

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

Jorge A. Cobb<sup>1</sup> and Mohamed G. Gouda<sup>2</sup>

<sup>1</sup> Department of Computer Science  
The University of Texas at Dallas  
cobb@utdallas.edu

<sup>2</sup> Department of Computer Science  
The University of Texas at Austin  
gouda@cs.utexas.edu

**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.

**keywords:** 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 advance forms of quality of service, such as guaranteeing bandwidth or delay, 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 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 [6–9]. 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.

The organization of this paper is as follows. Section 2 presents our notation and defines stabilization. We assume two types of flows in our system: rigid flows, whose bandwidth is constant, and adaptive flows, whose bandwidth is determined by the max-min algorithm. Section 4 presents our signaling protocol and how it is used to support rigid flows. Section 5 defines max-min fairness formally and introduces adaptive flows. The stabilization of our algorithm is discussed in Section 6. Finally, concluding remarks are given in Section 7.

## 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

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

where  $\langle \text{guard} \rangle$  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  $\mathbf{rcv} \ m$  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,  $\langle \text{assignment} \rangle$  is a sequence of assignment statements, each of which is of the form

$$x := E(y, \dots) \ \mathbf{if} \ P$$

where  $x$  is a variable in process  $p$ ,  $E$  is an expression of the same type as variable  $x$ , and  $y$  is 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

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

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  $s_{(k+1)}$ .

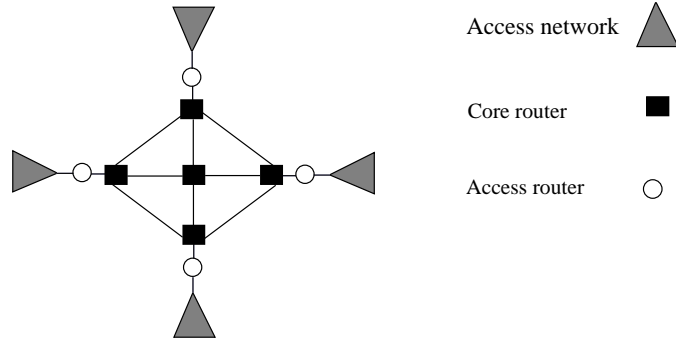
A *predicate*  $P$  of a system  $S$  is a boolean expression over the variables in all processes in system  $S$  and the contents of the channels in  $S$ .

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 [12–14].

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 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. 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*.



**Fig. 1.** Core network.

As it is commonly assumed [15–18], 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.

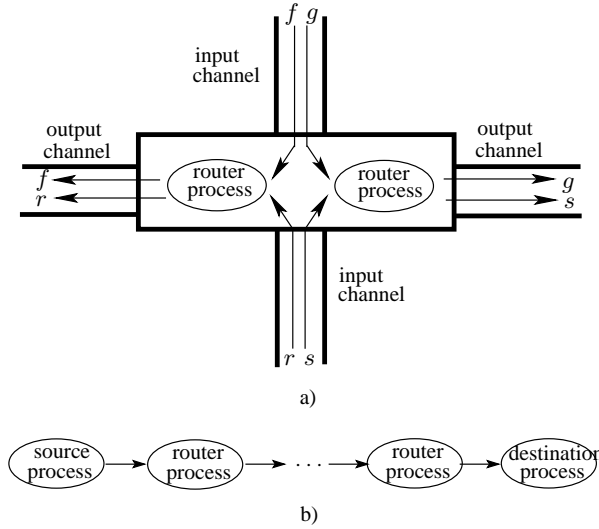
We model this by having three types of processes in our system: source processes, router processes, and destination processes. Each source process corresponds to the actions that an access router must perform for an individual flow. Thus, there are multiple source processes per access router, and each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

There are two types of sources: rigid and adaptive. A source is *rigid* if the bandwidth it reserves from the network is non-changing. A source is *adaptive* if it must probe 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. Through signaling messages, the sources are able to modify this aggregate information in order to maintain its accuracy.

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  $\varepsilon$  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.

We conclude by defining the fairness we expect to achieve for adaptive sources. 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  $ch(p, q)$  minus the bandwidth of the rigid flows traversing 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 edges.

- Let  $(p, q)$  be an edge such that

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

- 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  $(p, q)$ ,

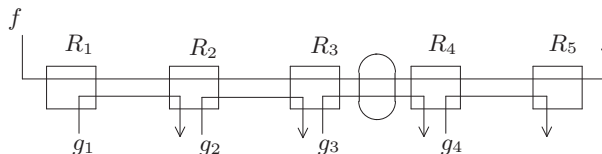
- Reduce  $B(x, y)$  by the sum of the bandwidths of the flows in  $F(p, q)$  that also traverse  $(x, y)$ .
  - Remove from  $F(x, y)$  any flow that is also in  $F(p, q)$ .
- $F(p, q)$  is assigned the empty set and  $B(p, p)$  is assigned zero.

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  $3 \cdot C/4$ .



**Fig. 3.** Max-Min Fairness example.

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

## 4 Rigid-Source Signaling

In this section, we present our signaling protocol, and show how it may be used by the rigid sources to reserve bandwidth from the network. It is a variation of a signaling protocol we presented in [20, 21] for a different network model. The protocol presented here however is strengthened to become stabilizing.

We make the following assumptions about the rigid sources:

- First, the set of rigid sources is assumed to be fixed. The reason for this requirement is that converging to a stable assignment of bandwidth to sources is not possible if the set of sources changes over time. We make this assumption also for the adaptive sources.

- Since the set of rigid sources are fixed, we do not address the steps required to setup/tear-down a source, and focus only on refreshing/correcting information at the routers. This is a practical assumption in some core networks, where flows would correspond to “data pipes” across the core, and the set of these pipes changes infrequently.
- We assume that the sum of the bandwidth requirements of all the rigid flows sharing a link is less than the bandwidth of the link.

As mentioned earlier, routers only maintain a constant amount of state information. Hence, each router maintains, for each of its output channels, the sum of the bandwidths of the rigid flows that traverse that channel. The remaining bandwidth of the channel will be distributed among the adaptive flows.

The objective of the signaling protocol is to maintain the above information current at each router, even though faults occur. For example, source processes may die, or the path between a source and its destination may change.

To maintain updated the state at each router along its path, each rigid source sends a *Reserve* message periodically. This message contains the desired bandwidth of its flow, and, as mentioned earlier, it is sent across the path with high priority and bounded round-trip time.

The router process maintains two variables,  $R$  and its “shadow copy”  $\widehat{R}$ . Variable  $R$  contains the sum of the bandwidth of the rigid flows. The router also maintains a boolean bit  $s$ , known as the “shadow bit”. Every  $T$  seconds, where  $T$  is a predefined constant, the router updates its state in the following way:

$$s, R, \widehat{R} := \neg s, \widehat{R}, 0$$

That is, the  $s$  bit is flipped, the shadow copy  $\widehat{R}$  is assigned to  $R$ , and the shadow copy  $\widehat{R}$  is cleared to zero.

The objective of the *Reserve* message is to add the bandwidth of the flow to  $\widehat{R}$  exactly once before the above assignments are done. In this way, the bandwidth of the flow will always be included in  $R$ . This is accomplished as follows.

The *Reserve* message contains a bit vector  $\mathbf{s}$ , with one bit for each router along the path of the flow. These bits are the last-known values of the  $s$  bit of each router along the path. The bandwidth of the flow is added to the shadow variable only if the state has been updated (and thus  $s$  has changed) from the time of the previous *Reserve* message of the flow. That is, the following two steps are performed at the  $i^{\text{th}}$  router whenever it receives a *Reserve*( $r, \mathbf{s}$ ) message, where  $r$  is the bandwidth of the flow.

- If  $\mathbf{s}_i \neq s$ , then, assign  $\widehat{R} + r$  to  $\widehat{R}$ , and assign  $s$  to  $\mathbf{s}_i$ .
- Forward the *Reserve*( $r, \mathbf{s}$ ) message along the next hop to the destination of the flow.

When the destination receives this message, it returns a *ReserveAck* message back to the source, containing the updated vector  $\mathbf{s}$ . A new *Reserve* message is not sent until an acknowledgment is received for the previous *Reserve* message.

We next address how often the source of a flow should send a *Reserve* message. As mentioned earlier, we assume a bound,  $\varepsilon$ , on the time for a

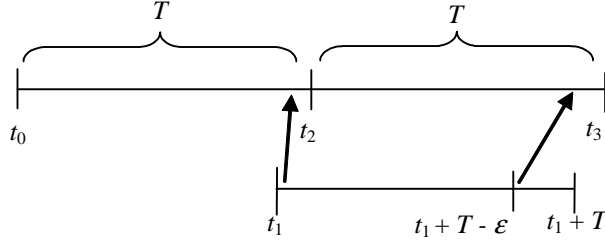


Fig. 4. Timing of *Reserve* messages

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  $t + \epsilon$ . State updates of different routers are not required to be synchronized. The only assumption is that each scheduler 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 a *Reserve* message, which arrives at the router in the interval  $(t_0, t_2)$ . Thus, at least one *Reserve* 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 *Reserve* message must arrive at the router no later than  $t_1 + T$ , i.e., it must be sent no later than  $t_1 + T - \epsilon$ . Furthermore, the next *Reserve* cannot be sent until a *ReserveAck* is received for the first *Reserve*, which at the latest will occur at time  $t_1 + 2 \cdot \epsilon$ . Thus, we require

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

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

The above signaling protocol is robust to a variety of faults. E.g., if a source dies, then all the bandwidth reservation from the source will be removed within  $2 \cdot T$  seconds, as follows. Within the first  $T$  seconds,  $\hat{R}$  is set to zero. Since the source has died, its bandwidth is never added to  $\hat{R}$ , and within the next  $T$  seconds,  $\hat{R}$  is assigned to  $R$ . Similarly, if the path of a source changes, routers along the previous path will remove information about the source in  $2 \cdot T$  seconds, while routers along the new path will add information about the source. If the information at the routers is incorrect, it will also correct itself within  $2 \cdot T$  seconds.

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

```

process src[ $i$ ]
const
   $r$       : integer      {data rate}

```

```

     $d$       : process id    {destination}
     $\varepsilon$  : integer      {max. e2e delay}
var
     $s$       : bit vector    {shadow-bit vector}
     $t$       : integer      {time msg is sent}
begin
    rcv  $ack\_f(i, d, s)$  → skip
□
    timeout  $clock \in [t + 2 \cdot \varepsilon, t + T - \varepsilon]$  →
        send  $Reserve(i, d, r, s)$  to  $dst[d]$ 
         $t := clock$ ;
□
     $t + T - \varepsilon < clock < t$  →  $t := clock$ ;
end

```

The source process contains two actions. In the first action, it receives a *ReserveAck* message, which has traversed the network from the destination back to the source. The only purpose of the message is to update the bit vector  $s$ , which is done as a side effect of receiving the message. Thus, the right-hand side of the action is empty.

The second action is a timeout action, in which a *Reserve* message is sent to the destination. Variable  $t$  stores the time at which the last *Reserve* message was sent. To ensure old *Reserve* and *ReserveAck* messages have left the network before sending a new one, *Reserve* 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  $router[i]$ 
const
     $C$       : integer    {channel bandwidth}
     $T$       : integer    {shadow interval}
var
     $s$       : boolean    {shadow bit}
     $R, \hat{R}$  : integer    {fixed bandwidth}
     $t$       : integer    {time of last timeout}
begin
    rcv  $Reserve(x, y, r, s)$  →
         $\hat{R} := \hat{R} + r$  if  $s_i \neq s$ 
         $s_i := s$ ;
        send  $Reserve(x, y, r, s)$  to  $dst[y]$ ;
□

```

```

rcv ReserveAck( $x, y, r, s$ )  $\rightarrow$ 
    send ReserveAck( $x, y, r, s$ ) to src[ $x$ ];
□
timeout  $clock > t + T$   $\rightarrow$ 
     $s, R, \widehat{R} := \neg s, \widehat{R}, 0;$ 
     $t := clock;$ 
□
 $clock < t$   $\rightarrow$   $t := clock;$ 
end

```

In the first action, a *Reserve* message is received, and is forwarded along the next hop to the destination. Before forwarding the message, the rate of the flow is added to the shadow variable  $\widehat{F}$ , provided a state change has occurred from the last time a *Reserve* message from this flow was received, i.e.,  $s_i \neq s$ . Also,  $s_i$  is updated to the value of  $s$  before forwarding the message. This ensures that the flow is counted only once in  $\widehat{F}$ .

In the second action, a *ReserveAck* 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,  $\widehat{R}$  is assigned to  $R$ ,  $\widehat{R}$  is set 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 dst[ $i$ ]
begin
    rcv Reserve( $x, i, r, s$ )  $\rightarrow$ 
        send ReserveAck( $x, i, r, s$ ) to src[ $x$ ];
end

```

It simply consists of a single action that receives a *Reserve* message and returns a *ReserveAck* in the direction of the source of the message.

## 5 Adaptive-Source Signaling

We next address how to modify the system to support adaptive sources. The system should converge to a state where all adaptive sources have been given their max-min fair share of the network bandwidth.

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 adaptive 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 adaptive 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 a *Probe* 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 greater than the bottleneck bandwidth  $B$  of the router ( $r > B$ ), 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 be no longer 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  $r > B$ , 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.

In order to refresh the information in a fault-tolerant manner, we also introduce shadow copies of  $n$  and  $A$ , i.e.,  $\hat{n}$  and  $\hat{A}$ . Furthermore, in order for the source to be aware of which routers consider it to be bottlenecked, each *Probe* message carries an additional bitmap  $\mathbf{b}$ , where  $\mathbf{b}_i$  is true if the flow is bottlenecked at router  $i$  along its path.

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

```

process src[i]
const
  d      : process id    {destination}
   $\varepsilon$  : integer      {min. interpacket time}
var
  s      : bit vector   {shadow-bit vector}
  b      : bit vector   {bottleneck-bit vector}
  r, r', r'' : integer   {allocated rate}
  t      : integer     {time msg is sent}

```

```

begin
  rcv ProbeAck(i, d, r'', s, b) →
    r := r';
    r' := r'';
  □
  timeout clock ∈ [t + 2 · ε, t + T - ε] →
    send Probe(i, d, r, r', ∞, s, b) to dst[d];
    t := clock;
  □
  t + T - ε < clock < t → t := clock;
end

```

An adaptive source has several more variables than a rigid source. It contains a bitmap  $\mathbf{b}$  (discussed above) and three bandwidth variables,  $r, r', r''$ , that are included in each *Probe* 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 *Probe* 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 a *ProbeAck* message. The values of  $r$  and  $r'$  are updated. The values of  $\mathbf{s}$  and  $\mathbf{b}$  are updated as a side effect of receiving the message.

The timeout action is similar to the timeout action of a rigid source, except that a *Probe* message is sent instead of a *Reserve* message. The last action is again, a corrective action for the value of  $t$ .

```

process router[i]
const
  C      : integer {channel bandwidth}
  T      : integer {shadow interval}
var
  s      : boolean {shadow bit}
  n, n̂  : integer {bottlenecked users}
  A, Â  : integer {adaptive bandwidth}
  R, R̂  : integer {fixed bandwidth}
  t      : integer {time of last timeout}
begin
  rcv Reserve(x, y, r, s) →
    R̂ := R̂ + r if si ≠ s
    si := s;
    send Reserve(x, y, r, s) to dst[y];
  □
  rcv ReserveAck(x, y, r, s) →
    send ReserveAck(x, y, r, s) to src[x];

```

```

□
rcv Probe(x, y, r, r', r'', s, b) →
  {add flow to shadow variables}
   $\hat{n} := \hat{n} + 1$       if  $s_i \neq s \wedge \mathbf{b}_i$ ;
   $\hat{A} := \hat{A} + r$       if  $s_i \neq s \wedge \neg \mathbf{b}_i$ ;
  {change flow from category if necessary}
   $n, \hat{n} := n - 1, \hat{n} - 1$     if  $r' < B \wedge \mathbf{b}_i$ ;
   $A, \hat{A} := A + r', \hat{A} + r'$     if  $r' < B \wedge \mathbf{b}_i$ ;
   $n, \hat{n} := n + 1, \hat{n} + 1$     if  $r' \geq B' \wedge \neg \mathbf{b}_i$ ;
   $A, \hat{A} := A - r, \hat{A} - r$     if  $r' \geq B' \wedge \neg \mathbf{b}_i$ ;
  {update values before forwarding}
   $s_i, \mathbf{b}_i := s, (r' \geq B)$ ;
   $r'' := \min(r'', B)$ 
  send Probe(x, y, r, r', r'', s, b) to dst[d]
□
rcv ProbeAck(x, y, r'', s, b) →
  send ProbeAck(x, y, r'', s, b) to src[x]
□
timeout clock > t + T →
   $s, n, A, R, \hat{n}, \hat{A}, \hat{R} := \neg s, \hat{n}, \hat{A}, \hat{R}, 0, 0, 0$ ;
   $t := clock$ ;
□
clock < t → t := clock;
end

```

The router contains seven actions. The first two are the same as before: they receive messages originating from rigid sources.

The last two actions are also similar to before. The last action restores the value of  $t$  to a sensible value, and the timeout action performs a state change of the router by assigning the shadow variables to their corresponding regular variables, and flipping the shadow bit.

In the third action, a *Probe* message is received. The first step consists of adding the bandwidth information of the flow to the shadow variables, provided the shadow bit indicates this is necessary. The second step consists of evaluating the four conditions mentioned above to ensure the flow is correctly placed in the bottlenecked or not bottlenecked category. In this action,  $B$  and  $B'$  are defined as follows.

$$B = \frac{C - A - R}{n} \quad B' = \frac{C - A - R - r}{n + 1}$$

The destination is similar to before; it receives a *Probe* message and returns a *ProbeAck* message.

```

process dst[i]
begin
  rcv Probe(x, i, r, r', r'', s, b) →
    send ProbeAck(i, d, r'', s, b) to src[x];
end

```

## 6 Stabilization of Max-Min Fairness

We next present an overview of the stabilization properties of our system. Detailed proofs will be available in [22]. Below, we refer only to *Probe* and *ProbeAck* messages of adaptive sources. Similar lemmas and theorems can be derived for messages from rigid sources. As discussed earlier, routing between access networks is outside the scope of the paper. We simply assume that routing is stabilizing<sup>3</sup>, and thus the routing tables converge to a sound and stable set of values. This, combined with the timing restrictions on sending messages, gives the following.

**Lemma 1.** *The system stabilizes to the following predicate: every  $Probe(x, y, \dots)$  message is located only along the path from  $x$  to  $y$ , and every  $ProbeAck(x, y, \dots)$  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  $Probe(x, y, \dots)$  messages plus the number of  $ProbeAck(x, y, \dots)$  messages is at most one.*

We next consider the relationship between the rates of the sources and the information stored at the routers. Before this, the following two lemmas are necessary. First, due to the timing of the state changes of the routers and the timing on the generation of signaling messages by the source we have the following.

**Lemma 3.** *Every computation of the system has a suffix such that the following holds. In every state  $u_i$  of the suffix, if the shadow bit of a router at state  $u_i$  differs from its value at a later state  $u_j$ , then the router has received a *Probe* message between  $u_i$  and  $u_j$  for every adaptive source that traverses the router.*

Due to the above, we obtain the following relationship of the shadow bits of messages, routers, and sources.

**Lemma 4.** *The system stabilizes to the conjunction of the following predicates:*

- if there exists a  $Probe(x, y, \dots, \mathbf{s}, \dots)$  message along the  $i^{\text{th}}$  hop of the path from source  $x$  and destination  $y$ , then,
  - $\langle \forall j, (src[x].s_j = router[j].s) \Rightarrow (Probe.s_j = router[j].s) \rangle$ ,
  - $\langle \forall j, (src[x].s_j \neq router[j].s \wedge i \leq j) \Rightarrow (Probe.s_j = src[x].s_j) \rangle$ ,
  - $\langle \forall j, (Probe.s_j \neq src[x].s_j) \Rightarrow (i > j \wedge Probe.s_j = router[j].s) \rangle$ ,
 where  $router[j]$  is the  $j^{\text{th}}$  router along the path from source  $x$  to destination  $y$ .

<sup>3</sup> Most routing protocols such as link-state routing and distance-vector routing are in essence stabilizing.

- if there exists a  $ProbeAck(x, y, \dots, \mathbf{s}, \dots)$  message along the path from destination  $y$  back to source  $x$ , then

$$\langle \forall j, (ProbeAck.s_j \neq src[x].s_j) \Rightarrow (ProbeAck.s_j = router[j].s) \rangle$$

From the above, we can derive the relationship between the aggregate bandwidth information at the routers and the bandwidth information of each individual source, as follows.

**Theorem 1.** *Let  $S(i)$  be the set of adaptive sources whose flows traverse router  $i$ . Let  $\mathbf{b}(x), \mathbf{s}(x), r(x), r'(x)$  be the fields in the Probe and ProbeAck messages of source  $x$ , and if neither of the two messages are in transit, then these values correspond to the variables of the source.*

*Then, the system stabilizes to the following predicate. For all  $i$ ,*

- $router[i].A = (\sum x, x \in S(i), \alpha(x) \cdot r(x) + \alpha'(x) \cdot r'(x))$ , and
- $router[i].n = |\{x, x \in S(i) \wedge \mathbf{b}(x)_i\}|$ , and
- $router[i].\hat{A} = (\sum x, x \in S(i), \hat{\alpha}(x) \cdot r(x) + \hat{\alpha}'(x) \cdot r'(x))$ , and
- $router[i].\hat{n} = |\{x, x \in S(i) \wedge \mathbf{b}(x)_i \wedge \mathbf{s}(x)_i = router[i].s\}|$

where

- $\alpha(x) = 1$  if  $\neg \mathbf{b}(x)_i$  and either there is a Probe message along the path from  $src[x]$  to  $router[i]$  or there is no message from  $src[x]$  in the network. It is zero otherwise.
- $\alpha'(x) = 1$  if  $\neg \mathbf{b}(x)_i$  and either there is a Probe message along the path from  $router[i]$  to the destination of  $src[x]$ , or there is a ProbeAck message along the path from the destination back to  $src[x]$ . It is zero otherwise.
- $\hat{\alpha}(x) = 1$  if  $\alpha(x) = 1 \wedge (router[i].s = \mathbf{s}(x)_i)$ . It is zero otherwise.
- $\hat{\alpha}'(x) = 1$  if  $\alpha'(x) = 1 \wedge (router[i].s = \mathbf{s}(x)_i)$ . It is zero otherwise.

Finally, the bandwidth values must converge to the max-min allocation for each flow. The first lemma serves as a stepping stone for an induction proof leading to the main theorem.

**Lemma 5.** *Let  $B_0$  be the bandwidth assigned to the first set of flows in the max-min algorithm. Then, every computation has a suffix where all of the following hold.*

- For any  $i$ ,  $router[i].B \geq B_0$ .
- For any  $i$ , each of  $src[i].r, src[i].r', src[i].r''$  are at least  $B_0$ .
- For each Probe message, each of  $Probe.r, Probe.r', Probe.r''$  are at least  $B_0$ .
- For each ProbeAck message,  $ProbeAck.r'' \geq B_0$ .

**Theorem 2.** *Let  $S^A(i)$  and  $S^R(i)$  be the set of adaptive and rigid sources, respectively, whose flows traverse router  $i$ . Then, the system stabilizes to the following predicate. For all  $i$  and  $j$ ,*

- if  $src[j]$  is an adaptive source, then  $src[j].r$  equals the max-min fair bandwidth corresponding to the source, and
- $router[i].R = (\sum x, x \in S^R(i), src[x].r)$ , and
- $router[i].A = (\sum x, x \in S^A(i) \wedge \neg src[x].\mathbf{b}_i, src[x].r)$ , and
- $router[i].n = |\{x, x \in S^A(i) \wedge src[x].\mathbf{b}_i\}|$

## 7 Concluding Remarks

Above, we did not discuss the stabilization time of our system. The stabilization predicate of Theorem 1 can be shown to stabilize in  $O(T)$  time, where  $T$  is the interval between state changes at a router.

The stabilization time of Theorem 2, 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  $\Delta$  is the time interval between signaling messages from a source. We have shown in Sec. 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  $\Delta$ .

## References

1. J. Heinanen, F. Baker, W. Weiss, and J. Wroclawski, "Assured forwarding phb group," Internet RFC 2597.
2. V. Jacobson, K. Nichols, and K. Poduri, "An expedited forwarding phb," Internet RFC 2598.
3. R. Braden, D. Clark, and S. Shenker, "Integrated services in the internet architecture," Internet RFC 1633.
4. J. Wroclawski, "Specification of controlled-load network element service," 1997, Internet RFC 2211.
5. J.-Y. L. Boudec, "Rate adaptation, congestion control and fairness," 2008, [http://ica1www.epfl.ch/PS\\_files/LEB3132.pdf](http://ica1www.epfl.ch/PS_files/LEB3132.pdf).
6. S. Abraham and A. Kumar, "A stochastic approximation approach for max-min fair adaptive rate control of abr sessions with mcrrs," in *Proceedings of IEEE INFOCOM*, New York, NY, March 1999.
7. A. Charny, "An algorithm for rate allocation in a packet switching network with feedback," May 1994, M.S. thesis, Massachusetts Institute of Technology.
8. Y. T. Hou, H. H. Y. Tzeng, and 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, March 1998.
9. J. Ros and 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, and 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, and J. Ros, "Minimum rate guarantee without per-flow information," in *ICNP '99: Proceedings of the Seventh Annual International Conference on Network Protocols*. Washington, DC, USA: IEEE Computer Society, 1999, p. 155.
12. A. Arora and M. Gouda, "Closure and convergence: A foundation of fault-tolerant computing," *IEEE Transactions on Software Engineering*, vol. 19, no. 11, pp. 1015–1027, 1993.

13. S. Dolev and T. Herman, "Superstabilizing protocols for dynamic distributed systems," *Chicago Journal of Theoretical Computer Science*, vol. 1997, no. 4, 1997.
14. E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Commun. ACM*, vol. 17, no. 11, pp. 643–644, 1974.
15. I. Stoica and H. Zhang, "Providing guaranteed services without per-flow management," in *Proc. of the ACM SIGCOMM Conference*, 1999.
16. Z. Zhang, Z. Duan, L. Gao, and 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 and H. M. Vin, "Core-stateless guaranteed rate scheduling algorithms," in *Proc. of the IEEE INFOCOM Conf.*, 2001.
18. —, "Core stateless guaranteed throughput networks," in *Proc. of the IEEE INFOCOM Conf.*, 2003.
19. R. Callon, P. Doolan, N. Feldman, A. Fredette, G. Swallow, and A. Viswanathan, "A framework for multiprotocol label switching," 1997, Internet 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)*, Nov. 2001.
21. —, "Scalable quality of service across multiple domains," *Computer Communications*, vol. 28, no. 18, pp. 1997–2008, Nov. 2005, Elsevier.
22. J. A. Cobb and M. G. Gouda, "Stabilization of max-min fair networks without per-flow state," Sept. 2008, Department of Computer Science Technical Report, The University of Texas at Dallas.