Converging to Approximated Max-Min Flow Fairness in Logarithmic Time

Baruch Awerbuch *Yuval Shavitt*[*]
{baruch, shavitt}@cs.jhu.edu
Department of Computer Science
Johns Hopkins University
Baltimore, MD 21218

June 26, 1997

Abstract

Max-min is a frequently praised criterion for flow control although its limitations. In practice, the fast rate of changes in the connection lay-out in modern networks makes it hard for any flow control algorithm to converge to an optimal point. It such an environment, it might be better to trade accuracy with speed.

We present algorithms that relax the optimality criterion of the max-min flow fairness but achieve a fast convergence time that is logarithmic in the link bandwidth and not a function of the number of active connections (or sessions). The relaxation we use requires rates to be increased or decreased by a certain factor, $1+\varepsilon$, or in other words, assigned rates can only be a natural power of some basic bandwidth $1+\varepsilon$. Under this criterion, the quiescence time of our flow control algorithms is $\log_{1+\varepsilon} B$, where B is the maximum link bandwidth in minimum allocation units. This is a great improvement over the super-linear quiescence time of known algorithms both exact and approximated.

keyword: flow control, ABR, max-min fairness

[*]Corresponding author

1 Introduction

Modern communication networks transfer traffic in a connection-oriented fashion. The connection between a source-destination pair is called a session and is assigned a route and bandwidth in the network. However, when many sessions are routed through the same links their bandwidth allocation should be controlled to prevent over utilization of the link which cause buffer overflow and data-packet loss.

Flow control algorithms [GK80, Jaf81, Kes91] are designed to prevent congestion from occurring by assigning bandwidth to sessions on a timely basis. Two main technologies where flow control is in great need are ATM ABR traffic flow control [BF95, AMO96a, AMO96b, CRL96, FMS97] and the control of TCP streams [Jac88, BP95]. A common criterion to assign bandwidth called max-min flow fairness [Jaf81]. The criterion allows a rate increase for a session providing this increase does not cause a decrease in the rate of a session with lower or equal rate.

In practice, the lay-out of connections in modern networks is constantly changing due to arrival of new connections, termination of existing ones, and, some times, rerouting of existing connections. In this environment, reaching an optimal working point is a Sisyphean task never to be achieved. Instead, the best a network flow control algorithm can hope for is to converge the network fast enough to a state which is close enough to the optimal state.

In this paper, we present algorithms that relax the optimality criterion of the max-min flow fairness but achieve a fast convergence time that is logarithmic in the link bandwidth and is not a function of the number of active connections (or sessions) or the network size. The relaxation we use requires rates to be increased or decreased by a certain factor, $1+\varepsilon$, or in other words, assigned rates can only be a natural power of some basic bandwidth $1+\varepsilon$. Under this criterion, the quiescence time of our flow control algorithms is $\log_{1+\varepsilon} B$, where B is the maximum link bandwidth in minimum allocation units. This is a great improvement over the super-linear quiescence time of most known algorithms both exact and approximated [AMO96a, FMS97]. We show in the appendix that when the number of sessions is sufficiently large (more than ε^{-1} in each link) not more than one change in allocation per session is needed, and thus the quiescence time is constant in a highly loaded network.

For example, in a network with link capacity of 155Mbps and a minimum allocation unit of 64Kbps, $B = 2424$. If we select $\varepsilon = 0.05$ (5%) then the quiescence time of our flow control algorithms is $c \cdot 160$, were c is some small constant, regardless of the network size and the number of connections. This is several orders of magnitude less than an algorithm with quiescence time that is linear in the number of connections, that is measured in millions and more.

A similar approach was presented by Charny et al. [CRL96] for explicit rate schemes in ATM.

However, our approach is simpler as the protocol works in a pure asynchronous fashion and the switches synchronize through the protocol messages. To the contrary, fine tuning of calculation cycles should be made to achieve good convergence time in [CRL96]. The general approach we took make our algorithm implementable in both explicit rate schemes, e.g., ATM explicit rate, and in networks where the only signaling is congestion indication such as control of TCP connections or ATM relative rate marking. In section 6, we show how to adapt our algorithm to the ATM relative rate marking.

We present two algorithms, the first requires only constant space per link (each node maintain the links emanating from it), but when session rates need to be decreased, e.g., due to the addition of new sessions, the algorithm must reset and start from scratch. Although theoretically this is not a problem, as this process requires only a logarithmic number of time steps, it might not be desired in practice due to the high-changing nature of the connection lay-out. To this end we present a second algorithm that requires logarithmic space per link but allows both increase and decrease adjustments in the rates due to changes in the lay-out of the sessions or due to arrival or tear-down of sessions.

The rest of the paper is organized as follows. In the next section, we give the model and explain our approximation. In sections 3 and 4 we present a constant space and a logarithmic space algorithms, correspondingly. In section 5 we prove the convergence time of the logarithmic-space algorithm, in section 6 we describe an ATM implementation, and in section 7 we present simulation results of our algorithms. We present our concluding remarks in the last section.

2 Model

We model the network as a directed graph $G = (V, E)$, where each vertex $v \in V$ represent a network node, and each edge $e \in E$ represents a network link. Associated with each link $e \in E$ is a capacity $cap(e) \leq B$ which is a real number.

A set of n sessions $S = \{S_1, \ldots, S_n\}$ is laid out in the network, where session S_i is a set of links that constitutes a simple path in G between a source and a destination. Allocated to each session is a bandwidth or rate, denoted $rate(S_i)$.

We divide the sessions in a network to equivalence classes called clusters using the relation "sharing a link". Note that in most practical cases we expect the entire session set to form a single equivalence class.

Control communication between the session sources and the network nodes is via periodic refresh messages, $MSG(\cdot)$, that are generated by each source, traverse the session route where the nodes may change the message content, and are sent back from the destination to the source along some arbitrary route. We assume no other means of communication unless otherwise specified.

A change of a session rate is called an *update*. Unlike [AMO96a, FMS97] updates are not required to be atomic, and can be done concurrently for several sessions.

We approximate max-min flow allocation by requiring all the components of the max-min vector to be non-negative integer powers of $1+\varepsilon$. Next we describe how a centralized algorithm for computing accurate max-min fairness can be modified to achieve the approximated allocation. A centralized algorithm to achieve an accurate max-min vector contains the following steps [BG92, section 6.5.2]:

C.1 Calculate the fare share on all non-saturated links.

C.2 Select the most congested link and assign a fare share to the sessions sharing this link.

C.3 Return to step 1 if not done

To achieve the approximated max-min vector we change the calculation in step C.1 to consider only non-negative integer powers of $1+\varepsilon$. Instead of selecting a single link as in line C.2, we assign a fare share to sessions passing through any of the links that achieve the minimal fare share in the last calculation.

To demonstrate our approximation consider the example of figure 1. In this example six sessions use route r_1, one session uses route r_2, one session uses route r_3, three sessions use route r_4, and one session uses route r_5. The link capacities are shown in the figure (8, 8, and 5). For calculation simplicity, assume $\varepsilon = 1$. The exact max-min vector for this example is $v^* = (\frac{8}{7}, \frac{8}{7}, \frac{8}{7}, \frac{8}{7}, \frac{8}{7}, \frac{8}{7}, \frac{8}{7}, \frac{12}{7}, \frac{12}{7}, \frac{12}{7}, \frac{12}{7}, \frac{23}{7})$.

Applying the centralized modified algorithms, we get in the first iteration the fair share values $\lfloor \log_2 \frac{8}{7} \rfloor = 1$, $\lfloor \log_2 \frac{8}{5} \rfloor = 1$, and $\lfloor \log_2 \frac{5}{2} \rfloor = 2$ for links (A,B), (B,C), and (C,D), correspondingly. As a result, rate 1 is allocated to the session sharing links (A,B) and (B,C) and the session that uses route r_5 receives rate 4 in the next iteration, yielding the vector $\bar{v} = (1,1,1,1,1,1,1,1,1,1,1,4)$. Note that the vector $v_1 = (1,1,1,1,1,1,1,2,2,2,2,2)$ is a better vector for this lay-out since lexicographically it is greater. The algorithms presented here achieve, at least, the vector obtained by running the above modification to the centralized algorithm. However, they may achieve a different vector that is superior (larger) to the resulted approximated vector in a lexicographical comparison. In the example of figure 1 our algorithm is guaranteed to achieve $\bar{v}$ but may achieve v_1.

3 A constant-space- log-time algorithm

The constant-space algorithm assumes that, initially, a set of n sessions is laid out in the network, and no bandwidth is allocated. To allocate bandwidth to the sessions following the approximate max-

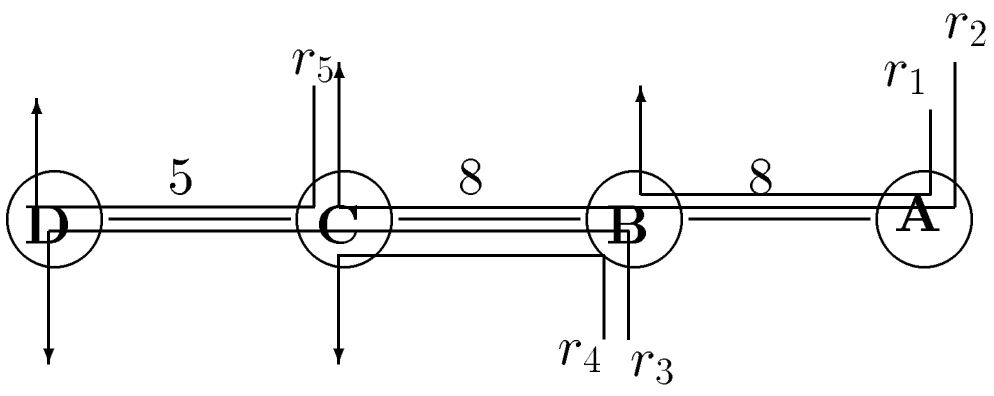

Figure 1: An approximate max-min fairness criterion example.

min fairness criterion, the algorithm tries to imitate the operation of the centralized approximated algorithm by working in cycles. It starts by granting each session a unit of bandwidth. In each successive cycle, if the reminder of the bandwidth is sufficient, the bandwidth of all the sessions is updated to be $1+\varepsilon$ times the previous bandwidth. In other words, all sessions starts with *level* 0 allocation, and in each cycle their allocation level is increased by one. Sessions whose bandwidth cannot be increased since they reached their fair share allocation (up to an ε) are *blocked* and their allocation stop changing. Clearly, if global synchronization among the sessions' rate increases is kept through the cycle mechanism, an approximated max-min fair allocation can be reached in $\Theta(\log B)$ time (taking the longest end-to-end propagation delay to be one time unit) and $\Theta(\log B)$ update operations per session.

For clarity, we first present an algorithm that uses linear space, and then show how to simulate its operation with only constant space. For the linear-space algorithm, each edge[1] maintains three lists of sessions:

$\mathcal{B}$ Blocked sessions. Sessions whose rate cannot be increased due to congestion in some other link.

$\mathcal{U}$ Unsynchronized sessions. Sessions that are at a lower level, lagging in the synchronization process.

$\mathcal{S}$ Synchronized sessions. Sessions that are at current level, and wait to synchronize with unsynchronized sessions.

In addition, each edge maintains CL, a level mark indicating the higher level it is ready to grant bandwidth permissions. Initially CL is set to 0, meaning that sessions can be granted basic unit allocation.

[1]The algorithm is performed by the node the link (or edge) emanates from. The 'edge' is the corresponding copy of the algorithm that is executed in the node.

Sessions try to increase their bandwidth allocation by sending periodic messages $MSG^s(level,\ flag)$, where s is the session id (sometimes omitted in the presentation), *level* is the current session allocation level, and *flag* can be either, *pending*, *blocked*, or *stalled*.

A session starts its activity by sending $MSG(-1,\ pending)$. If all the edges along the message route agree to accept the message it is forwarded unchanged, the destination sends the message $MSG(-1,\ pending)$ back to the source. The source increases its bandwidth allocation to level 0, and start transmitting $MSG(0,\ pending)$. An edge that receives a message $MSG^s(l,\ pending)$ with $l < CL$ forwards the message and puts s in $\mathcal{U}$. If $l = CL$ s is moved to $\mathcal{S}$. If $\mathcal{U}$ becomes empty, and sufficient capacity exists, the edge increases its CL counter and forwards the message. If $\mathcal{U}$ is not empty, the message is forwarded with the flag *stalled* to indicate that synchronization is not achieved, yet. If $l = CL$ reached the fair share of the link, all messages are forwarded with the flag *blocked* to signal the source about this situation. The session source starts transmitting messages with the *blocked* flag to free sessions in less congested edges the session traverses from synchronizing with the blocked session. A message that arrives with its flag *blocked*, inserts its session to $\mathcal{B}$ and thus not considered for synchronization. A source periodically sends $MSG(l,\ pending)$ where l is its current allocation level until it receives a *blocked* flag. At this point it starts transmitting $MSG(l,\ blocked)$.

Note that if a session starts a little later than others, it blocks all synchronization of sessions in its cluster until it reaches their level. This is an improvement over TCP where if two sessions start sharing a link with different bandwidth, they may never equalize their bandwidth allocation [Jac88]. However, if a session starts too late some sessions may receive an inappropriate allocation before the late session had a chance to stop the over-allocation. To identify this situation a warning field can be added to the periodic message. If a session is lagging behind the others in a certain edge (line n.6 in figure 2) an overflow condition can be checked namely if $\sum_{t\in\mathcal{B}\cup\mathcal{U}\cup\mathcal{S}} rate(t) + \varepsilon rate(s) > cap(e)$ the warning flag is set. If a message with a warning marking arrives at the source with its flag equals *pending* the algorithm reset and starts from scratch. Note that this reset process does not impact the algorithm quiescence time as the time measurement starts from the last change in the session set.

For achieving constant space consumption[2] in the edges we require that only one control message per session will be traveling in the network at a time, i.e., a source can send a control message only after it receives the previous one. The source need to send more information in the message that contains now five fields and has the form $MSG(level,\ lchange,\ flag,\ fchange,\ newflag)$, where *level* is the current session level, *lchnage* is a bit that indicates whether a level increase occurred due to the previous message reception, *flag* indicates if the session is *blocked* or not, *fchange* is a bit

[2]A constant amount of memory cells each should be $\log_2 B$ bits.

For $MSG^s(l,\ f)$
  $rate(s) \leftarrow (1+\varepsilon)^l$
  if $f = blocked$
        $\mathcal{B} \leftarrow \mathcal{B} + \{s\}$
        send $MSG^s(l,\ f)$
  else if $l < CL$
        $\mathcal{U} \leftarrow \mathcal{U} + \{s\}$
        send $MSG^s(l,\ f)$
  else if $l = CL$
        $\mathcal{S} \leftarrow \mathcal{S} + \{s\}$
        if $|\mathcal{U}| > 0$
                send $MSG^s(l,\ stalled)$
        if $|\mathcal{U}| = 0$ AND $\sum_{t\in\mathcal{S}} rate(t) * (1+\varepsilon) + \sum_{t\in\mathcal{B}} rate(t) > cap(e)$
                send $MSG^s(l,\ blocked)$
        if $|\mathcal{U}| = 0$ AND $\sum_{t\in\mathcal{S}} rate(t) * (1+\varepsilon) + \sum_{t\in\mathcal{B}} rate(t) \leq cap(e)$
                send $MSG^s(l,\ pending)$
                $\mathcal{U} \leftarrow \mathcal{S}$
                $\mathcal{S} \leftarrow \emptyset$
                $CL \leftarrow CL + 1$

Figure 2: A linear-space algorithm for link e of a network node.

For $MSG^s(l,\ f)$
  $rate(s) \leftarrow (1+\varepsilon)^l$
  if $f = pending$
        $level \leftarrow level + 1$
        $state \leftarrow pending$
  if $f = stalled$
        $state \leftarrow pending$
  if $f = blocked$
        $state \leftarrow blocked$
For time-out
  send $MSG^s(level,\ state)$

Figure 3: A linear-space algorithm for a session source.

that indicates if the session has become blocked in the previous round, and $newflag$ is used to signal to the source if the session is *blocked* or unsynchronized. $newflag$ is the only part of the message that can be altered by intermediate nodes.

Additional variables that are maintained in the edges are: CL its current synchronization and allocation level; B_B, U_B, S_B accumulated bandwidth holders for the three types of sessions, *blocked*, *unsynchronized*, and *synchronized*, respectively; and U_N, S_N counters for the number of of sessions in the *unsynchronized*, and *synchronized* classes, respectively. A formal description of the constant-space algorithm appears in figures 4 and 5.

4 A log-space- log-time algorithm

The algorithm presented in the previous section requires a reset operation whenever there is a need to decrease the bandwidth of some session, e.g., due to the addition of a session to a bottleneck link. In this section, we present an algorithm that has the same convergence time but enables us to 'role back' when sessions are added. In other words, the algorithm can run constantly and adapt the rate of the sessions to the changing pattern of session lay-out in the network, ensuring logarithmic quiescence time. The memory consumption of the algorithm in each edge is logarithmic in the capacity of the link and independent of the number of sessions in the network.

The algorithm operates (see figures 6 and 7) similarly to the constant-space algorithm. For the additional 'role-back' feature it uses $2\log B$ counters that keep track of the number of sessions in each level. We separately count blocked sessions from active ones, hence the factor of two. When a rate change is required, it is done as before in a synchronized way. A session is not allowed to increase its rate more than one level above any other active (i.e., not blocked) session in its cluster, or to decrease its level less than one level below any other session whether active or blocked.

The requirement for synchronization when rates are decreased might not seem necessary. The deletion of the second condition in line nl.16 allows sessions to decrease their rates without synchronization at the risk of rates decreasing too much and then readjusted upwards. The algorithm thus pushes the network out of an overload condition faster, and the quiescence time remains logarithmic. However, lowering the rate of a session below its final rate might cause unnecessary temporary rate change in other sessions that share links with this session, thus we do not recommend to abandon synchronization before these effects are studied.

A formal description of the algorithm is given in figures 6 and 7. The algorithm uses C a bidimensional array to count the number of *blocked* and active (termed *pending*) sessions in each level. When

```
For MSG(l, lc, f, fc, nf)
 if f = blocked
     if fc
         B_B ← B_B + (1 + ε)^l
         if l < CL
             U_B ← U_B − (1 + ε)^(l−lc)
             U_N ← U_N − 1
         if l = CL
             S_B ← S_B − (1 + ε)^(l−lc)
             S_N ← S_N − 1
     send MSG(l, lc, f, fc, nf)
 else if l < CL
     if lc
         U_B ← U_B + ε(1 + ε)^(l−1)
     send MSG(l, lc, f, fc, nf)
 else if l = CL
     if lc
         U_B ← U_B − (1 + ε)^(l−1)
         U_N ← U_N − 1
         S_B ← S_B + (1 + ε)^l
         S_N ← S_N + 1
     if U_N > 0
         if nf ≠ blocked
             send MSG(l, lc, f, fc, stalled)
         else
             send MSG(l, lc, f, fc, blocked)
     if U_N = 0 AND B_B + (1 + ε)S_B > cap(e)
         send MSG(l, lc, f, fc, blocked)
     if U_N = 0 AND B_B + (1 + ε)S_B ≤ cap(e)
         send MSG(l, lc, f, fc, nf)
         U_B ← S_B
         U_N ← S_N
         S_B ← 0
         S_N ← 0
         CL ← CL + 1
```

Figure 4: A constant-space algorithm for link e of a network node.

```
For MSG(l, lc, f, fc, nf)
  if nf = pending
        level ← level + 1
        send MSG(level, 1, pending, 0, pending)
  if nf = stalled
        send MSG(level, 0, pending, 0, pending)
  if nf = blocked
        if f = blocked
              send MSG(level, 0, blocked, 0, blocked)
        else
              send MSG(level, 0, blocked, 1, blocked)
```

Figure 5: A constant-space algorithm for a session source.

a new session starts, the source sends a message of the form $MSG(level,\ prev\_level,\ flag,\ prev\_flag,\ newflag)$ where $level$ and $prev\_level$ are the current and previous level of the session respectively; $flag$ and $prev\_flag$ are the current and previous session state either $blocked$ or $pending$, respectively; and $newflag$ is the only field that an intermediate node may alter. $newflag$ can have four values $decrease < blocked < stalled < pending$ it is initialized by the source to the highest value, $pending$, and can be decreased by intermediate nodes. As before, a session source can send a control message only after the receipt of the previous one. For a flag f we define

$$\tilde{f} = \begin{cases} blocked & \text{if } f = blocked \\ pending & \text{if } f \neq blocked \end{cases}$$

The algorithm maintains for each link b_level, a variable that holds the level at which the link becomes a bottleneck. The variable is required for the time when sessions reach their bottleneck level on a link. Since the active sessions find out they cannot increase their rate, they change their state to $blocked$ (one by one) until eventually the number of active sessions is small enough and $\sum_{i=0}^{\log B} C[i][blocked](1+\varepsilon)^i + C[CL][pending](1+\varepsilon)^{CL+1} \leq cap(e)$ evaluates to true. As a result, some of the sessions increase their rate one level above the maxmin level. The level increase signals the other sessions to change their state back to $pending$, which results in decrease in the allocation level of the first sessions to the max-min level. Thus, sessions alternate forever between the max-min level and one level higher. Using b_level this situation is identified, and all the sessions remains at the max-min fair level.

Instead of using a single rotating message per session one can use periodic submission of messages and keep the state in the intermediate edges. This change costs linear space in each of the edges but nl.1 For $MSG(l,\ pl,\ f,\ pf,\ nf)$

nl.2 $C[pl][pf] \leftarrow C[pl][pf] - 1$

nl.3 $C[l][f] \leftarrow C[l][f] + 1$

nl.4 if $\sum_{i=0}^{\log B} C[i][pending] = 0$

nl.5 $m = i : \max_{0 \leq i \leq \log B} C[i][blocked] \neq 0$

nl.6 if $\sum_{i=0}^{m-1} C[i][blocked](1+\varepsilon)^i + C[m][blocked](1+\varepsilon)^{m+1} \leq cap(e)$

nl.7 $CL \leftarrow m + 1$

nl.8 $b\_level \leftarrow 1 + \log B$

nl.9 else

nl.10 $CL \leftarrow m$

nl.11 else if $\sum_{i=0}^{\log B} C[i][pending] = C[CL][pending]$

nl.12 if $b\_level = CL$ AND $\sum_{i=0}^{CL-1} C[i][blocked](1+\varepsilon)^i + (C[CL][pending] + C[CL][blocked])(1+\varepsilon)^{CL+1} \leq cap(e)$

nl.13 $CL \leftarrow CL + 1$

nl.14 if $b\_level > CL$ AND $\sum_{i=0}^{\log B} C[i][blocked](1+\varepsilon)^i + C[CL][pending](1+\varepsilon)^{CL+1} \leq cap(e)$

nl.15 $CL \leftarrow CL + 1$

nl.16 if $\sum_{i=0}^{\log B} \left(C[i][pending](1+\varepsilon)^{CL} + C[i][blocked](1+\varepsilon)^i \right) > cap(e)$ AND $\sum_{i=CL+1}^{\log B} C[i][pending] + C[i][blocked] = 0$

nl.17 $CL \leftarrow CL - 1$

nl.18 if $l > CL$

nl.19 send $MSG(l,\ pl,\ f,\ pf,\ decrease)$

nl.20 if $l < CL$

nl.21 send $MSG(l,\ pl,\ f,\ pf,\ nf)$

nl.22 if $l = CL$

nl.23 if $\sum_{i=0}^{\log B} C[i][pending] = C[CL][pending]$

nl.24 $b\_level \leftarrow CL$

nl.25 send $MSG(l,\ pl,\ f,\ pf,\ \min\{nf,\ blocked\})$

nl.26 else

nl.27 send $MSG(l,\ pl,\ f,\ pf,\ \min\{nf,\ stalled\})$

Figure 6: A log-space algorithm for link e of a network node.

```
For MSG(l, pl, f, pf, nf)
  if nf = decrease
        level ← level − 1
        send MSG(level, l, pending, f̃, pending)
  else if nf = blocked
        send MSG(level, l, blocked, f̃, pending)
  else if nf = stalled
        send MSG(level, l, pending, f̃, pending)
  else
        level ← level + 1
        send MSG(level, l, pending, f̃, pending)
For START
  send MSG(−1, −1, pending, pending, pending)
```

Figure 7: A log-space algorithm for a session source.

yields an algorithm that is robust to control message loss. It is also closer in concept to the ATM use of resource management (RM) cells (see section 6), and to RSVP soft state refresh messages.

5 Convergence Time

We start by stating the correctness of the algorithm:

Theorem 5.1 *Sufficient time after all changes ceases*

1. *all sessions are blocked, and*
2. *for every link, CL is equal to the level of the session with the highest rate.*

Proof: Suppose the opposite. It easy to show from the code that either CL is changed or the rate of one of the sessions is changed, or a non-blocked session becomes blocked.

Suppose the theorem holds. Then the condition in line nl.4 holds and if the appropriate rates are assigned then the condition in line nl.6 does not hold and CL does not change. This is the only stable condition. □

To prove the convergence time, assume first that at time $t = 0$ no bandwidth is allocated in the network, and some sessions start sending control messages asking for bandwidth allocation. Let the round trip end-to-end delay (RTD) be one time unit for all sessions. Initially, $CL = 0$ for all network edges.

Let l_{min} be the appropriate allocation level for the most congested bottleneck link, and let N_e be the number of sessions sharing an edge e. We shall first prove

Lemma 5.1 *At time $t = 2l_{min}$, $level(s) = l_{min}$ for all sessions, and $C[l_{min} - 1][pending] = N_e$ and $CL = l_{min}$ for all network edges.*

Proof: By induction. By $t = 1$, every link processes exactly N_e messages of the form $MSG(-1, -1, pending, pending, pending)$, one from each of the session passing through it. Each of these messages is transferred unchanged as the edge level is higher than the session level (line nl.20). As a result all the sessions set their transmission level to 0. By $t = 2$, every link processes exactly N_e messages of the form $MSG(0, -1, pending, pending, pending)$, one from each of the sessions passing through it. Each of these messages increases $C[0][pending]$ by one (line nl.3). Due to the first received message in this time slot, the edge increases its level to 1 (line nl.14). This completes the induction base for $k = 1$.

Now suppose that at the end of time $t = 2k$, $level(s) = k$ for all sessions and $C[k-1][pending] = N_e$ and $CL = k$ for all network edges. Since $C[k-1][pending] = N_e$ and $level(s) = k$ the next message each session sends is $MSG(k, k-1, pending, pending, pending)$. After the receipt of N_e messages on an edge, $C[k][pending] = N_e$ and thus CL is increased to $k+1$. Note that all the messages are forwarded with $MSG(\cdot, stalled)$ as the session level and the edge level are equal (line nl.27). An exclusion is the last message to visit an edge, this message increases e's level to $k+1$ and thus is forwarded unchanged. However, for a session to be able to increase its level at this time slot, its control message should be the last in every switch, otherwise its $newflag$ parameter is decreased to $stalled$. In any case, in the next time slot all sessions can increase their rates to $k+1$ since $CL = k+1$.

Thus at time $t = 2k+2$, we have $level(s) = k+1$ for all sessions, and $C[k][pending] = N_e$ and $CL = k+1$ for all network edges. This concludes the induction. □

Lemma 5.2 *At time $t = 1 + 2\log_{1+\varepsilon} B$, all sessions have their appropriate rates.*

Proof: By lemma 5.1, the sessions sharing the most congested link have their appropriate rate. To prove this lemma, we show that when a link reaches its appropriate rate, say k at time t_{i_k} it cannot slow down the synchronization process in other less congested links.

We show that the induction step of lemma 5.1 holds here, too. Suppose that at the end of time $t = t_{i_k}$, $level(s) = k$ for all non-blocked sessions, and $C[k-1][pending] = N_e^a$ and $CL = k$ for all network edges. N_e^a is the number of active sessions in link e. Consider two edges e, which is a bottleneck link for certain sessions at level k, and e' that is less congested than e, that has some sessions that are routed through both of them.

By $t = t_{i_k} + 1$, all sessions passing either through links e or e' decrease $C[k-1][pending]$ by one and increase $C[k][pending]$ by one. Thus, all the received messages on links e and e' are forwarded as $MSG(\cdot,\ stalled)$ except for the last to decrease $C[k-1][pending]$ in a link that is forwarded with $MSG(\cdot,\ blocked)$. This last message in e' also increase CL to $k+1$. In the next time slot (no later than $t = t_{i_k} + 2$), as CL is not increased at link e, and $\sum_{i \neq CL} C[i][pending] = 0$ all sessions are forwarded by e with $MSG(\cdot,\ blocked)$. At the same time, sessions that traverse e' but not e are forwarded untouched since $CL = k+1$ and thus all sessions not restricted by e increase their rate.

By $t = t_{i_k} + 3$, one $MSG(k,\ k,\ blocked,\ pending,\ pending)$ for every session in e traverses e', decreasing $C[k][pending]$ by one and increase $C[k][blocked]$. All other unrestricted sessions decrease $C[k][pending]$ by one and increase $C[k+1][pending]$. Thus the last message that decreases $C[k][pending]$ to zero at e', whether it is a *blocked* or a *pending* message, finds the condition in line nl.14 true and increases CL by one. □

Note that if some of the sessions have a shorter round trip propagation delay, they simply wait and have 'idle' cycles in which a message is not triggering any state change. To capture this characteristic, assume that the network is at a stable state and some change occurs in a single link either a session addition or deletion. Assume that some sessions are slow, i.e., they have long round-trip delay (RTD), used as the time unit for our measurements, and some sessions are fast, i.e., have very short RTD with respect to the fast sessions. The difference enable the fast sessions to have an infinite number of control packets traveling trough a link between any two consecutive traversals of control packets of the slow connections. Under these assumptions, we prove the following lemmata.

Lemma 5.3 *The addition of a new session to a stable network requires logarithmic quiescence time.*

Proof: Let e be the most congested link used by the newly added session, S_0, let l_1 be the allocation level of the sessions whose bottleneck link is e, and let the appropriate rate in link e after the addition of session S_0 be l_2. Let t_0 be the time S_0 is added.

If the network is stable at t_0 then by theorem 5.1 all sessions are *blocked* and the value of CL in link e is l_1. $MSG^{S_0}(\cdot)$ finds the condition in line nl.20 true and thus S_0 increases its rate by one every RTD, namely at least every time unit. At the second visit (not later than $t_0 + 2$) of $MSG^{S_0}(\cdot)$ in e, the condition in line nl.16 becomes true since $C[0][pending] = 1$, and CL is decreased by one.

Thus, no later than t_0+3 all sessions with rate l_1 decrease their rate by one, and by time t_0+4 all these sessions become active in link e. At this point of time, S_0 evaluates the condition in line nl.20 to true, and increases its rate every RTD. At the same time, the first condition in line nl.16 evaluates to true, causing the rest of the sessions to decrease their rate synchronously. The fast sessions that wait for synchronization remain active since their $newflag$ is set to *stalled* (line nl.27). Thus, the decrease is done at the rate of the slowest session, which is one level per cycle.

Thus, the quiescence time is $4 + \max\{l_2,\ l_1 - l_2\} \in O(\log B)$. □

Lemma 5.4 *If at time t_0, all sessions have their appropriate rates, and a session whose bottleneck link is e is deleted, then after 2 time units either all sessions sharing link e as their bottleneck link increase their rate or non does.*

Proof:

By theorem 5.1, all sessions are blocked, and the rate of the sessions that share link e as their bottleneck link is equal to $CL = l_1$ at link e.

The sessions whose level is below l_1 are not effected from the deletion of a session in e as these sessions are limited by other more congested links where the deleted session is not routed through since e is its bottleneck link. Also not effected are sessions at level l_1 that has another bottleneck link where new bandwidth is not becoming available.

The first control message that arrives at e after bandwidth becomes available finds the condition in line nl.6 true and increase CL. Subsequently, this message and all other coming messages are forwarded without change, arriving to the source with the value of $newflag$ at its initial setting, *pending*. The sources increase their level and return to state *active*. In the worst case, the fast sessions increase their level while synchronizing among themselves until they use the entire free bandwidth before the slow sessions have a chance to indicate that they are active, too. However, note that by time $t_0 + 1$ all the slow sessions increase their level since the condition in line nl.20 is evaluated true, and by time $t_0 + 2$ all the slow sessions become active in e.

Note that as a result of the change in state of a slow session from *blocked* to active condition nl.16 is evaluated to true if the fast session increased their rates too high. However, in this case CL is decreased, and all the fast sessions are thus required to lower their level. When all the slow sessions become active, the fast sessions reach their final level, as they react fast to the restriction condition in line nl.16. □

Next we prove that in a stable network a change in the session lay-out requires a logarithmic quiescence time. For this end we use the following definitions. Given a graph $G = (V, E)$, and a session lay-out S, the *cross session graph* $G^x = (V^x, E^x)$ is defined by $V^x = E$, $E^x = \{(v_1, v_2) | v_1, v_2 \in V^x,\ \exists s \in S : s \in v_1 \cap s \in v_2\}$. In other words, the cross section graph has a node for every edge in the original graph and an edge in the cross session graph exists between every two nodes whose corresponding edges in the original graph are used by at least one common session. Let the weigh of a node v in G^x, $w(v)$, be the level at which its corresponding link in G become a bottleneck link ($w(v) = -1$ for links that never become bottleneck links), The influence graph $G^I = (V^I, E^I)$ is define as a directed graph where $V^I = V^x$, $E^I = \{(v_1, v_2) | (v_1, v_2) \in E^x, v_1 < v_2\}$, and $\forall e \in E^I$ $w(e) = w(v_2) - w(v_1) > 0$. By its definition, G^I is a DAG and the maximal weight of any path in G^I is $\log_{1+\varepsilon} B$.

Theorem 5.2 *The quiescence time of the algorithm is in $O(\log_{1+\varepsilon} B)$.*

Proof: To prove the theorem we need to prove the claim that every chain of changes in session weights is mapped to a path in G^I.

Suppose, a change occurs in the network, i.e., the session lay-out on the graph G is changed. If the change in the link (a node of G^x and G^I) is a rate reduction it cannot cause the increase in bandwidth of a link in a lower level because that imply that the initial rate allocation was not fair. If the change in the link is a rate increase it cannot cause the decrease of a rate below the new level or else the fairness criteria is violated.

Now, by lemmata 5.3 and 5.4 the time for increasing or decreasing the level of all sessions in a link by one unit is at most 2. The maximal time for a level change to propagate through a path of length x is thus less than $2x$. $\square$

We show in the appendix that when the number of sessions is sufficiently large (more than ε^{-1} in each link) not more than one change in allocation per session is needed. Applying this result to theorem 5.2 implies that the quiescence time is constant in a highly loaded network.

6 Applicability for ATM signaling

In this section, we examine the applicability of our algorithms to the current ATM traffic management specification [Com96]. We assume that switches do not support explicit rate signaling, as for this case the implementation is trivial. Thus, we concentrate on relative rate marking and show how our algorithm can be implemented in cooperation of non participating switches, i.e., with switches that do not implement our algorithm but only relative rate marking.

For the simplicity of the discussion, we shall refer to a linear space implementation of the log-space algorithm (figures 6 and 7). For this implementation, we assume that link processors maintain, $Tab[s]$, a table of linear size where each connection, s, has an entry that holds its current rate and its status flag (*blocked* or *pending*). Each link processor should maintain the number of connections in the two states in each level (the array CL), and should register the current allocation level, CL. As a result, fields pl and pf are not required in the RM cell.

The code of figure 6 should be changed as follows. The first three lines should be replaced by

For $MSG^s(rate,\ f,\ nf)$
  $l \leftarrow \log_{1+\varepsilon} rate$
  if $Tab[s] \neq (l,\ f)$
    $C[Tab[s]] \leftarrow C[Tab[s]] - 1$
    $C[l][f] \leftarrow C[l][f] + 1$
    $Tab[s] \leftarrow (l,\ f)$

The rest of the lines should be renumbered. In addition, $MSG^s(l,\ pl,\ f,\ pf,\ \cdot)$ should be replaced with $MSG(l,\ f,\ \cdot)$ throughout the code. Similar change to MSG should be applied to the code in figure 7.

Let us first assume that the MCR parameter is zero for all connections. We show that by using several fields in the RM cell [Com96, section 5.10] in a way coherent with the standard, our algorithm can be implemented in a heterogeneous environment where not all switches implement the suggested algorithm.

The CCR (Current Cell rate) field is set by the source to indicate the current ACR (Allowed Cell rate). In the code in figure 6 we assume that the level is sent in the message. However, calculating the level, l, from the rate requires only a logarithm that can either be calculated or extracted from a table.

The flag nf, can be implemented by using the two RM-cell bit-fields: CI and NI. CI is used to indicate congestion, NI is used to prevent rate increase. These two bit-fields map to our flag nf value as follows:

pending — A connection is free to increase its rate. CI=0. NI=0.

stalled — A connection is waiting for synchronization. CI=?. NI=?.

blocked — A connection reached its pair share. CI=0. NI=1.

decreased — A connection is required to decrease its rate. CI=1. NI=1.

The mapping of *pending*, *blocked*, and *decreased* is natural. The mapping of the *stalled* value requires a careful examination. The only combination not in use is CI=1 and NI=0. However. at this state, the switch is not congested, thus, we do not wish to set the CI flag. If CI=0, any selection for the NI mapping will result in two values mapped to the same flag combination. To differentiate between the two values, an additional bit from the reserved bits in the RM cell should be used.

Any mapping should consider the option that some of the switches along the route may not be programmed to use our algorithm and are only capable of setting the NI and CI bits (according to *relative rate marking* [Com96, sec. 5.10.6]). If such a switch on the way is setting the NI bit, the setting must be understood as "the session is *blocked*", or otherwise the algorithm may livelock by waiting to a synchronization that cannot occur. This can be done by mapping the *stalled* value to CI=0 and NI=0 with the additional reserved bit set. This way, setting NI either by a participating or by a non-participating switch results in the blocking of this session without livelocking others. A source that does not implement our algorithm, may increase its rate by one level without synchronizing with the rest of the sessions but in the next cycle it will be forces to decrease its rate back since its rate will be greater than CL and thus participating switches will set both the CI and NI bits. An additional reserved bit is required to implement the f field.

Switches that implement EFCI marking or explicit rate marking can coexist with switches that use our relative rate marking variant. It is important that the rate decrease factor (RDF) and the rate increase factor (RIF) are are set to corresponding values and that these two parameters are equal for all sessions. If MCR is greater than zero, our scheme can be easily used to achieve the "MCR plus equal share" as suggested in [Com96, sec. I.3].

7 Simulation Results

We simulated the logarithmic algorithm both for the version that appears in figures 6 and 7, and for the version that appears on page 16. We first report the earlier version that require a single circulating control message per connection as described in figures 6 and 7.

Figures 8 and 9 show the session allocation level and link utilization for a single link network with round trip delay (RTD) of 2μsec. The minimum allocation level is 64Kbps and the link capacity is 25Mbps, and $\varepsilon = 0.05$.

At time t=0, five sessions become active and raise their allocation level in a synchronous fashion,

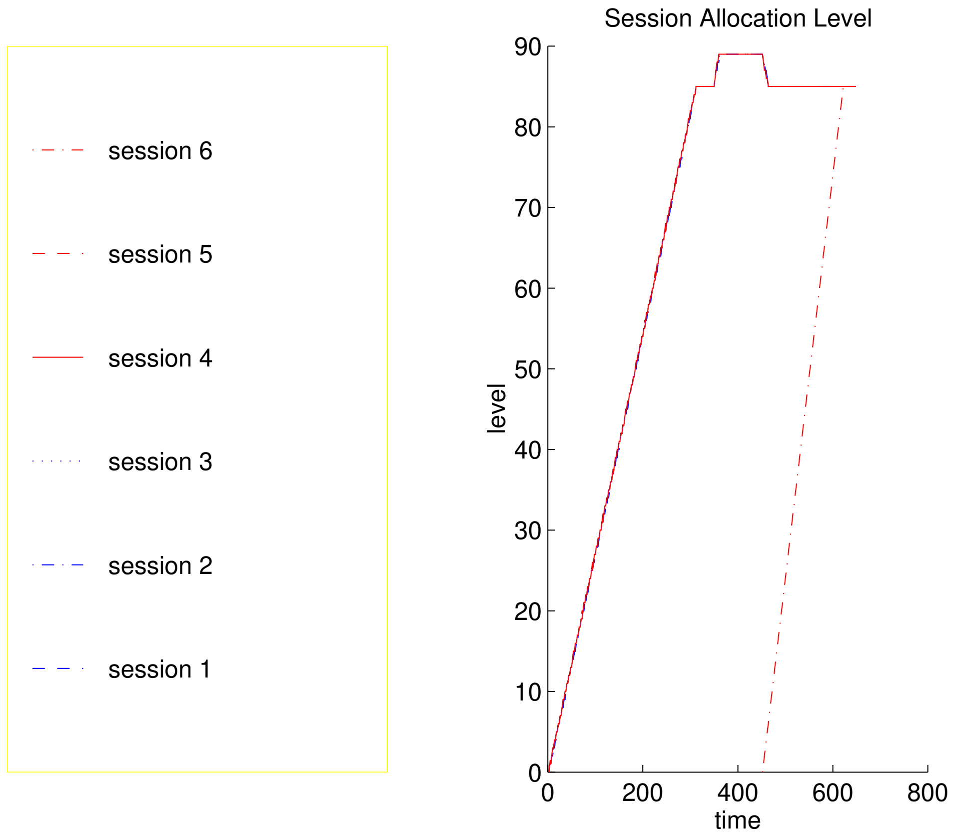

Figure 8: A one link network - session allocation level.

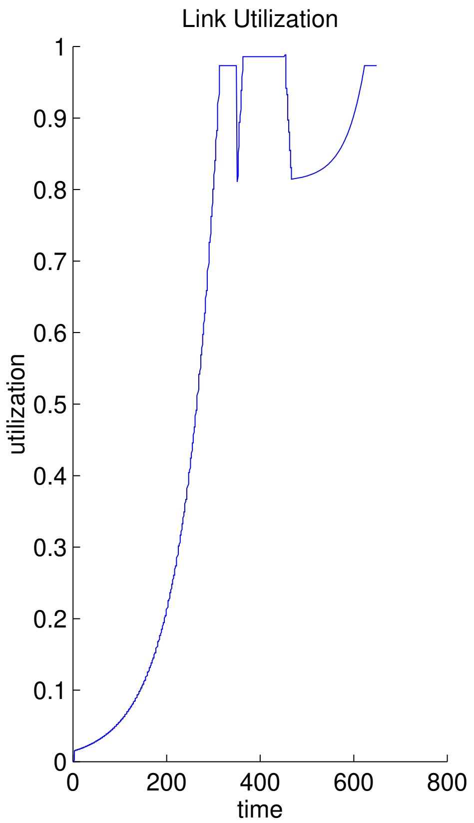

Figure 9: A one link network - link utilization.

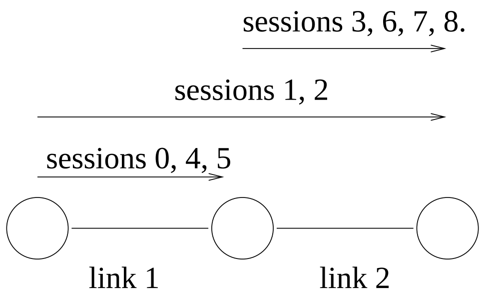

Figure 10: A two link network - session layout.

as can be seen in figure 8. Note that the Y-axis unit is in allocation levels, i.e., the graph shows the logarithm of the bandwidth used by the session. The five sessions reach their fair share in a time equal to about two RTD times their fare level as stated by lemma 5.2. At time t=350, session 5 terminated, and as a result the other four session increase their allocation level in full synchrony. At time t=450, a new session become active. The four active session reduce their rate to the proper level in synchrony, while the new session increases its rate until it reaches its proper level.

Figure 9 shows that link utilization is always above 97% in steady state. When session 5 terminates the freed bandwidth is shared by the rest of the session very quickly since these session are already at high allocation level and every increase by a factor of ε is substantial. This is not the case when a new session joins, as it has to increase its rate from level 0. A simple addition of a scout control message that traverses the route to check for the bottleneck link allocation level and starta at a level slightly lower that that can solve this problem.

Figures 11 and 12 show the session allocation level and link utilization for a two link network, where link 1 has a delay of 1μsec, and link 2 a delay of 9μsec. The session layout is depicted in figure 10. All the session become active at time 0, except session 1 that becomes active at time 750.

The following can be observed from figure 11.

- In the beginning, before the session 2 is registered in link 1 (due to its long RTD), sessions 0, 4, and 5 raise their allocation level at a fast rate and are stopped after one RTD of session 2 (20μsec) waiting to synchronize with it.
- When session 1 becomes active, all the other session stop raising their level and wait to synchronize with it. Session 1 raises its rate one level every RTD until it reaches the level of the other sessions. At this point all sessions continue to raise their level every 2RTD (40μsec).
- After the sessions with high RTD value reach their fair level, the remaining sessions (0, 4, and

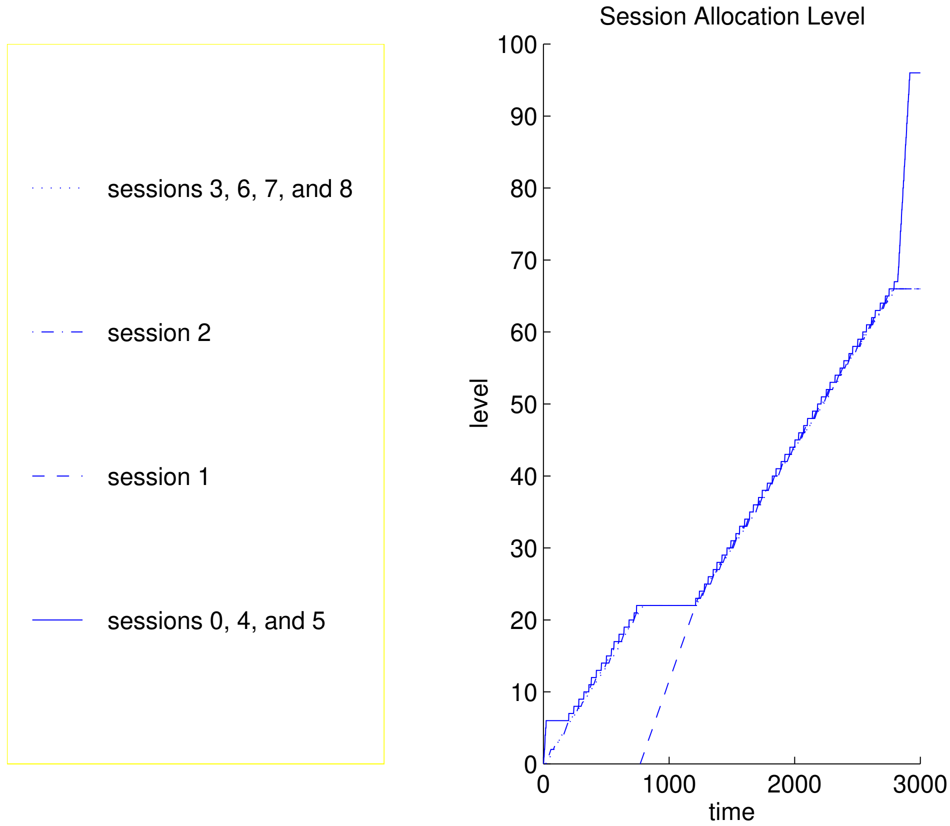

Figure 11: A two link network - session allocation level.

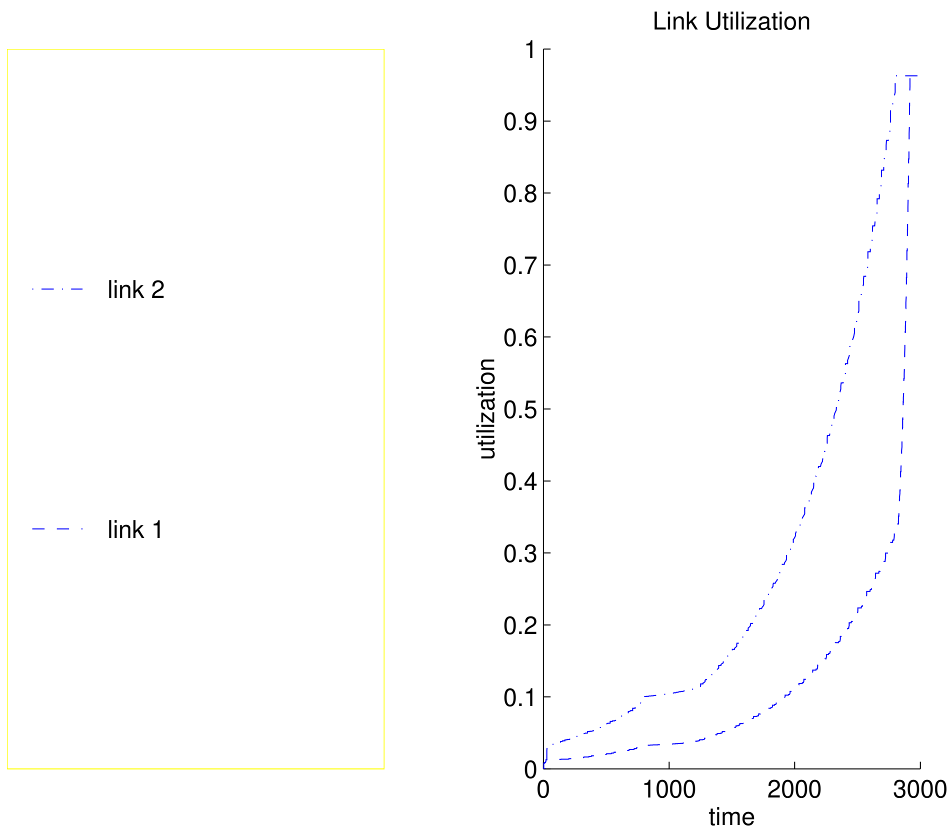

Figure 12: A two link network - link utilization.

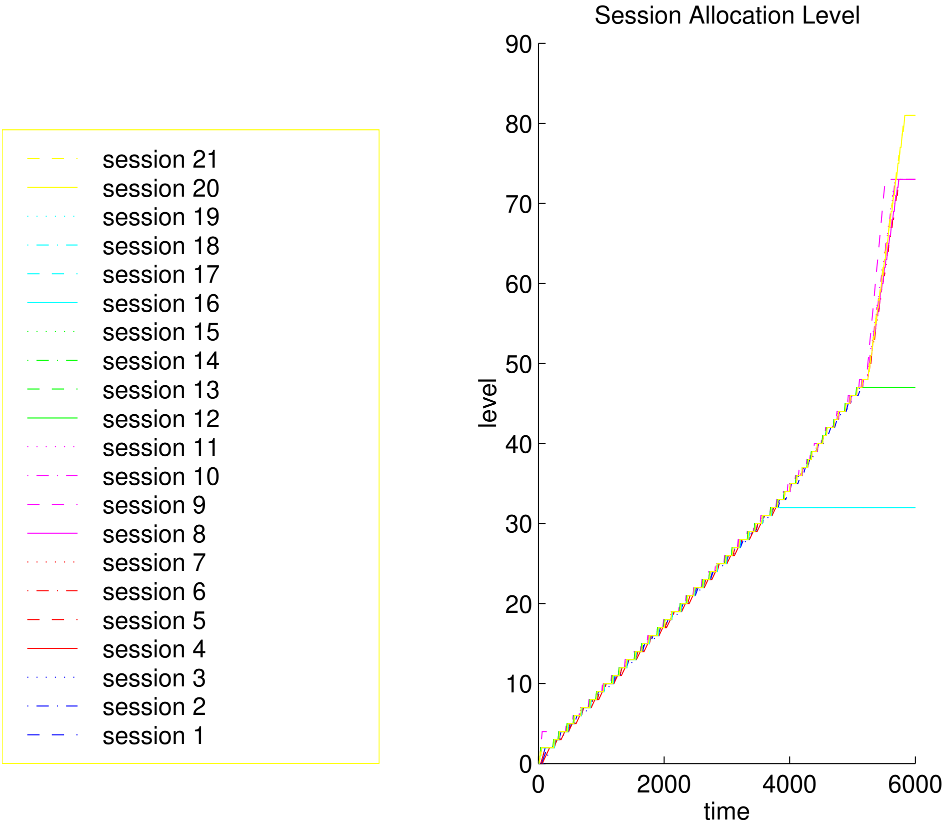

Figure 13: A multi link network - session allocation level.

5) continue to increase their rate every 2RTD of the active sessions, i.e., every 4μsec.

Figures 13 and 14 show the simulation results for the network simulated in [CRL96]. The link delay is set to 6μsec to match the switching and the propagation delay in [CRL96].

Note that the once the allocation level of the sessions with long RTD is set, the rest of the sessions can increase their allocation level at a higher rate since the synchronization is required only among the *pending* sessions. In the first As expected utilization increases exponentially.

8 Concluding Remarks

The logarithmic-space algorithm can be generalized to allow session to be assigned any finite set of rates as required by the nature of the applications. To this end each node and end-user (session) should hold a finite table of sorted rates to be used instead of the natural powers of $1 + \varepsilon$. A more practical generalization is to change the value of ε for different ranges of rates. For example,

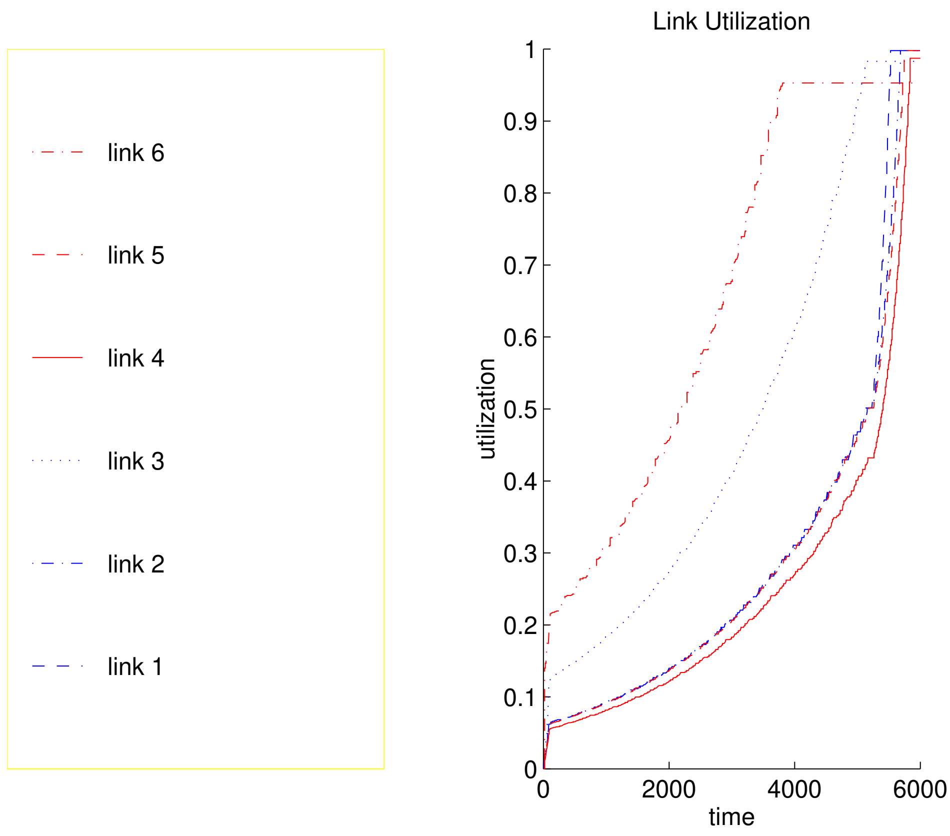

Figure 14: A multi link network - link utilization.

for lower and higher allocation levels to use a rather high ε and for the middle levels that are the most frequently used to set a low ε. This allocation scheme does not consume memory, minimizes the wasted bandwidth due to quantization, and speeds-up the rate at which sessions reach their appropriate allocation level at start-up.

For the case of non greedy sessions, wasted bandwidth can be reduced by measuring the bandwidth that is used at every link, instead of summing the allocated bandwidth (in lines nl.6, nl.12, nl.14, and nl.16 of the logarithmic space algorithm). This way permanent non-used allocated bandwidth can be reused.

When a new connection is established, convergence can be made faster if instead of starting at the lowest level, the connection will first probe its route for the minimal allocated value, and start from this allocation level. As demonstrated in the appendix, the convergence time should not exceed 2 RTD in most cases.

Acknowledgement

We would like to thank Lei Zhang for her help in obtaining the simulation results.

References

[AMO96a] Yehuda Afek, Yishay Mansour, and Zvi Ostfeld. Convergence complexity of otimistic rate based flow control algorithms. In *28th Annual ACM Symposium on Theory of Computing*, pages 89 – 98, May 1996.

[AMO96b] Yehuda Afek, Yishay Mansour, and Zvi Ostfeld. Phantom: A simple and effective flow control scheme. In *ACM SIGCOMM*, pages 169 – 182, August 1996.

[BF95] Flavio Bonomi and Kerry W. Fendick. The rate-based flow control framework for the available bit rate ATM service. *IEEE Network*, 9(2):25 – 39, March/April 1995.

[BG92] Dimitri Bertsekas and Robert Gallager. *Data Networks*. Prentice Hall, second edition, 1992.

[BP95] Lawrence S. Brakmo and Larry Peterson. TCP Vegas: end to end congestion aviodance on a global Internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465 – 1480, October 1995.

[Com96] The ATM Forum Technical Committee. Traffic management specification version 4.0, April 1996. af-tm-0056.000.

[CRL96] Anna Charny, K. K. Ramakrishnan, and Anthony G. Lauck. Scalability issues for distributed explicit rate allocation in ATM networks. In *INFOCOM96*, pages 1198 – 1205, March 1996.

[FMS97] Panagiota Fatourou, Marios Mavronicolas, and Paul Spirakis. Efficiency of oblivious versus non-oblivious schedulers for optimistic rate-based flow control. In *PODC*, August 1997.

[GK80] Mario Gerla and Leonard Kleinrock. Flow control: A comparative survey. *IEEE Transactions on Communications*, COM-28(4):553 – 574, April 1980.

[Jac88] Van Jacobson. Congestion aviodance and control. In *ACM SIGCOMM*, pages 314 – 329, August 1988.

[Jaf81] Jeffrey M. Jaffe. Bottleneck flow control. *IEEE Transactions on Communications*, COM-29(7):954 – 962, July 1981.

[Kes91] Srinivasan Keshav. A control-theoretic approach to flow control. In *ACM SIGCOMM*, pages 3 – 16, September 1991.

A Conditions for Allocation Level Change

In this appendix we show that the change in allocation level due to a session addition is minimal when the number of sessions is sufficiently high.

Let C be the residual capacity on a link, let m be the number of sessions sharing this capacity and let l_1 be their allocation level. By definition:

$$m(1+\varepsilon)^{l_1} \leq C \tag{1}$$

Let l_2 be the appropriate allocation level after the addition of one session. By definition

$$(m+1)(1+\varepsilon)^{l_2} \leq C \tag{2}$$

$$(m+1)(1+\varepsilon)^{l_2+1} > C \tag{3}$$

A simple algebraic manipulation of relations 1 and 3 yields

$$
\begin{aligned}
l_1 &\leq \log_{1+\varepsilon} \frac{C}{m} && (4)\\
l_2 &> \log_{1+\varepsilon} \frac{C}{m+1} - 1 && (5)\\
& && (6)
\end{aligned}
$$

Thus the change in levels is bounded by

$$l_1 - l_2 < \log_{1+\varepsilon} \frac{m+1}{m} + 1 \tag{7}$$

For $m > \frac{1}{\varepsilon}$, we have $\log_{1+\varepsilon} \frac{m+1}{m} < 1$, meaning that the addition of one session for high enough m may require, at most, a decrease of one level.

Similarly, two is the level change limit when $m > \frac{1-\varepsilon^2}{2\varepsilon}$, and one is level change limit when k new sessions are added and $m > \frac{k}{\varepsilon}$.