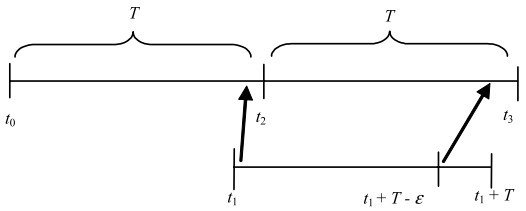

Fig. 4. Timing of *Aggregate* messages.

The source process contains three actions. In the first action, it receives an *Ack* message, which has traversed the network from the destination back to the source. The action has the side effect of updating the bit-vector values at the source to those in the message.

The second action is a timeout action, in which an *Aggregate* message is sent to the destination. Variable t stores the time at which the last *Aggregate* message was sent. To ensure old *Aggregate* and *Ack* messages have left the network before sending a new one, *Aggregate* messages are sent with at least $2 \cdot \varepsilon$ seconds in between. Furthermore, to ensure the message arrives in time at the routers, the message should be sent no later than time $t + T - \varepsilon$. We assume that execution of actions is done such that the timeout will be executed within the right time interval. Failure to do so is considered a fault.

The last action is a sanity action in which t is restored to a sensible value in case of a fault.

The specification of the router is as follows.

process *router R*
const
  T : **integer** {shadow interval}
var
  s : **boolean** {shadow bit}
  $A, \widehat{A}$: **data** {data aggregate and its shadow copy}
  t : **integer** {time of last timeout}
begin
  rcv $Aggregate(x, y, v, v', \vec{b}, \vec{s})$ $\rightarrow$
    {add flow to shadow aggregate}
    $\widehat{A} := \widehat{A} \oplus v$ **if** $\vec{s}_R \neq s \wedge \vec{b}_R$;
    {update shadow bit before forwarding}
    $\vec{s}_R := s$;
    {remove old value from aggregate}
    $A, \widehat{A} := A \ominus v, \widehat{A} \ominus v$ **if** $\vec{b}_R$;
    {choose whether to add or not the new value}
    $\vec{b}_R :=$ **any**;
    $A, \widehat{A} := A \oplus v', \widehat{A} \oplus v'$ **if** $\vec{b}_R$;
    send $Aggregate(x, y, v, v', \vec{b}, \vec{s})$ **to** y
□
  rcv $Ack(x, y, v, v', \vec{b}, \vec{s})$ $\rightarrow$
    send $Ack(x, y, v, v', \vec{b}, \vec{s})$ **to** x
□
  timeout $clock > t + T$ $\rightarrow$
    $s, A, \widehat{A} := \neg s, \widehat{A}, 0$;
    $t := clock$;
□
  $clock < t$ $\rightarrow$ $t := clock$;
end

In the first action, an *Aggregate* message is received, and is forwarded along the next hop to the destination. Before forwarding the message, the value of the flow is added to the shadow variable $\widehat{A}$, provided a state change has occurred from the last time an *Aggregate* message from this flow was received, i.e., $\vec{s}_R \neq s$, and also, provided the value has been added already to the aggregate, i.e., $\vec{b}_R$. Also, the router nondeterministically chooses whether to add the new value v' to the aggregate, and updates $\vec{b}_R$, A, and $\widehat{A}$ accordingly.

In the second action, an *Ack* is received. The router simply forwards the message in the direction of the source.

In the third action, the router changes its state after T seconds from its last state change. Thus, $\widehat{A}$ is assigned to A, $\widehat{A}$ is reset to zero, and bit s is flipped. The time of the state change is recorded in t.

The last action is a sanity action to restore t to a sensible value in case of a fault.

The specification of the destination process is given next.

process *destination y*
begin
rcv $Aggregate(x, y, v, v', \vec{b}, \vec{s})$ $\rightarrow$
send $Ack(x, y, v, v', \vec{b}, \vec{s})$ **to** x
end

It simply consists of a single action that receives an *Aggregate* message and returns an *Ack* in the direction of the source of the message.

The correctness proof of the signaling protocol is given in Appendix A.

5.5. Stabilization of the signaling protocol

The above signaling protocol is robust to a variety of faults. E.g., if a source dies, then its value will be removed from the aggregate of all routers within $2 \cdot T$ seconds, as follows. Within the first T seconds, $\widehat{A}$ is reset to zero. Since the source has died, its value is never added again to $\widehat{A}$, and within the next T seconds, $\widehat{A}$ is assigned to A. Similarly, if the path of a source changes, then, within $2 \cdot T$ seconds, routers along the previous path will remove the source from their aggregate, while routers along the new path will add it to their aggregate. If the aggregate at a router has a corrupted value, it will also correct itself within $2 \cdot T$ seconds. Thus, the protocol reaches a normal operating state within $2 \cdot T$ seconds.

A proof of the stabilization of the signaling protocol is given in Appendix A.

6. Signaling for max–min fairness

We next address how to modify the signaling protocol for the specific case of computing a max–min fair bandwidth assignment for all flows.

6.1. State maintained by routers

Consider the algorithm to compute max–min fairness given in Section 4. Recall that

- $G(p, q)$ is the set of flows whose bandwidth has been assigned at a link other than (p, q) (i.e., flows that are bottled at a link other than (p, q)), and
- $U(p, q)$ is the set of flows whose bandwidth has not been assigned.

When an edge (p, q) is chosen at an iteration step, the bandwidth that is assigned to each flow in $U(p, q)$ is the following:

$$\frac{C(p, q) - \sum_{f \in G(p,q)} f.w}{|U(p, q)|}. \tag{3}$$

Due to the distributed nature of our signaling protocol, the iterative approach is not possible. Instead, a router could maintain an estimate of the values used in (3). In particular, note that the individual values of $f.w$ are not important; only their *sum* is important. Hence, (3) consists simply of three integers, which require only constant space at the router process in charge of channel (p, q).

This suggests that the information we maintain at each router process is as follows.

C: The (constant) bandwidth of the output channel of the process.
W: The sum of the bandwidths of flows that are not bottled at this router; that is, flows who cannot increase their bandwidth because another router is preventing them from doing so. This is an estimate of $\sum_{f \in G(p,q)} f.w$.
n: The total number of flows that are bottled at this router; that is, their bandwidth is limited by this router. This is an estimate on $|U(p, q)|$.

The bandwidth allocated to these flows, which we denote by B, is simply

$$B = \frac{C - W}{n}.$$

Consider the example in Fig. 5(a). Five flows traverse a router R whose output channel capacity is 100. Three flows, f_1, f_2 and f_3, are bottled at a different router, and their collective bandwidth is 60. The remaining flows, f_4 and f_5, are bottled at this router. Hence, $W = 60$, $n = 2$, and $B = 20$. Because f_4 and f_5 are bottled at R, their bottled bandwidth is B, i.e., 20.

Note that the router is unaware of the individual bandwidth of flows f_1, f_2, and f_3. In particular, the bandwidth of f_2 is greater than B, which is not allowed; f_2 should become bottled at R, and its bandwidth should be shared with the other bottled flows. Since R is unaware of the bandwidth of f_2, the source of f_2 must correct this problem via signaling. We address this below.

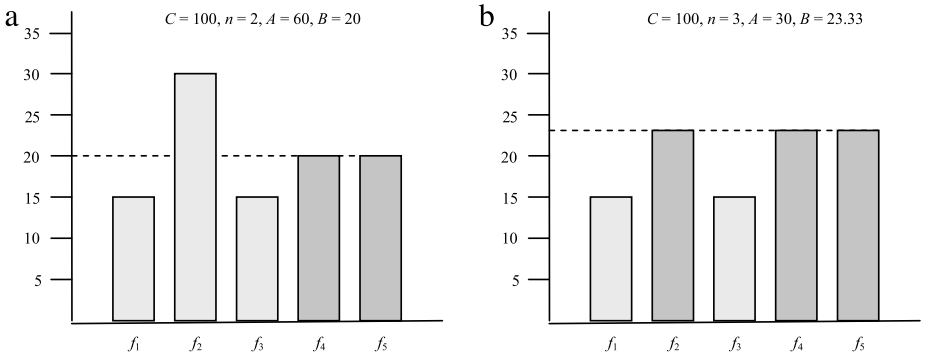

Fig. 5. State maintained at a router R.

6.2. Updating of router state via signaling

Consider again Fig. 5(a). Since f_2 is not bottled at R, it must be bottled at another router, R', which limits the bandwidth of f_2 to 30. Hence, $R'.B = 30$. However, since $30 = R'.B > R.B = 20$, f_2 must become bottled at R instead. The steps to change the bottleneck router of f_2 from R' to R are as follows.

For f_2 to become bottled at R, $R.n$ must be increased by one, and aggregate $R.W$ must be decreased by f_2's current bandwidth. Thus, $R.n = 2 + 1 = 3$, and $R.W = 60 - 30 = 30$. The new bandwidth of f_2, and of any other flow bottled at R, will be $R.B = (R.C - R.W)/R.n = (100 - 30)/3 = 23.33$. This new state of R is shown in Fig. 5(b).

For f_2 to become unbottled at R', $R'.n$ must be reduced by one, and its aggregate, $R.W$, must be increased by the new bandwidth of f_2, i.e., by 23.33.

To perform the above steps, sources are required to periodically send an *Aggregate* message, with the following fields:

x: id of source node.
y: id of destination node.
$\vec{b}$: bit vector, with one bit per router along the path; $\vec{b}_R$ is true if the flow is bottled at router R.
w_{old}: existing bandwidth of the flow; this is already included in the aggregate of all non-bottleneck routers.
w_{new}: new bandwidth of the flow; w' will replace w in the aggregate of all non-bottleneck routers.
w_{next}: this field is discussed further below.

When a router R receives the above *Aggregate* message, it performs the following steps.

- Update $R.W$ by adding w_{new} and subtracting w_{old}, provided the flow is bottled at R, i.e., if $\vec{b}_R$ is true.
- Decide if the bottled state of the flow must change, i.e., if the flow is bottled and should instead be unbottled, and vice versa. There are four cases to consider for this:
  1. If the flow is bottlenecked at the router and its bandwidth is greater than the bottleneck bandwidth ($\vec{b}_R \wedge R.w_{new} > R.B$), then the flow remains bottlenecked at the router, but the source must be informed that its bandwidth should be decreased (see the discussion on w_{next} below).
  2. If the flow is bottlenecked at the router and its bandwidth is less than the bottleneck bandwidth ($\vec{b}_R \wedge R.w_{new} < R.B$) $r < B$, then the flow should no longer be considered bottlenecked at this router. Thus, its bandwidth w_{new} is added to $R.W$, and the number of bottlenecked flows $R.n$ is decreased by one.
  3. If the flow is not bottlenecked at this router and its bandwidth is greater than the bottleneck bandwidth ($\neg\vec{b}_R \wedge R.w_{new} > R.B$), then the flow must become bottlenecked at this router. Hence, $R.n$ increases by 1, and $R.w$ decreases by w_{new}.
  4. If the flow is not bottlenecked at this router, and its bandwidth is less than the bottleneck bandwidth ($\neg\vec{b}_R \wedge R.w_{new} < R.B$), then the state of the router remains the same.

From the above discussion, it follows that routers maintain two aggregate values: an aggregate of the bandwidth w of each source, and a count of bottled flows (aggregating the value "1" for each source). Also, the router has the freedom to choose whether or not to include a flow in these aggregates, depending on its current bottleneck bandwidth. Note that these requirements match those of the signaling protocol of Section 5 and, hence, it may be used to maintain these two aggregates in a self-stabilizing manner.

To simplify our code below, we do not deal with shadow copies of the aggregate variables nor with shadow bits, which are part of the signaling protocol. We focus instead on the bandwidth of each flow and the aggregates at the routers.

We conclude by describing the final field, w_{next}, in the *Aggregate* message. This field is set to infinity by the source, and is used to determine the new bandwidth of the source during the next round of signaling messages. As the *Aggregate* message traverses each router, w_{next} is set to the minimum of its previous value and the bottled bandwidth of the router. The field is then returned to the source via the *Ack* message. This allows the source to learn of the minimum bottled bandwidth of all its

routers. The source's bandwidth cannot exceed this minimum, thus, w_{next} is assigned to w_{new} at the beginning of the next signaling round.

6.3. Specification of max–min signaling

We now present the specification of the source, router, and destination processes.

process *source x*
const
y : **process id** {destination}
var
$\vec{b}$: **bit vector** {bottleneck-bit vector}
w_{old},
w_{new},
w_{next} : **non-negative integer** {flow bandwidth}
$wait$: **boolean** {waiting for an Ack}
begin
rcv $Ack(x, y, w_{old}, w_{new}, w_{next}, \vec{b})$ $\rightarrow$
$w_{old} := w_{new}$;
$w_{new} := w_{next}$;
$wait :=$ **false**

$\square$

$\neg wait$ $\rightarrow$
send $Aggregate(x, y, w_{old}, w_{new}, \infty, \vec{b})$ **to** y
$wait :=$ **true**
end

The source contains two actions. In the first action, the source receives an *Ack* message. The values of w_{old} and w_{new} are updated. The value of $\vec{b}$ is updated as a side effect of receiving the message. The second action sends a new *Aggregate* message provided the previous *Ack* has been received.

The specification of a router process is as follows.

process *router R*
const
C : **non-negative integer** {channel bandwidth}
var
n : **non-negative integer** {count of bottled flows}
W : **non-negative integer** {bandwidth of flows bottled elsewhere}
begin
rcv $Aggregate(x, y, w_{old}, w_{new}, w_{next}, \vec{b})$ $\rightarrow$
{update rate if not bottled}
$W := W - w_{old} + w_{new}$ **if** $\neg \vec{b}_R$;
{un-bottle the flow if necessary}
$n, W, \vec{b}_R := n - 1, W + w_{new},$**false** **if** $\vec{b}_R \wedge w_{new} < B \wedge n > 1$;
{bottle the flow if necessary}
$n, W, \vec{b}_R := n + 1, W - w_{new},$**true** **if** $\neg \vec{b}_R \wedge (w_{new} > B \vee n = 0)$;
{update w_{next} before forwarding}
$w_{next} := \min(w_{next}, B)$;
send $Aggregate(x, y, w_{old}, w_{new}, w_{next}, \vec{b})$ **to** y

$\square$

rcv $Ack(x, y, w_{old}, w_{new}, w_{next}, \vec{b})$ $\rightarrow$
send $Ack(x, y, w_{old}, w_{new}, w_{next}, \vec{b})$ **to** x
end

The router contains two actions. In the first action, an *Aggregate* message is received. The router determines if a flow that is bottled should be unbottled, and vice versa. In this action, B is defined as follows:

$$B = \frac{C - W}{n}.$$

To ensure B is well defined, we ensure that each router has at all times at least one bottled flow. The second action simply helps to forward an acknowledgment back to the source.

The destination is similar to before; it receives an *Aggregate* message and returns an *Ack* message.

process *destination y*
begin
rcv $Aggregate(x, y, w_{old}, w_{new}, w_{next}, \vec{b}) \rightarrow$
send $Ack(x, y, w_{old}, w_{new}, w_{next}, \vec{b})$ **to** x
end

6.4. Stabilization of max–min signaling

We next present an overview of the stabilization properties of our system. The proof of the following theorem is given in Appendix B.

Theorem 1. *Let f be a flow with source process src and destination process dst. Then, the max–min signaling system stabilizes to the following:*

$$src.w_{old} = f.w$$

where $f.w$ is the bandwidth assigned to flow f by the max–min bandwidth allocation algorithm in Section 4.2.

Corollary 1. *Let F^R be the set of source processes whose flows traverse router process R. Then, the system stabilizes to the conjunction of the following two predicates:*

$$R.W = \left(\sum src \ : \ src \in F^R \ \wedge \ \neg src.\vec{b}_R \ : \ src.w_{old}\right)$$

$$R.n = \left|\{src \mid src \in F^R \ \wedge \ src.\vec{b}_R\}\right| .$$

Above, we did not discuss the stabilization time of our system. As shown in Appendix A, the signaling protocol stabilizes in $O(T)$ time, where T is the interval between state changes at a router. The stabilization time of Theorem 1, on the other hand, still remains an open problem.

It can be shown that if bandwidth values are discrete, then the convergence time is in the order of $O(N \cdot \Delta)$, where N is the number of discrete bandwidth values, and Δ is the time interval between signaling messages from a source. We have shown in Section 5 that $\Delta \leq T - \varepsilon$, so in the worst case, the convergence time is $O(N \cdot T)$, unless a tighter bound is imposed on Δ. To accomplish this, however, we require some synchronization between routers. In particular, the network must operate in "phases". During some phases, routers are not allowed to increase their bottleneck rate, forcing some flows to become bottled. In other phases, the bottleneck rate is allowed to increase, and flows become unbottled. We leave this result to future work.

7. Concluding remarks

All our sources are assumed to be flexible, in the sense that they adapt to the bandwidth provided by the network. If some sources are rigid, that is, they always require a fixed amount of bandwidth, then this can simply be implemented by having each rigid source include its constant rate in the aggregate message, and each router subtracts this rate from the output channel rate C.

We considered only a fixed set of sources. In practice, the set of sources varies over time. Thus, when a source is removed from the network, its information must be subtracted from the aggregate information at each of its routers. Similarly, when a new source appears in the network, its information must be added to the aggregate information at each of its routers. Adding or removing a flow can be performed with a single *Aggregate* message, as follows.

Consider adding a flow to the network. The flow may send an *Aggregate* message with $w_{old} = 0$, $w_{new} =$ {flow's initial rate}, and $\vec{b} = 0$. I.e., a "new" flow can be considered to be an old flow with rate of zero that is not bottled at any router. Similarly, removing a flow could be done by sending an *Aggregate* message with $w_{new} = 0$. In this manner, the flow will be unbottled at all routers, and its bandwidth is removed from all routers. Thus, it no longer contributes to the aggregate variables n and W.

Although routers do not maintain per-flow state, all signaling messages associated with a link must access the aggregate variables n and W of the link. This may potentially become a performance bottleneck of the core router, if the number of flows traversing the link is large. However, given that sources need to transmit a signaling packet only once every T seconds, and given that there is no time synchronization, the processing load of the signaling messages is spread over T seconds. Assuming T is sensible, the signaling load on the router may be kept small.

Further investigation is needed on the stabilization time of the system. We speculate this to be significantly large, due to many possible intermediate values that could be generated at the routers while the system converges. In practice, however, it is likely that the average convergence time is sensible. We plan to investigate this via a future simulation study.

Appendix A. Correctness proof of signaling protocol

As discussed earlier, routing between access networks is outside the scope of the paper. We simply assume that routing is stabilizing[3], and thus the routing tables converge to a sound and stable set of values. This implies the following.

Lemma 1. *The system stabilizes to the following predicate: every Aggregate*$(x, y, \ldots)$ *message is located only along the path from x to y, and every Ack*$(x, y, \ldots)$ *message is located only along the path from y to x.*

Similarly, due to the time restrictions on the sending of messages by the source and the fast processing of messages at the routers, we have the following.

Lemma 2. *The system stabilizes to the following predicate: for every x and y, the number of Aggregate*$(x, y, \ldots)$ *messages plus the number of Ack*$(x, y, \ldots)$ *messages is at most one.*

Finally, due to the timing of the state changes of the routers, and the timing on the generation of signaling messages by the source, as argued in Section 5, we have the following.

Lemma 3. *For every flow with source x and destination y traversing router R, and for every interval of time of length at least T, router R will receive an Aggregate*$(x, y, \ldots)$ *message from source x.*

We next consider the relationship between the rates of the sources and the information stored at the routers.

To simplify our lemmas, we introduce two auxiliary variables at each router. No other variable will depend on the values of the auxiliary variables, i.e., they are simply used to reason about the state of the router.

At each router, we add two auxiliary arrays: $A[]$ and $\widehat{A}[]$. Intuitively, $A[x]$ corresponds to the value from source x that is added to aggregate A. Similarly, $\widehat{A}[x]$ corresponds to the value from source x that is added to aggregate $\widehat{A}$.

In this manner, when 0 is assigned to $\widehat{A}$, then, for all x, $\widehat{A}[x]$ is also assigned 0. Also, when a value from source x is added or subtracted from A (or $\widehat{A}$), the same value is added or subtracted from $A[x]$ (respectively, $\widehat{A}[x]$). Finally, when $\widehat{A}$ is assigned to A, then for all x, $\widehat{A}[x]$ is assigned to $A[x]$.

Lemma 4. *Within* $O(T)$ *units of time, for every router R,*

$$R.A = (\oplus\, i :: R.A[x]) \tag{4}$$

$$R.\widehat{A} = \left(\oplus\, i :: R.\widehat{A}[x]\right). \tag{5}$$

Proof. Within T seconds, zero will be assigned to $\widehat{A}$, and hence also to all elements of $\widehat{A}[]$, making predicate (5) true. From this moment onward, every individual value added to or subtracted from $\widehat{A}$ is also added to or subtracted from the appropriate element of $\widehat{A}[]$. Hence, (5) continues to hold. T seconds later, zero is assigned to (A), and the argument repeats. Hence, after T seconds, (5) will hold and continue to hold.

Once (5) holds, then, within another T seconds, $\widehat{A}$ is assigned to A. When this occurs, for all x, $\widehat{A}[x]$ is assigned to $A[x]$. This, along with (5), ensures that (4) holds. Furthermore, from this moment onward, every value added to or subtracted from A is also added to or subtracted from the appropriate element of $A[]$. If $\widehat{A}$ is assigned to A again, then the argument repeats. Hence, within $2 \cdot T$ seconds from our initial state, (4) will hold and continue to hold. □

Given Lemma 4 above, we next continue making reference only to the auxiliary variables and not the aggregate variables.

Let $P(x, y)$ correspond to the sequence of channels along the path from node x to node y. In particular, let path $P(x, R)$ correspond to the sequence of channels from source x to router R.

To simplify our notation, and from Lemma 3, we refer to the *Aggregate* or *Ack* message of source x and destination y simply as $(x, y).msg$. Whether the message is an *Aggregate* or an *Ack* depends on where the message is located. I.e., if the message is in $P(x, y)$, then it is an *Aggregate* message. If it is in $P(y, x)$, then it is an *Ack* message.

Note that the source x has variables, v, v', $\vec{s}$, and $\vec{b}$, and that message $(x, y).msg$ has fields with the same names. Also, when the *Ack* is received back at source x, the variables of x are replaced with the fields in the *Ack*. Therefore, to simplify our proofs, $(x, y).f$ refers to field f in message $(x, y).msg$. If currently there is no signaling message for the pair (x, y), then $(x, y).f$ refers to variable f at source x.

Lemma 5. *For any source x and destination y, and any router R along path* $P(x, y)$*:*

1. *The values of* $(x, y).\vec{s}_R$ *and* $(x, y).\vec{b}_R$ *can only be modified by actions in R.*
2. *The values of* $(x, y).v$ *and* $(x, y).v'$ *can only be modified by actions in x.*

[3] Most routing protocols such as link-state routing and distance-vector routing are in essence stabilizing.

Proof. For part 1, we consider $\vec{s}_R$; the argument for $\vec{b}_R$ is similar.

By definition, $(x, y).\vec{s}_R$ obtains its value from the signaling message between x and y, and if there is no such message, from $x.\vec{s}_R$.

Assume there is no signaling message for the pair (x, y). Thus, by definition, $(x, y).\vec{s}_R = x.\vec{s}_R$. Note that no process other than x can modify $x.\vec{s}_R$. For x, the first action is disabled (there is no *Ack* message), and the third action does not refer to $x.\vec{s}_R$. The second action sends an *Aggregate* message, so, by definition, $(x, y).\vec{s}_R$ becomes the field $\vec{s}_R$ in this message. However, the field's value is obtained from $x.\vec{s}_R$ and, hence, the value of $(x, y).\vec{s}_R$ is unchanged.

Assume now that there is a signaling message for the pair (x, y). Any router Q, where $Q \neq R$, cannot affect the value of $(x, y).\vec{s}_R$; this value is simply forwarded along (either in an *Aggregate* or *Ack* message). The destination simply transforms an *Aggregate* message into an *Ack* message without changing the field values. Thus, $(x, y).\vec{s}_R$ remains unchanged. If the *Ack* is received at the source, then, after the message is received, $(x, y).\vec{s}_R$ is defined to be $x.\vec{s}_R$. However, $x.\vec{s}_R$ updated its value from the *Ack* message and, hence, $(x, y).\vec{s}_R$ remains unchanged.

For part 2, notice that no router modifies fields v or v' in a signaling message, whether an *Aggregate* or *Ack* message, and the destination turns an *Aggregate* message into an *Ack* message, without changing the field values. Hence, only the source x can change the values of $(x, y).v$ and $(x, y).v'$. □

We next examine the relationship between the shadow bit from a source x, and the aggregate values stored at a router R. The relationship depends on the location of the signaling message, i.e., if it is on its way to R or if it has gone beyond R.

Lemma 6. *Within $O(T)$ time, the system stabilizes to the following predicate: for every source–destination pair (x, y) and every router R along their path,*

$$\begin{array}{c} R.s = (x, y).\vec{s}_R \ \wedge\ R.\widehat{A}[x] = U_{x,R} \\ \vee \\ R.s \neq (x, y).\vec{s}_R \ \wedge\ R.\widehat{A}[x] = 0 \end{array} \tag{6}$$

where

$$U_{x,R} = \begin{cases} (x, y).v & \text{if } (x, y).\vec{b}_R \wedge (x, y).msg \in P(x, R) \\ (x, y).v' & \text{if } (x, y).\vec{b}_R \wedge (x, y).msg \notin P(x, R) \\ 0 & \text{if } \neg(x, y).\vec{b}_R. \end{cases}$$

Proof.

Predicate (6) holds within $O(T)$ seconds

We first show that predicate (6) above will hold within $2 \cdot T$ seconds. For the initial state of the computation, we separately consider two cases: $R.s = (x, y).\vec{s}_R$ and $R.s \neq (x, y).\vec{s}_R$.

Assume first that $R.s = (x, y).\vec{s}_R$. From Lemma 2, only R can change the value of $(x, y).\vec{s}_R$ (and obviously of $R.s$). Thus, $R.s = (x, y).\vec{s}_R$ continues to hold until R changes one of these two values. Only the first and third actions of R refer to $R.s$ or $(x, y).\vec{s}_R$. The first action maintains $R.s = (x, y).\vec{s}_R$ (the message bit is set to the router's bit). On the other hand, the third action, which is guaranteed to execute in T seconds, flips the value of $R.s$. Thus, $R.s \neq (x, y).\vec{s}_R$ holds after the action. The action also sets $R.\widehat{A}[x] = 0$, which implies (6) holds.

Assume next that $R.s \neq (x, y).\vec{s}_R$. From Lemma 3, within the next T seconds, a message from source x is received at R, and the first action of R sets $\vec{s}_R$ to be equal to $R.s$. Hence, $R.s = (x, y).\vec{s}_R$ holds after the action, and we may repeat the argument in the previous paragraph.

Preserving predicate (6)

We next show that once predicate (6) above holds it will continue to hold, by showing that all actions preserve the predicate.

Source actions

First action: Of the values in (6), this action only makes reference to $(x, y).\vec{s}_R$. From Lemma 5, source x cannot change this value, and predicate (6) continues to hold.

Second action: Since (6) holds before the action, consider first that $R.s \neq (x, y).\vec{s}_R \ \wedge\ R.\widehat{A}[x] = 0$ holds. From Lemma 5, source x cannot change any of these values, so executing the action preserves (6).

Assume instead that $R.s = (x, y).\vec{s}_R \ \wedge\ R.\widehat{A}[x] = U_{x,R}$ holds before the action. As above, x cannot change the value of $R.s = (x, y).\vec{s}_R$, nor the value of $R.\widehat{A}[x]$ (it is local to R). However, it may affect $U_{x,R}$, so we must argue that $U_{x,R}$ remains the same.

From Lemma 5, x cannot change the value of $(x, y).\vec{b}_R$. If $(x, y).\vec{b}_R$ is false, then $U_{x,R}$ does not change, and we are done.

Assume instead that $(x, y).\vec{b}_R$ is true. Then, before the action is executed, Lemma 2 implies that $(x, y).msg \notin P(x, R)$ and, hence, $U_{x,R} = (x, y).v'$. From Lemma 2 again, and by definition, $(x, y).v' = x.v'$.

The action sends a signaling message and, thus, $(x, y).msg \in P(x, R)$ holds after the action. By definition, $U_{x,R} = (x, y).v$ after the action. However, the action also assigns $x.v'$ to $x.v$ and, thus, $U_{x,R}$ does not change, as desired.

Third action: This is a sanity action that will be executed only once in the system. Thus, it will not affect predicate (6).

Destination actions

The destination has only a single action that receives an *Aggregate* message and sends an *Ack* message. Thus, it has no effect on (6).

Router actions

First action: We have two cases to consider.

- Assume $R.s = (x, y).\vec{s}_R$ holds before the action. From (6), we have $R.\widehat{A}[x] = (x, y).v$ if $(x, y).\vec{b}_R$, or zero otherwise.
  Consider the assignments of the action. From $R.s = (x, y).\vec{s}_R$, the first two assignments do not change the state of the system. The third assignment sets $R.\widehat{A}[x]$ to zero; this is because, if $(x, y).\vec{b}_R$, then $(x, y).v$ is subtracted from $R.\widehat{A}[x]$, and if $\neg(x, y).\vec{b}_R$, then $R.\widehat{A}[x]$ was zero already.
  Then, the router nondeterministically chooses to add $(x, y).v'$ to $R.\widehat{A}[x]$, and the nondeterministic choice is stored in $(x, y).\vec{b}_R$. Hence, after this statement, $R.\widehat{A}[x] = (x, y).v'$ if $(x, y).\vec{b}_R$, and 0 otherwise, i.e., $R.\widehat{A}[x] = U_{x,R}$.
  Combining the above, the first disjunct in (6) holds after the message is forwarded.
- Assume $R.s \neq (x, y).\vec{s}_R$ holds before the action. From (6), we have $R.\widehat{A}[x] = 0$.
  Consider the value of $R.\widehat{A}[x]$ after the first three assignment statements. If $\neg\vec{b}_R$ holds beforehand, then $R.\widehat{A}[x]$ remains unchanged. If $\vec{b}_R$ holds beforehand, then v is added to $\widehat{A}[x]$ and then subtracted from it. Hence, $R.\widehat{A}[x]$ remains unchanged (i.e., zero).
  Note that the second assignment statement causes $R.s = (x, y).\vec{s}_R$ to be true.
  In the fourth and fifth assignment statements, the router nondeterministically chooses to add $(x, y).v'$ to $R.\widehat{A}[x]$, and the nondeterministic choice is stored in $(x, y).\vec{b}_R$. Hence, after this statement, $R.\widehat{A}[x] = (x, y).v'$ if $(x, y).\vec{b}_R$, and 0 otherwise, i.e., $R.\widehat{A}[x] = U_{x,R}$.
  Hence, the first disjunct in (6) holds after the message is forwarded.

Second action: This action only moves an *Ack* message one step closer to the source, and hence it does not affect (6).
Third action: The router performs a state update by flipping its shadow bit and assigning zero to $\widehat{A}$. From (6), we have two cases to consider regarding the state before the action.

- If $R.s = (x, y).\vec{s}_R \wedge R.\widehat{A}[x] = U_{x,R}$ holds before the action, then $R.s \neq (x, y).\vec{s}_R \wedge R.\widehat{A}[x] = 0$ holds after the action.
- Instead, assume $R.s \neq (x, y).\vec{s}_R \wedge R.\widehat{A}[x] = 0$ holds before the action. We claim that this cannot hold when the action is going to be executed. The reason is as follows.
  The third action executes at least T seconds apart. In the interim from the previous execution of this action, due to Lemma 3, a message from the source must have been received, causing action one to execute. Action one causes $(x, y).\vec{s}_R = R.s$ to hold after its execution.
  Note that the source does not change the value of $(x, y).\vec{s}_R$ (Lemma 5), and the router can only change the value of $R.s$ in the third action. Thus, $(x, y).\vec{s}_R = R.s$ remains true until the third action is executed again. Hence, the second disjunct in (6) cannot hold before the third action is executed.

Fourth action: This action makes no reference to any values in (6). □

From the above lemma, we can conclude the proof of the signaling message with the following theorem.

Theorem 2. *Within $O(T)$ seconds, the system stabilizes to the following predicate: for every source–destination pair (x, y) and every router R along its path,*

$$R.A[x] = U_{x,R}. \tag{7}$$

Proof. We first prove that if (7) above holds, it will continue to hold. We assume we have already reached a state where all the previous lemmas hold. We will prove later that eventually (7) holds.

Predicate (7) is stable

Consider the actions of every process.

Source actions

First action: The value of $(x, y).f$ for any field f of the message is defined to be that of the source variable when there is no message from the source. Hence, the side effect of receiving a message, i.e., changing the variables of the source to those values in the message, does not affect predicate (7).

Second action: Source x cannot change the value of $R.A[x]$ (local to R), nor the value of $(x, y).\vec{b}$ (Lemma 5), but it can affect $U_{x,R}$ by changing $(x, y).v$, $(x, y).v'$, and $(x, y).msg \in P(x, R)$.

Before the action executes, and from Lemma 2, $(x, y).msg \notin P(x, R)$ holds, and after the action, $(x, y).msg \in P(x, R)$ holds. We must show that in both cases, $U_{x,R}$ remains unchanged.

If $\neg(x, y).\vec{b}_R$ before the action executes, then, by definition, $U_{x,R}$ remains zero (and thus unchanged).

If $(x, y).\vec{b}_R$, then, by definition, $U_{x,R} = (x, y).v'$ holds before the action is executed, and $U_{x,R} = (x, y).v$ holds afterward. Note that the action assigns $(x, y).v'$ to $(x, y).v$. Thus, $U_{x,R}$ remains unchanged, as desired.

Third action: This is a sanity action that will be executed only once in the system. Thus, it will not affect predicate (7).

Destination actions

Consider the destination. It has only a single action that receives an *Aggregate* message and sends an *Ack* message. Thus, it does not affect any values in (7).

Router actions

First action: We first argue that the first disjunct of (6) holds after the first two assignment statements of the action. First, assume $R.s = (x, y).\vec{s}_R \wedge R.\widehat{A}[x] = U_{x,R}$ holds before the action. The first two assignments do not change this. Assume instead that $R.s \neq (x, y).\vec{s}_R \wedge R.\widehat{A}[x] = 0$. The first statement ensures $R.\widehat{A}[x] = U_{x,R}$, and the second that $R.s \neq (x, y).\vec{s}_R$.

Next, the router chooses whether or not to assign the value to its aggregate. Because $R.\widehat{A}[x] = U_{x,R}$ holds before this statement, we have that $R.A[x] = U_{x,R}$ will hold after the action.

Second action: This action only moves an *Ack* message one step closer to the source, and hence it does not affect (7).

Third action: The router performs a state update by flipping its shadow bit and assigning zero to $\widehat{A}$. Given the timing of messages, at least one *Aggregate* message from the source must be received from the last time this action was executed. When this message is received and processed, from (6) we have

$$R.s = (x, y).\vec{s}_R \ \wedge \ R.\widehat{A}[x] = U_{x,R}.$$

Since (6) always holds, and since the shadow bit of the router will not change until the third action is executed, then this also holds the moment before the action is executed. Thus, after the action is executed we will have

$$R.s \neq (x, y).\vec{s}_R \ \wedge \ R.A[i] = U_{x,R}$$

which implies our desired result of $R.A[i] = U_{x,R}$.

Fourth action: This is a sanity action that is executed only once in the router, and hence will not affect (7) above.

Predicate (7) holds within $O(T)$ seconds

Note that, from the argument of the third action, (7) will hold after the third action is executed, which happens every T seconds, and, hence, the system stabilizes to (7). □

Appendix B. Proof of Theorem 1

In this section, we provide a proof for Theorem 1. We begin with a few corollaries and lemmas.

Corollary 2. *For every every router R,*

$$R.W \ = \left(\sum x, y : \ R \in P(x, y) \ : \ V_{x,R}\right) \tag{8}$$

$$R.n \ = \left(\sum x, y : \ R \in P(x, y) \ : \ W_{x,R}\right) \tag{9}$$

where

$$V_{x,R} = \begin{cases} (x, y).w_{old} & \text{if } \neg(x, y).b_R \ \wedge \ (x, y).msg \in P(x, R) \\ (x, y).w_{new} & \text{if } \neg(x, y).b_R \ \wedge \ (x, y).msg \notin P(x, R) \\ 0 & \text{if } (x, y).b_R \end{cases}$$

and

$$W_{x,R} = \begin{cases} 1 & \text{if } (x, y).b_R \ \wedge \ (x, y).msg \in P(x, R) \\ 1 & \text{if } (x, y).b_R \ \wedge \ (x, y).msg \notin P(x, R) \\ 0 & \text{if } \neg(x, y).b_R. \end{cases}$$

Proof. We have shown earlier that the signaling protocol allows routers to aggregate data from source nodes, and to nondeterministically choose whether to aggregate the data of each individual source.

In max–min signaling, the router is aggregating two values from each source: its bandwidth w, and a count (i.e. the simple value 1). Rates are aggregated for flows which are not bottled at the router, and the count is for flows that are bottled. Thus, we can view this as two copies of the aggregating signaling protocol of Section 5. However, only one bitmap $\vec{b}$ is needed, because, for each flow, either the router aggregates the flow's rate or the value 1, but not both. Thus, V and W are the duals of U in Theorem 2, and the corollary follows. □

For a router R, let us define the bandwidth $R.B$ of its bottled flows as follows:

$$R.B = \frac{R.C - R.W}{R.n}.$$

Also, define F^R to be the set of flows traversing R.

We next show that $R.W$ will eventually have a sensible value, that is, the sum of the bandwidths of unbottled flows is not more that the channel bandwidth of the router. By definition, any remaining bandwidth is shared among the bottled flows.

Lemma 7. *The system stabilizes to the following predicate: for every router R, where* $F^R \neq \emptyset$,

$$R.W \leq R.C \ \wedge \ R.n > 0$$

i.e., $R.B \geq 0$.

Proof. We assume we have reached a state where Corollary 2 already holds and, hence, $R.W$ and $R.n$ correctly reflect the aggregate rate and number of bottled flows at the router.

Given that the router will always bottle a flow if $R.n = 0$, then when the next *Aggregate* message arrives from any flow, the flow will be bottled, regardless of its rate. From the first action of the router, it will never allow $R.n$ to reach 0 again. Hence, $R.B$ is well defined from this moment onwards.

Consider any router R. If $R.W > R.C$, this implies that $R.B < 0$. Thus, whenever a message is received from *any* unbottled flow, the router will bottle the flow, which in turn decreases $R.W$.

As unbottled flows become bottled, $R.W$ decreases. While $R.W$ remains greater than C, no unbottled flow can increase or decrease its rate, it will simply become bottled, and thus $R.W$ continues to decrease, until $R.W \leq R.C$.

We next need to show that once $R.W \leq R.C$, this continues to hold.

$R.W$ increases under two conditions: (a) a bottled flow becomes unbottled, and (b) a flow requests an increase in rate (i.e. $w_{new} > w_{old}$). If a flow becomes unbottled, then from the definition of $R.B$, $R.W$ remains at most $R.C$, because the remaining bandwidth is distributed evenly among the bottled flows. If a flow requests an increase in its rate (and it is granted without being bottled), it has to be the case that $R.W < R.C$ after the increase, since, otherwise, the flow would become bottled and $R.W$ would actually decrease. □

Lemma 8. *Let* Φ *be the following set of values.*

- *For every router R, R.B.*
- *For every source–destination pair* (x, y)*, the values:*

$$(x, y).w_{old},\ (x, y).w_{new},\ (x, y).w_{next}.$$

Then, the minimum value in Φ *is non-decreasing.*

Proof. We consider the actions of each process, and show that Φ does not decrease.

Source actions

First action: An *Ack* is received, and its values are copied to the variables of the sender. Hence, no new values are generated, and the minimum in Φ cannot decrease.

Second action: An *Aggregate* message is sent, with values from the variables of the source (and hence from Φ), with the only new value being infinity, so again the minimum in Φ cannot decrease.

Destination actions

The destination simply returns an *Ack* after receiving an *Aggregate* message. No new values are generated, so the minimum of Φ cannot decrease.

Router actions

First action: The values of router R and pair (x, y) that get affected are $R.B$, and $(x, y).w_{next}$.

Note that if $R.B$ does not decrease, then $(x, y).w_{next}$ cannot decrease below the minimum in Φ, because it is set to the minimum of its previous value and $R.B$, and $R.B$ is in Φ. Our concern is when $R.B$ indeed decreases. We must show that its new value is not smaller than another existing value in Φ.

Consider thus the first action of the router. There are multiple cases to consider.

(a) The rate of an unbottled flow (x, y) decreases (i.e. $(x, y).w_{old} > (x, y).w_{new}$). When $R.W$ is updated, it will decrease, and this decrease in $R.W$ increases the value of $R.B$, as desired. If, in addition, the flow becomes bottled at the end of the action, then $R.B$ increases even more, as desired.

(b) The rate of an unbottled flow (x, y) increases (i.e. $(x, y).w_{old} \leq (x, y).w_{new}$). This increases $R.W$, which in turn reduces the value of $R.B$, and we must ensure its new value is at least as large as another value in Φ. We have two subcases.

i. The flow does not become bottled (i.e., $(x, y).w_{new} < R.B$ after $R.W$ is updated): in this case, the new value of $R.B$ is greater than an existing value of Φ (i.e., $(x, y).w_{new}$), as desired.

ii. The flow becomes bottled (i.e., $(x, y).w_{new} \geq R.B$ after $R.W$ is updated): in this case, we have to consider the relationship between $(x, r).w_{old}$ and the old value of $R.B$, which we denote $R.B_{old}$.
   (1) If $(x, y).w_{old} \geq R.B_{old}$, then, after bottling the flow, the amount of bandwidth added to the bottled set is more than $R.B_{old}$, hence, $R.B$ increases as desired.
   (2) If $(x, y).w_{old} < R.B_{old}$, then, after bottling the flow, the net effect is bottling a flow whose original rate is w_{old} (since the new rate w_{new} is given back to the bottled flow bandwidth). Thus, the new value of $R.B$ is $(R.B_{old} \cdot n + w_{old})/(n + 1)$. Since $w_{old} < R.B_{old}$, even though $R.B_{old}$ decreases, it cannot decrease beyond w_{old}, i.e., an existing value of Φ, as desired.

(c) A bottled flow becomes unbottled. In this case, $R.B$ increases.
(d) An unbottled flow becomes bottled. This was already covered as a subcase of cases (a) and (b).

Second action: The second action is trivial (just forwards an *Ack* message) so the minimum in Φ cannot decrease. □

Let B_1 correspond to the bandwidth of the minimum link (first step) in the computation of the max–min fairness allocation of flows.

Lemma 9. *The minimum value of Φ (as defined in Lemma 8) increases until it reaches the value B_1.*

Proof. Let m be the current minimum value in Φ. From Lemma 8, all values in Φ will remain at least m.

Consider any router R. Assume $R.B$ reaches a value greater than m. Once this happens, $R.B$ will always be at least m. This is because $R.B$ decreases only when a flow (x, y) traversing R increases its rate. However, the flow's initial rate $(x, y).w_{old}$ is at least m, and, whether the flow becomes bottled or not, the new value of $R.B$ will be between its previous value and $(x, y).w_{old}$, i.e., $R.B$ remains greater than m.

We must show that if $R.B$ equals m (and less than B_1), then it will eventually increase. Because $R.B$ is less than B_1, then, from the definition of B_1, some flows in R must have a rate in $R.W$ that is greater than B_1. When an *Aggregate* message from any one of these flows is received at R, either the flow becomes bottlenecked (increasing $R.B$), or the flow decreases its rate (also increasing $R.B$). Hence, $R.B$ must grow and become larger than m.

Consider any pair (x, y). What remains to be shown is that $(x, y).w_{old}$, $(x, y).w_{new}$, and $(x, y).w_{next}$ will also increase to a value larger than m. As argued above, for every router R traversed by (x, y), $R.B$ increases beyond m and remains above m. The values of $(x, y).w_{old}$ and $(x, y).w_{new}$ depend on $(x, y).w_{next}$, which in turns receives as value the minimum of the $R.B$ values of all routers R along its path. Since these values have been shown to be strictly greater than m, then $(x, y).w_{old}$, $(x, y).w_{new}$, and $(x, y).w_{next}$, and hence the minimum value in Φ, will become greater than m. □

We now conclude the section with the proof of Theorem 1. We first generalize some definitions.

Let F_i, $1 \leq i \leq n$, be the set of flows that become bottlenecked at step i in the max–min allocation algorithm, and B_i be the bottleneck bandwidth found for flows in F_i. Let the max–min algorithm have a total of n iterations.

Define Φ_i to be the set of values $\{w_{old}, w_{new}, w_{next}\}$ of flows in F_x, where $i \leq x \leq n$, united with the set of $R.B$ values for every router R whose flows, denoted F^R, are a subset of $\bigcup_{i \leq x \leq n} F_x$.

Proof. From Lemma 9, Φ_1 grows to at least B_1.

Consider any router R that becomes a bottleneck in the first iteration of the max–min fairness algorithm. If $R.B > B_1$, then this implies that there is a flow (x, y) through R whose contribution to $R.W$ is less than B_1, which from Theorem 2 implies that $(x, y).w_{old}$ or $(x, y).w_{new}$ is less than B_1, which violates the assumption on Φ_1. Also, $R.B$ cannot be less than B_1, since this would also violate our assumption on Φ_1. Hence, $R.B$ remains fixed at B_1.

Consider any flow (x, y) going through the router R defined above, with $R.B$ fixed at B_1. From Lemma 9, each of $(x, y).w_{old}$, $(x, y).w_{new}$ and $(x, y).w_{next}$ is at least B_1. So whenever a message from (x, y) arrives at R, $(x, y).w_{next}$ becomes equal to B_1. From $(x, y).w_{next}$, $(x, y).w_{new}$ becomes B_1, and then after another message is sent by (x, y), $(x, y).w_{old}$ becomes equal to B_1. Hence, all flows in F_1 will have bandwidth values equal to B_1 and remain fixed at this value.

Since F_1 flows are fixed, and all other values in Φ_1 are at least B_1, we can basically ignore flows in F_1, and repeat the argument of Lemma 9 to show that Φ_2 grows to B_2, and repeat the above argument to show that all flows in F_2 will have a bandwidth of B_2 and remain in this state.

The first part of the theorem follows by induction. Corollary 1 follows from Theorem 2. □

References

[1] J. Heinanen, F. Baker, W. Weiss, J. Wroclawski, Assured Forwarding PHB Group, RFC 2597 (Proposed Standard), updated by RFC 3260, June 1999, http://www.ietf.org/rfc/rfc2597.txt.
[2] V. Jacobson, K. Nichols, K. Poduri, An Expedited Forwarding PHB, RFC 2598 (Proposed Standard), obsoleted by RFC 3246, June 1999, http://www.ietf.org/rfc/rfc2598.txt.
[3] R. Braden, D. Clark, S. Shenker, Integrated Services in the Internet Architecture: an Overview, RFC 1633 (Informational), June 1994, http://www.ietf.org/rfc/rfc1633.txt.
[4] J. Wroclawski, Specification of the Controlled-Load Network Element Service, RFC 2211 (Proposed Standard), Sep. 1997, http://www.ietf.org/rfc/rfc2211.txt.
[5] T. Bonald, L. Massoulié, A. Proutière, J. Virtamo, A queueing analysis of max–min fairness, proportional fairness and balanced fairness, Queuing Systems 53 (2006) 65–84.
[6] J.-Y.L. Boudec, Rate adaptation, congestion control and fairness: a tutorial, 2008, http://ica1www.epfl.ch/PS_files/LEB3132.pdf.

[7] S. Abraham, A. Kumar, A stochastic approximation approach for max–min fair adaptive rate control of ABR sessions with MCRs, in: Proceedings of IEEE INFOCOM, New York, NY, 1999.
[8] A. Charny, An algorithm for rate allocation in a packet switching network with feedback, M.S. Thesis, Massachusetts Institute of Technology, May 1994.
[9] Y.T. Hou, H.H.Y. Tzeng, S.S. Panwar, A generalized max–min rate allocation policy and its distributed implementation using the ABR flow control mechanism, in: Proceedings of the IEEE INFOCOM, San Francisco, CA, 1998.
[10] J. Ros, W.K. Tsai, A general theory of constrained max–min rate allocation for multicast networks, in: IEEE International Conference on Networks, Singapore, 2000.
[11] S. Sarkar, T. Ren, L. Tassiulas, Achieving fairness in multicasting with almost stateless rate control, in: Proceedings of the conference on Scalability and Traffic Control in IP Networks, SPIE, ITcom, 2002.
[12] Y. Kim, W.K. Tsai, M. Iyer, J. Ros, Minimum rate guarantee without per-flow information, in: Proceedings of the IEEE International Conference on Network Protocols, CNP, IEEE Computer Society, Washington, DC, USA, 1999, pp. 155–162.
[13] D.H.K. Tsang, W.K.F. Wong, New rate-based switch algorithm for abr traffic to achieve max–min fairness with analytical approximation and delay adjustment, in: Proceedingsof te IEEE Globecom Conference, 1996.
[14] R. Wu, A. Chien, Gtp: group transport protocol for lambda-grids, in: IEEE International Symposium on Cluster Computing and the Grid, 2004, pp. 228–238.
[15] Y. Liu, K.C. Chua, G. Mohan, Achieving max–min fairness in wdm optical burst switching networks, in: IEEE Workshop on High Performance Switching and Routing, 2005, pp. 187–191.
[16] X. Su, S. Chan, Max–min fair rate allocation in multi-hop wireless ad hoc networks, in: IEEE International Conference on Mobile Adhoc and Sensor Systems, MASS, 2006, pp. 513 –516.
[17] C. Zhou, N. Maxemchuk, Scalable max–min fairness in wireless ad hoc networks, Ad Hoc Networks 9 (2) (2011) 112–119.
[18] A. Sridharan, B. Krishnamachari, Max–min fair collision-free scheduling for wireless sensor networks, in: IEEE International Conference on Performance, Computing, and Communications, 2004, pp. 585–590.
[19] A. Sridharan, B. Krishnamachari, Maximizing network utilization with max–min fairness in wireless sensor networks, Wireless Networks 15 (5) (2009) 585–600.
[20] F. Jafari, M.H. Yaghmaee, M.S. Talebi, A. Khonsari, Max–min-fair best effort flow control in network-on-chip architectures, in: Proceedings of the International Conference on Computational Science, in: LNCS, vol. 5101, Springer, 2008, pp. 436–445.
[21] S. Mohanty, C. Liu, B. Liu, L. Bhuyan, Max–min utility fairness in link aggregated systems, in: IEEE Workshop on High Performance Switching and Routing, 2007, pp. 1 –7.
[22] A. Arora, M. Gouda, Closure and convergence: a foundation of fault-tolerant computing, IEEE Transactions on Software Engineering 19 (11) (1993) 1015–1027.
[23] S. Dolev, T. Herman, Superstabilizing protocols for dynamic distributed systems, Chicago Journal of Theoretical Computer Science (4) (1997).
[24] E.W. Dijkstra, Self-stabilizing systems in spite of distributed control, Communications of the ACM 17 (11) (1974) 643–644.
[25] I. Stoica, H. Zhang, Providing guaranteed services without per-flow management, in: Proceedings of the ACM SIGCOMM Conference, 1999.
[26] Z. Zhang, Z. Duan, L. Gao, Y.T. Hou, Decoupling QoS control from core routers: a novel bandwidth architecture for scalable support for guaranteed services, in: Proceedings of the ACM SIGCOMM Conference, 2000.
[27] J. Kaur, H.M. Vin, Core-stateless guaranteed rate scheduling algorithms, in: Proceedings of the IEEE INFOCOM Conference, 2001.
[28] J. Kaur, H.M. Vin, Core stateless guaranteed throughput networks, in: Proceedings of the IEEE INFOCOM Conference, 2003.
[29] E. Rosen, A. Viswanathan, R. Callon, Multiprotocol Label Switching Architecture, RFC 3031 (Proposed Standard) (Jan. 2001). http://www.ietf.org/rfc/rfc3031.txt.
[30] J. Cobb, Preserving quality of service without per-flow state, in: Proc. IEEE International Conference on Network Protocols, ICNP, 2001.
[31] J. Cobb, Scalable quality of service across multiple domains, Computer Communications 28 (18) (2005) 1997–2008.