Stabilization of Max-Min Fair Networks without Per-Flow State

Jorge A. Cobb

Department of Computer Science, The University of Texas at Dallas, Richardson, TX 75083-0688

Mohamed G. Gouda

Department of Computer Science, The University of Texas at Austin, Austin, TX 78712-0233

Abstract

Let a *flow* be a sequence of packets sent from a source computer to a destination computer. Routers at the core of the Internet do not maintain any information about the flows that traverse them. This has allowed for great speeds at the routers, at the expense of providing only best-effort service. In this paper, we consider the problem of fairly allocating bandwidth to each flow. We assume some flows request a constant amount of bandwidth from the network. The bandwidth that remains is distributed fairly among the rest of the flows. The fairness sought after is max-min fairness, which assigns to each flow the largest possible bandwidth that avoids affecting other flows. The distinguishing factor to other approaches is that routers only maintain a constant amount of state, which is consistent with trends in the Internet (such as the proposed Differentiated Services Internet architecture). In addition, due to the need for high fault-tolerance in the Internet, we ensure our protocol is self-stabilizing, that is, it tolerates a wide variety of transient faults.

Key words: stabilization, max-min fairness, quality of service, computer networks

---

1. Introduction

As the Internet grows, scalability at the core of the Internet has become a significant concern. To provide simple best-effort service, core routers do not need to maintain any state information about the flows of packets that traverse them. To provide more advanced forms of quality of service, such as guaranteeing bandwidth or delay, the Differentiated Services Architecture [1, 2],

Email addresses: cobb@utdallas.edu (Jorge A. Cobb), gouda@cs.utexas.edu (Mohamed G. Gouda)

which maintains only a constant amount of state per router, is favored over the Integrated Services Architecture [3, 4], where each core router maintains state for each individual flow.

In this paper, we focus on providing fair bandwidth allocation among different flows in a core network. There are many different notions of fairness, and each of these leads to a different optimization objective. We adopt the notion of *max-min fairness*. A bandwidth allocation is max-min fair [5], if no flow can be allocated a higher bandwidth without hurting another flow having equal or lower bandwidth.

Max-min fairness satisfies many intuitive fairness properties, and it has been studied extensively [6, 7, 8, 9]. However, all of these proposed algorithms need-per flow state.

In this paper, we present a fault-tolerant distributed algorithm for the computation of max-min bandwidth allocations. Our algorithm only requires a constant amount of state information at each router.

Although constant-state algorithms have been presented earlier, [10, 11], they have disregarded fault tolerance altogether. Our algorithm is presented formally and is shown to be stabilizing, i.e., resilient against a wide-variety of transient faults.

2. 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 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 *process* in a system is specified by finite sets of constants, variables, and actions. The values of each variable are taken from some bounded domain of values. Each action of a process p is of the form

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

where $\langle \mathsf{guard} \rangle$ can be in one of three forms: a) local, b) receiving, or c) 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 executed when the clock of p has reached a certain value.

In the above action, $\langle \mathsf{assignment} \rangle$ 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.

i. *Maximality:* Either the sequence is infinite or it is finite and 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 [12, 13, 14].

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.

3. Network Model

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.

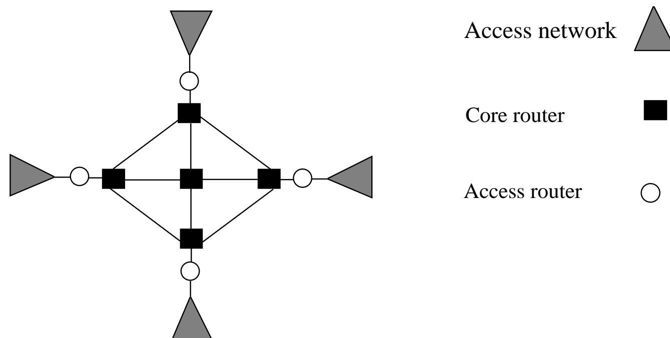

Figure 1: Core network.

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 [15, 16, 17, 18], 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 [19]. 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 an fixed upper bound on their bandwidth.

Each source probes the network to determine how much bandwidth it is allowed to use. Routers only keep 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

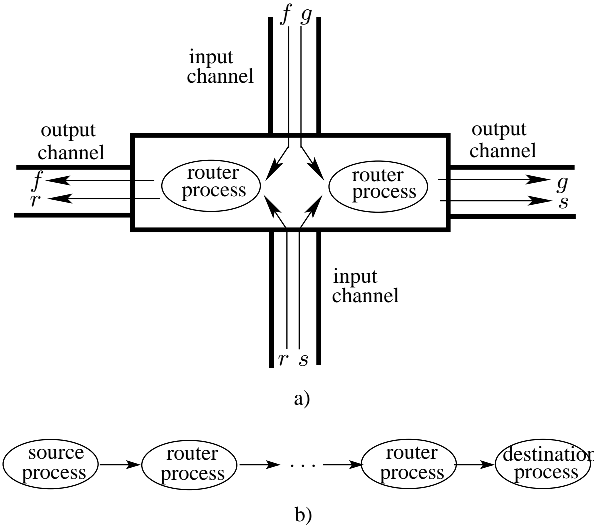

Figure 2: Processes and flows in a core router.

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.

We conclude by defining our fairness objective; we will consider max-min fairness [5], which is intuitively defined as follows: bandwidth is allocated to each flow so that an increase of the bandwidth allocated to any flow f must be done at the expense of decreasing the bandwidth of a flow g where the bandwidth allocated to g is smaller than that of f.

The bandwidth allocation to each flow can be defined iteratively as follows.

For each pair of neighboring processes p and q, we define the following variables:

- Let $B(p,q)$ initially have the bandwidth of channel $ch(p,q)$. B will contain the unallocated bandwidth of the channel.
- Let $F(p,q)$ be the set of adaptive flows traversing channel $ch(p,q)$. F will contain the set of flows whose bandwidth has not yet been determined.

The following steps are repeated until all flows have been assigned a bandwidth, i.e., until F is empty for all channels.

- Let (p,q) be channel such that

$$\frac{B(p,q)}{|F(p,q)|} = \min_{x,y} \left\{ \frac{B(x,y)}{|F(x,y)|} \right\}$$

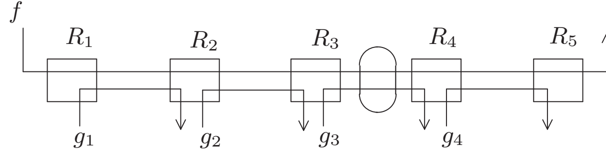

Figure 3: Max-Min Fairness example.

- For every flow $f \in F(p,q)$, assign to f a bandwidth of

$$\frac{B(p,q)}{|F(p,q)|}$$

- For every edge (x,y) other than (p,q),
  - Reduce $B(x,y)$ by the sum of the bandwidths of the flows in $F(p,q)$ that also traverse (x,y).
  - Remove from $F(x,y)$ any flow that is also in $F(p,q)$.
- $F(p,q)$ is assigned the empty set and $B(p,q)$ is assigned zero.

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.

4. Signaling

In this section, we present a signaling protocol. Its purpose is for each router process to aggregate information from each of its flows. 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 5.

Our protocol below is an abstraction of the protocol we presented in [20, 21]. In addition to being more general, it is strengthened to become stabilizing.

We assume that the set of flows in the network is fixed. Otherwise, a flow's fair share of the network bandwidth varies over time, and hence, the system never converges.

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.

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

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

For example, in Section 5, each router process will maintain 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 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 3, it is sent along the path of f with high priority and bounded round-trip time.

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 as the "shadow bit". Every T seconds, where T is a predefined constant, the router updates its state in the following way:

$$s, A, \widehat{A} := \neg s, \widehat{A}, 0 \qquad (*)$$

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.

The objective of the periodic *Aggregate* message is the following. For each flow f, the router should add f_v to $\widehat{A}$ once and only once before the update in (*) above is performed. In this manner, A will always contain the aggregate of the values of all flows.

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.

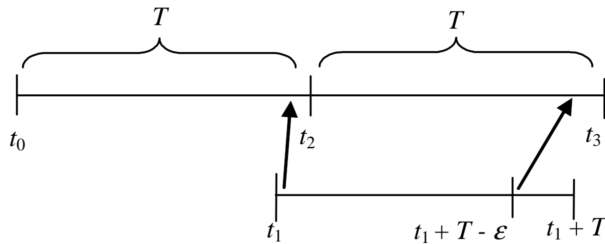

Figure 4: Timing of *Aggregate* messages

- 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.

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

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 routers has a corrupted value, it will also correct itself within $2 \cdot T$ seconds.

To adapt the signaling protocol to the max-min fairness protocol in Section 5, we add the following two enhancements:

(i) The router may choose whether or not to add the flow to its aggregate.

(ii) The source of a flow f may change its value f_v over time.

Note that routers do not maintain per-flow information, and thus, they are unaware for a given flow f whether its f_v is included in their aggregate or not. This fact will be stored at the source of the flow 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 ith router along the path of f.

Furthermore, since the source of f may change its value f_v, each *Aggregate* message will contain 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 of 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. interpacket time}

var

$\vec{s}$	:	**bit vector**	{$\vec{s}_R$ = shadow-bit at router R along the path}
$\vec{b}$	:	**bit vector**	{$\vec{b}_R$ = is v 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 msg is sent}

begin
  rcv $Ack(x, y, v, v', \vec{b}, \vec{s})$ $\rightarrow$
    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
```

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 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
  C     : integer   {channel bandwidth}
  T     : integer   {shadow interval}
var
  s     : boolean   {shadow bit}
  A, Â  : data      {data aggregate and its shadow copy}
  t     : integer   {time of last timeout}
begin
  rcv Aggregate(x, y, v, v', b⃗, s⃗)  →
        {add flow to shadow aggregate}
        Â := Â ⊕ v              if s⃗_R ≠ s ∧ b⃗_R;
        {update shadow bit before forwarding}
        s⃗_R := s;
        {remove old value from aggregate}
        A, Â := A ⊖ v, Â ⊖ v    if b⃗_R;
        {choose whether to add or not the new value}
        b⃗_R := any;
```

```
        A, Â := A ⊕ v', Â ⊕ v' if b⃗_R;
        send Aggregate(x, y, v, v', b⃗, s⃗) to y
[]
  rcv Ack(x, y, v, v', b⃗, s⃗)    →
        send Ack(x, y, v, v', b⃗, s⃗) to x
[]
  timeout clock > t + T    →
        s, A, Â := ¬s, Â, 0;
        t := clock;
[]
  clock < t   →   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', b⃗, s⃗)    →
        send  Ack(x, y, v, v', b⃗, s⃗) 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. Max-Min Signaling

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

Consider the algorithm to compute max-min fairness given in Sec. 3. In order to implement it, at each iteration we need to know, for each link, the number

of flows whose bandwidth has not been allocated, and the total bandwidth that remains unallocated on the link.

This suggests that the information we maintain at the router is as follows:

- The sum of the bandwidths of flows that are not bottlenecked at this router, that is, flows who cannot increase their bandwidth because another router is preventing them from doing so. We will denote this sum as A.

- The total number of flows that are bottlenecked at this router, denoted by n. The bandwidth allocated to these flows will be the total bandwidth C of the channel minus A above divided by the number of flows n. We denote this bottleneck bandwidth by B, i.e., $B = (C - A)/n$.

In order for this information to be updated at the routers, the source needs to know which router is its bottleneck router, what is the bottleneck bandwidth of that router, and inform all other routers of this limit on the flow's bandwidth. Furthermore, this information may change over time, as the system converges to a steady state.

We thus require sources to send an *Aggregate* message along the path to their destination. The message contains the rate r currently being used by the source, and whether the source is considered bottlenecked or not at each router. With this information, the router can determine which of the following four cases apply to the flow:

1. If the flow is bottlenecked at the router and its rate r is at least the bottleneck bandwidth B of the router ($r \geq B$), then the flow remains bottlenecked at the router, but its new rate should be decreased to B.
2. If the flow is bottlenecked at the router and $r < B$, then the flow should no longer be considered bottlenecked at this router. Thus, its bandwidth r is added to A, and the number of bottlenecked flows n at the router is decreased by one.
3. If the flow is not bottlenecked at this router and $r \geq B$, then the flow must become bottlenecked at this router. Hence, n increases by 1, and A decreases by r.
4. If the flow is not bottlenecked at this router, and $r < B$, then the state of the flow and the router remain the same.

Routers have to maintain the aggregate of the rate r of the flow (if not bottled at this router), and the sum of bottled flows, i.e., aggregate the value 1 for each bottled flow. Thus, we can use the signaling protocol of the previous section to maintain these 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 rates of the flow and their aggregates at the routers.

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

process *source x*

```
const
  y          :  process id               {destination}
var
  b⃗          :  bit vector               {bottleneck-bit vector}
  r, r′, r″  :  non-negative integer     {allocated rate}
  wait       :  boolean                  {waiting for an Ack}
begin
  rcv Ack(x, y, r, r′, r″, b⃗)   →
        r := r′;
        r′ := r″;
        wait := false
[]
  ¬wait  →
        send Aggregate(x, y, r, r′, ∞, b⃗) to y
        wait := true
end
```

A source contains a bitmap $\vec{b}$ (discussed above) and three bandwidth variables, r, r', r'', that are included in each *Aggregate* message. Variable r contains the current bandwidth of the source, i.e., this value has been added to the bandwidth sum A at each router. On the other hand, r' contains the updated bandwidth, that is, the new value that should be stored at the routers. Finally, r'' is initialized to infinity, and, as the *Aggregate* message traverses to the destination, r'' stores the minimum of the bottleneck bandwidths of the routers along the path.

In the first action, the source receives an *Ack* message. The values of r and r' are updated. The value of $\vec{b}$ is updated as a side effect of receiving the message.

The last action sends a new *Aggregate* message provided the previous *Ack* has been received.

```
process router R
const
  C   :  non-negative integer     {channel bandwidth}
var
  n   :  non-negative integer     {bottlenecked users}
  A   :  non-negative integer     {adaptive bandwidth}
begin
  rcv Aggregate(x, y, r, r′, r″, b⃗)   →
        {update rate if not bottled}
        A := A − r + r′                        if ¬b⃗_R;
        {unbottle the flow if necessary}
        n, A, b⃗_R := n − 1, A + r′, false      if b⃗_R ∧ r′ < B ∧ n > 1;
        {bottle the flow if necessary}
        n, A, b⃗_R := n + 1, A − r′, true       if ¬b⃗_R ∧ (r′ ≥ B ∨ n = 0);
```

{update r'' before forwarding}
$r'' := \min(r'', B)$;
send $Aggregate(x, y, r, r', r'', \vec{b})$ **to** y

▯

rcv $Ack(x, y, r, r', r'', \vec{b})$ $\rightarrow$
send $Ack(x, y, r, r', r'', \vec{b})$ **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 viceversa. In this action, B is defined as follows.

$$B = \frac{C - A}{n}$$

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 a *Ack* message.

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

6. Stabilization of Max-Min Fairness

We next present an overview of the stabilization properties of our system. Below, the notation $x.v$ represents variable v in process x.

The proof of the following theorem is given in Appendix B.

Theorem 1. *Let F^R be the set of sources whose flows traverse router R. Then, the system stabilizes to the following predicates.*

- *For every source x, $x.r$ equals the max-min fair bandwidth corresponding to this source.*
- *For every router R,*

$$R.A = \left(\sum x,\ x \in F^R \wedge \neg x.\vec{b}_R,\ x.r\right)$$

- *For every router R,*

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

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 4 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 operates 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 to 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.

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 on average the convergence time would not be too large. We plan to investigate this via a future simulation study.

References

[1] J. Heinanen, F. Baker, W. Weiss, J. Wroclawski, Assured forwarding phb groupInternet RFC 2597.

[2] V. Jacobson, K. Nichols, K. Poduri, An expedited forwarding phbInternet RFC 2598.

[3] R. Braden, D. Clark, S. Shenker, Integrated services in the internet architectureInternet RFC 1633.

[4] J. Wroclawski, Specification of controlled-load network element serviceInternet RFC 2211.

[5] J.-Y. L. Boudec, Rate adaptation, congestion control and fairnessHttp://ica1www.epfl.ch/PS_files/LEB3132.pdf.

[6] 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.

[7] A. Charny, An algorithm for rate allocation in a packet switching network with feedbackM.S. thesis, Massachusetts Institute of Technology.

[8] 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 IEEE Infocom, San Francisco, CA, 1998.

[9] 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.

[10] 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.

[11] Y. Kim, W. K. Tsai, M. Iyer, J. Ros, Minimum rate guarantee without per-flow information, in: ICNP '99: Proceedings of the Seventh Annual International Conference on Network Protocols, IEEE Computer Society, Washington, DC, USA, 1999, p. 155.

[12] A. Arora, M. Gouda, Closure and convergence: A foundation of fault-tolerant computing, IEEE Transactions on Software Engineering 19 (11) (1993) 1015–1027. doi:http://dx.doi.org/10.1109/32.256850.

[13] S. Dolev, T. Herman, Superstabilizing protocols for dynamic distributed systems, Chicago Journal of Theoretical Computer Science 1997 (4).

[14] E. W. Dijkstra, Self-stabilizing systems in spite of distributed control, Commun. ACM 17 (11) (1974) 643–644. doi:http://doi.acm.org/10.1145/361179.361202.

[15] I. Stoica, H. Zhang, Providing guaranteed services without per-flow management, in: Proc. of the ACM SIGCOMM Conference, 1999.

[16] 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: Proc. ACM SIGCOMM Conference, 2000.

[17] J. Kaur, H. M. Vin, Core-stateless guaranteed rate scheduling algorithms, in: Proc. of the IEEE INFOCOM Conf., 2001.

[18] J. Kaur, H. M. Vin, Core stateless guaranteed throughput networks, in: Proc. of the IEEE INFOCOM Conf., 2003.

[19] R. Callon, P. Doolan, N. Feldman, A. Fredette, G. Swallow, A. Viswanathan, A framework for multiprotocol label switchingInternet draft draft-ietf-mpls-framework-02.txt.

[20] J. Cobb, Preserving quality of service without per-flow state, in: Proc. IEEE International Conference on Network Protocols (ICNP), 2001.

[21] J. Cobb, Scalable quality of service across multiple domains, Computer Communications 28 (18) (2005) 1997–2008, Elsevier.

A. Correctness Proof of Signalling Protocol

As discussed earlier, routing between access networks is outside the scope of the paper. We simply assume that routing is stabilizing[1], and thus the routing tables converge to a sound and stable set of values.

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.*

We next consider the relationship between the rates of the sources and the information stored at the routers. First, 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 4, we have the following.

Lemma 3. *For every source x 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.*

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, 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 are replaced with the fields of the *Ack*. To simplify our proofs, $(x, y).f$ refers to field f in message $(x, y).msg$. If currently there is no message for the pair (x, y), then $(x, y).f$ refers to variable f at source x.

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

Also, we also refer to variable v at a process p with the expression $p.v$.

We next show the relationship between the shadow bits at a source and the value stored in the shadow aggregate variable in a router along its path.

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,*

$$R.A = (\oplus\, i :: R.A[x]) \tag{1}$$

$$R.\widehat{A} = \left(\oplus\, i :: R.\widehat{A}[x]\right) \tag{2}$$

Proof. Within T seconds, zero will be assigned to $\widehat{A}$, and hence also to all $\widehat{A}[]$, and (2) above holds. From this moment onward, every value added to or subtracted from $\widehat{A}$ is also added to or subtracted from the appropriate element of $\widehat{A}[]$. Hence, (2) will hold and continue to hold.

Within another T seconds, $\widehat{A}$ is assigned to A, and at this moment, (1) 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 $A[]$. Hence, (1) 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.

Lemma 5. *The system stabilizes to the following predicate: for every source-destination pair (x, y) and every router R along their path,*

$$\begin{array}{c} R.s = (x, y).\vec{s}_R \;\wedge\; R.\widehat{A}[x] = U_{x,R} \\ \vee \\ R.s \neq (x, y).\vec{s}_R \;\wedge\; R.\widehat{A}[x] = 0 \end{array} \tag{3}$$

where

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

Proof. We first show that predicate (3) above will hold within $2 \cdot T$ seconds.

Assume that the computation starts when $R.s = (x, y).\vec{s}_R$. Note that this remains true until the router updates its state and flips its shadow bit. That is,

as the message moves along the network, $(x, y).\vec{s}_R$ remains the same, even if it is received at router R, since the router and the message's value are the same. When the message arrives back at the source, $(x, y).\vec{s}_R$ is assigned to $x.s_R$, and by the definition of $(x, y).\vec{s}_R$, it now refers to $x.\vec{s}_R$. When the router eventually flips its shadow bit, $\widehat{A}[x]$ becomes zero, and (3) holds.

Assume instead that the computation starts when $R.s \neq (x, y).\vec{s}_R$. If the router flips its shadow bit, then $R.s = (x, y).\vec{s}_R$ holds, and we continue with the previous paragraph's argument. If it doesn't, within the next T seconds a message from source x is received, and the router sets the bit in the message to be equal to its own. Again, $R.s = (x, y).\vec{s}_R$ holds, and we continue with the previous paragraph's argument.

We next show that once predicate (3) above holds it will continue to hold, by showing that all actions preserve the predicate.

Source Actions

Consider the 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 of 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 (3).

Consider next the second action. Recall first that the fields of (x, y) are defined to be that of the source variables until the message is created. Furthermore, the only variables that changed in the source are v and v'. From (3), we have two cases to consider. When $R.s \neq (x, y).\vec{s}_R$, sending the message does not affect any value in (3). When $R.s = (x, y).\vec{s}_R$, then, before the message is created, from the definition of $U_{x,R}$, we have $R.\widehat{A}[x] = (x, y).v'$. After the message is sent, $(x, y).msg \in P_{x,R}$ holds, and from $U_{x,R}$ we require that $R.\widehat{A}[x] = (x, y).v$. This holds because v' is assigned to v before the message is sent.

Consider next the last action. This is a sanity action that will be executed only once in the system. Thus, it will not affect predicate (3).

Destination Actions

Consider the destination. It has only a single action that receives an *Aggregate* message and sends an *Ack* message. Thus, it has no effect on (3).

Router Actions

Consider the first action. We have two cases to consider. First, assume $R.s = (x, y).\vec{s}_R$ holds before the action. From (3), we have $R.\widehat{A}[x] = (x, y).v$ or zero depending on $(x, y).\vec{b}_R$. From the assignments of the action, first, $R.\widehat{A}[x]$ is set to zero (by subtracting v), and then the router nondeterministically chooses to add v' to $\widehat{A}[x]$, and the nondeterministic choice is stored in $\vec{b}_R$. Finally, the shadow bit in the message is set to that of the router. Hence, the first disjunct in (3) holds after the message is forwarded.

Next, assume $R.s \neq (x, y).\vec{s}_R$ holds before the action. In this case, from (3), $R.\widehat{A}[x] = 0$. Then, if $\neg\vec{b}_R$, then $R.\widehat{A}[x] = 0$ remains unchanged. If $\vec{b}_R$,

then v is added to $\widehat{A}[x]$ and in the next step it is subtracted from it. Hence, $R.\widehat{A}[k]$ remains zero. Then, the router chooses whether or not to add $(x,y).v'$ to $R.\widehat{A}[x]$, by nondeterministically updating $(x,y).\vec{b}_R$. Finally, the shadow bit in the message is set to that of the router. Hence, the first disjunct in (3) holds after the message is forwarded.

Consider now the second action. This action only moves an *Ack* message one step closer to the source, and hence it does not affect (3).

In the third action, the router performs a state update by flipping its shadow bit and assigning zero to $\widehat{A}$. In (3), we have two cases to consider. First, if $R.s = (x,y).\vec{s}_R$ holds before the action, then $R.s \neq (x,y).\vec{s}_R \wedge R.\widehat{A}[x] = 0$ holds after the action. Second, assume $R.s \neq (x,y).\vec{s}_R \wedge R.\widehat{A}[x] = 0$ holds before the action. We claim this cannot hold when the action is going to be executed. The reason is as follows.

From the last time the action was executed (and $\widehat{A}[x]$ was set to zero), due to Lemma 3, a message from the source has been received. After receiving a message from the source, $(x,y).\vec{s}_R = R.s$. Note that the source does not change the value of $(x,y).\vec{s}_R$, and the router does not change the value of $R.s$ until the third action gets executed. Thus, $(x,y).\vec{s}_R = R.s$ remains true until the action gets executed. Hence, the second disjunct in (3) cannot hold before the action is executed.

The last action is a sanity action that is executed only once in the router, and hence will not affect (3) above.

∎

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

Theorem 2. *The system stabilizes to the following predicate: for every source-destination pair (x,y) and every router R along its path,*

$$R.A[x] = U_{x,R} \tag{4}$$

PROOF. We first prove that if (4) 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 (4) holds.

Consider the actions of every process.

Source actions

Consider the 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 (4).

Consider next the second action. Again, $(x,y).f$ is defined to be variable f at the source until the message is created. From the definition of $U_{x,R}$, since the message will now be contained in $P(x,R)$, but was not in $P(x,R)$ before the action, then, because v is assigned v', then the value of $U_{x,R}$ does not change, and (4) continues to hold.

Consider next the last action. This is a sanity action that will be executed only once in the system. Thus, it will not affect predicate (4).

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 (4).

Router actions

Consider the first action. We first argue that the first disjunct of (3) 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.

Consider now the second action. This action only moves an *Ack* message one step closer to the source, and hence it does not affect (4).

In the 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 (3) we have

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

Since (3) 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 $R.A[i] = U_{x,R}$.

The last action is a sanity action that is executed only once in the router, and hence will not affect (4) above.

Note that, from the argument of the third action, (4) will hold after the third action is executed, and hence, the system stabilizes to (4).

■

B. Proof of Theorem 1

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

Corollary 1. *For every every router R,*

$$R.A = \left(\sum x, y : r \in P(x, y) \; : \; V_{x,R}\right) \quad (5)$$

$$R.n = \left(\sum x, y : r \in P(x, y) \; : \; W_{x,R}\right) \quad (6)$$

where

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

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 signalling 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 signalling, the router is aggregating two values from each source: its rate r, 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 signalling protocol of Section 4. 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.

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

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

We next show that $R.A$ 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 6. *The system stabilizes to the following predicate: for every router R, where $F^R \neq \emptyset$,*

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

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

PROOF. We assume we have reached a state where Corollary 1 already holds, and hence, $R.A$ 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 $R.A > R.C$, 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 $R.A$.

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

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

$R.A$ increases under two conditions: a) a bottled flow becomes unbottled, and b) a flow requests an increase in rate (i.e. $r' > r$). If a flow becomes unbottled, then from the definition of $R.B$, $R.A$ 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 $R.A < R.C$ after the increase, since otherwise, the flow would become bottled and $R.A$ would actually decrease. ∎

Lemma 7. *Let Φ be the following set of values.*

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

$$(x, y).r,\ (x, y).r',\ (x, y).r''$$

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

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

Consider first the actions of the sender. In the 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. In the 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.

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

Finally, consider a router process. It contains two actions. The second action is trivial (just forward an *Ack* message) so the minimum in Φ cannot decrease. The more complex case is the first action, because the values of router R get affected, in particular, $R.B$, and $(x, y).r''$ also becomes affected. Note that if $R.B$ does not decrease, then $(x, y).r''$ 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. $(x, y).r > (x, y).r'$). This decrease increases the value of $R.B$ as desired. If, in addition, the flow becomes bottled, then $R.B$ increases even more, as desired.

(b) The rate of an unbottled flow (x, y) increases (i.e. $(x, y).r \leq (x, y).r'$). This will reduce the value of $R.B$, and we must ensure its new value is not lesser than another value in Φ. We have two subcases.

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

   ii. The flow becomes bottled (i.e., $(x, y).r' \geq R.B$ after $R.A$ is updated): in this case, we have to consider the relationship between $(x, r).r$ and the old value of $R.B$, which we denote $R.B_{old}$.

      (1) If $(x, y).r \geq R.B_{old}$, 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 $(x, y).r < R.B_{old}$, then, after bottling the flow, the net effect is bottling a flow whose original rate is r (since the new rate r' is given back to the bottled flow bandwidth). Thus, the new value of $R.B$ is $(R.B_{old} \cdot n + r)/(n + 1)$. Since $r < R.B_{old}$, even though $R.B_{old}$ decreases, it cannot decrease beyond r, i.e., an existing value of Φ, as desired.

(c) An 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).

■

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 8. *The minimum value of Φ (as defined in Lemma 7) increases until it reaches the value B_1.*

PROOF. Let m be the current minimum value in Φ. From Lemma 7, 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 $(x, y).r$ 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 $(x, y).r$, 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 $R.A$ 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.

What remains to be shown is that $(x, y).r$, $(x, y).r'$, and $(x, y).r''$ will also increase to a value larger than m. Note that for every router R traversed by (x, y), $R.B$ increases beyond m and remains above m. The values of $(x, y).r$ and $(x, y).r'$ depend on $(x, y).r''$, 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 $(x, y).r$, $(x, y).r'$, and $(x, y).r''$, 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 be the set of flows who 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 $\{r, r', r''\}$ of flows in F_k, where $i \leq k \leq n$, union with the $R.B$ values for every router R whose set of flows, denoted F^R, is a subset of $\bigcup_{i \leq x \leq n} F_x$.

Proof. From Lemma 8, Φ_1 grows to at least B_0.

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 $R.A$ is less than B_1, which from Theorem 2 implies $(x, y).r$ or $(x, y).r'$ is less than B_1, which violates assumption on Φ_1. Also, $R.B$ can't be less than B_1, since this would also violate our assumption on Φ_1. Hence, $R.B$ remains at B_1.

Consider any flow (x, y) going through the router R defined above. From Lemma 8, each of $(x, y).r$,$(x, y).r'$ and $(x, y).r''$ are at least B_1. So whenever a message from (x, y) arrives at R, (x, y) will become bottlenecked. Hence, $(x, y).r''$ becomes equal to B_1. From $(x, y).r''$, $(x, y).r'$ becomes B_1, and then after another message is sent by (x, y), $(x, y).r$ becomes equal to B_1. Hence, all flows in F_1 will have rate values equal to B_1 and remain at B_1, bottlenecked, and without changing.

Since F_1 flows are fixed, and all other values in Φ are at least B_1, we can basically ignore flows in F_1, and repeat the argument of Lemma 8 to show that Φ_2 grows to B_2, and repeat the above argument to show that all flows in F_2 will be bottlenecked with rate B_2 and remain in this state.

The first part of the theorem follows by induction. The remaining two parts follow from Theorem 2.