

AN IP IMPLEMENTATION OF OPTIMIZATION FLOW CONTROL

David Lapsley and Steven Low *

Department of EEE, University of Melbourne, Australia

Email: {d.lapsley, s.low}@ee.mu.oz.au

Abstract

Flow control allows sources to adjust their bandwidth usage to the level of availability in a network, and hence reduces packet loss, increases network utilization, and prevents/reacts to network congestion. The Internet uses TCP flow control; Asynchronous Transfer Mode networks will use rate based flow control. In this paper we present an optimization approach to rate based flow control. We describe an implementation of this flow control on an IP network, and we present some performance results of this implementation.

1 Introduction

Flow control is an essential part of any data transport protocol. Whether it be the Available Bit Rate (ABR) service in an Asynchronous Transfer Mode (ATM) network, or a best effort TCP connection across an IP network, flow control reduces packet loss, increases network utilisation, and prevents network congestion. Flow control schemes can be divided into two types, according as they use implicit or explicit feedback. Traditional TCP uses implicit feedback, such as the round trip time or loss of acknowledgement packets, to adjust the source bandwidth usage. Round trip times control, through windowing, the rate at which packets are injected into the network, and packet losses, or the lack thereof, adjust dynamically the window size. Schemes that use implicit feedback are well suited to heterogeneous networks such as the Internet where intermediate routers may not provide any congestion information to the source. They are however less effective. The Internet Research Task Force's End to End Group is currently exploring methods that use explicit feedback to enhance the flow control mechanism in TCP. One such method proposed by Floyd in [1] was the use of Explicit Congestion Notification (ECN) in which a single bit in the header of an IP packet could be used to provide congestion information from the network to the source. This proposal is similar to the DECBit scheme of Ramakrishnan and Jain [7].

In this paper we take Floyd's binary feedback proposal a

*The first author would like to thank the Commonwealth government and the Australian Telecommunications and Electronics Board for their financial support.

step further and propose the use of multiple bits for feedback flow control of TCP. The justification of our scheme is its ability to provide differentiated services to users according to their relative valuation of bandwidth, as described by their utility functions. The overall goal of our scheme is to maximize total user utility, and an iterative method to achieving it takes the form of a distributed asynchronous flow control scheme where users adjust their rates based on network feedback and their utility. In equilibrium users receive different bandwidth allocations that reflect their valuation of bandwidth and how their use implies a cost to others.

In [5] we formulate our optimisation approach to flow control and show how to implement such a scheme in a network that conforms to the ABR standard [8]. Here we apply the same scheme for IP networks and propose the use of Resource Management (RM) packets for feedback, the same way as RM cells are used in Explicit Rate control of ABR service. In [6] we present a detailed analysis of the convergence and fairness properties of our algorithm, and in [4] we present an extension to our algorithm that uses single-bit feedback, and report extensive performance measurements of our implementations.

Our paper is structured as follows. In section §2 we briefly review the problem formulation and the distributed asynchronous solution. In §3 we describe an implementation of our algorithm on a testbed IP network. We present the results of our investigations in §4 and conclude in §5.

2 The Algorithm

Our approach models the network as a set $L = \{1, \dots, L\}$ of unidirectional links of capacity c_l , $l \in L$ ¹. The network is shared by a set $S = \{1, \dots, S\}$ of sources. Source s is characterized by four parameters $(L(s), U_s, m_s, M_s)$. The path $L(s) \subseteq L$ is a subset of links that source s uses, $U_s : \mathcal{R}_+ \rightarrow \mathcal{R}$ is a utility function, $m_s \geq 0$ and $M_s \leq \infty$ are respectively the minimum and peak cell rate of source s . Source s attains a utility $U_s(x_s)$ when it transmits at rate x_s that satisfies $m_s \leq x_s \leq M_s$. Let $I_s = [m_s, M_s]$ denote the range in which source rate x_s must lie and $I =$

¹The capacity c_l in the model should be set to ρ_l times the real link capacity where $\rho_l \in (0, 1)$ is a target utilization.

$(I_s, s \in S)$ be the vector. We assume U_s is increasing and strictly concave in its argument on I_s . For each link l let $S(l) = \{s \in S \mid l \in L(s)\}$ be the set of sources that use link l .

Our objective is to choose source rates $x = (x_s, s \in S)$ so as to:

$$\mathbf{P:} \quad \max_{x_s \in I_s} \sum_s U_s(x_s) \quad (1)$$

$$\text{subject to} \quad \sum_{s \in S(l)} x_s \leq c_l, \quad l = 1, \dots, L. \quad (2)$$

The constraint (2) says that the total source rate at any link l is less than the capacity. A unique maximizer exists since the objective function is strictly concave, and hence continuous, and the feasible solution set is compact. Solving the primal problem P directly is however infeasible in practical networks due to complex coupling among sources through shared links. In [5] we developed a decentralized solution via the dual problem. It can be implemented as a distributed computation in which each link advertises a bandwidth price and a source adjusts its rate according to the sum of the bandwidth prices of all the links on its path. The links then adjust their prices in response to the new source rates, and the cycle repeats. We now present the asynchronous distributed algorithm developed in [5]. We call the algorithm Optimization Flow Control(OFC).

Let $T_s \subseteq \{1, 2, \dots\}$ be a set of times at which source s updates its rates based on its current knowledge of bandwidth prices along its path. At time $t \in T_s$ the bandwidth prices $(p_l(\tau_l^s(t)), l \in L(s))$ available at source s are the prices computed by the links $l \in L(s)$ at earlier times $\tau_l^s(t)$, where $0 \leq \tau_l^s(t) \leq t$ for all $t \in T_s$. The difference $t - \tau_l^s(t)$ represents the communication delay from link l to source s . Note that this delay depends on (l, s, t) and can be different for different link-source pairs and at different times. At an update time $t \in T_s$, source s computes its new transmission rate based on the bandwidth cost $\sum_{l \in L(s)} p_l(\tau_l^s(t))$. At times $t \notin T_s$ between updates source rates are unchanged.

Similarly let $\Theta_l \subseteq \{1, 2, \dots\}$ be a set of times at which link l adjusts its price. At an update time $t \in \Theta_l$ link l has available source rates $(x_s(\theta_s^l(t)), s \in S(l))$ computed by $s \in S(l)$ at earlier times $\theta_s^l(t)$, where $0 \leq \theta_s^l(t) \leq t$ for all $t \in \Theta_l$. Again the difference $t - \theta_s^l(t)$ represents the communication delay from source s to link l , and can be different for different s and l and at different times t .

Source s 's algorithm:

1. At each update time $t \in T_s$ source s chooses a new rate based on its current knowledge of prices:

$$x_s(t+1) = \arg \max_{m_s \leq x_s \leq M_s} U_s(x_s) -$$

$$x_s \sum_{l \in L(s)} p_l(\tau_l^s(t))$$

It then transmits at this rate until the next update, i.e., $x_s(t+1) = x_s(t)$ for $t \notin T_s$.

2. When a RM² packet returns source s replaces the price in its local memory with the value in the Bandwidth Price(BP) field.
3. From time to time source i transmits a RM packet with the Current Rate(CR) field set to the current source rate $x_s(t)$ and BP field to zero.

Link l 's algorithm:

1. At each update time $t \in \Theta_l$ link l computes a new price

$$p_l(t+1) = [p_l(t) + \gamma(\sum_{s \in S(l)} x_s(\theta_s^l(t)) - c_l)]^+.$$

At times $t \notin \Theta_l$, $p_l(t+1) = p_l(t)$.

2. When a RM packet from source s comes by (in the forward path), link l :

- increments the BP field by the current price $p_l(t)$.
- replaces the current copy of rate x_s by the value of the CR field.

3 Implementing OFC On an IP network

3.1 Overview

The scheme has been implemented on an experimental network. The testbed consists of 2 IBM compatible PCs (Pentium 233 Mhz) running the FreeBSD 2.2.5 operating system. Each PC was equipped with 64 Mbytes of RAM and 100 Mbps PCI ethernet cards. The PCs were connected via ethernet. Implementing the protocol involved writing two applications: *ofc* client application, and *ofcd* routing demon.

Two instances of the *ofc* client application are required for each connection: a source instance operating in ACTIVE mode and a destination instance operating in PASSIVE mode. For more information refer to [3]. parameter. Other parameters control the rate of packet generation, source and destination address, connection type (TCP, UDP, OFC)³ and connection duration.

Whenever the OFC transport protocol is used the *ofcd* demon must be run on all computers that have OFC

²Note that the feedback mechanism we use is based on the ATM ABR service model described in the introduction.

³The *ofc* application allows us to choose which transport level protocol we require. This makes comparison of performance between the protocols easier.

clients (sources/destinations) and on intermediate computers. The *ofc* client processes communicate with each other via the *ofcd* routing demons. All OFC clients transmit their packets to the routing demon on their host, which then either forwards the packet to another machine, or delivers it to a client process on the host. The OFC demons are also responsible for calculating the price of bandwidth on their outgoing links, and placing this price in the BP field of Forward RM (FRM) packets as they pass through.

Transport Layer Interface Each computer has a standard internet protocol stack consisting of TCP/UDP running on IP which sits above the network device drivers. The OFC protocol runs on top of the UDP layer, with OFC packets transported across the network on UDP connections. OFC packets are 500 bytes long and consist of a 10 byte header, 1 byte end of packet (EOP) marker, and a 489 byte data payload. The header contains fields that indicate the start of packet (SOP), OFC destination address (DST), packet sequence number (SN), payload type indicator (PTI), Header Error Check (HEC), Bandwidth Price (BP), and Current Rate (CR). Currently, all fields are 1 byte long (with the exception of the BP and CR fields which are 2 bytes).

Design Philosophy In designing the client and routing software, we had three key design goals: *portability*, *performance* and *ease of analysis*. For best performance, an in-kernel implementation of the protocol would have been desirable, however this would require recompiling the kernel of any machine on which we wanted to implement OFC. A user-space implementation, on the other hand, is much more portable: all that is required is to recompile the application software and then execute it on the target machine. We opted for portability and hence chose to implement our software in user-space. Keshav provides an excellent discussion on the different approaches to implementing protocol stacks, and their advantages/disadvantages in [2]. We have tried a number of different architectures and designs. We found that the design with the best performance was also the easiest one to analyze: a single context, monolithic implementation (see [2]) where all of the packet processing was performed within a single thread.

3.2 Implementing Client Software

Figure 1(a) shows the process and data structure for the client software. The software consists of two main threads of execution, the *proc* thread, and the periodic *monitor* thread. The *proc* thread is responsible for all of the packet processing, while the *monitor* thread periodically updates the statistics logs with receive and transmit rates. All of the information required by the client process is stored in the *connection* data structure.

The client software communicates with its local router demon using UNIX domain sockets. The client appli-

cation creates two such sockets: one for transmitting data (*tx_socket*) and one for receiving data (*rx_socket*). Each instance of the client application has its own socket for receiving data. To transmit data to the router, all client applications on a host share a socket with a predefined address.

The processing performed in the *proc* thread can be divided into a number of tasks which are scheduled according to a simple scheduling algorithm. There are four main tasks: *recv*, *recv_proc*, *send_proc*, and *send*.

Conceptually, *recv* and *send* can be thought of as residing in one protocol sublayer, with *recv_proc* and *send_proc* sitting above them in a higher protocol sublayer. *recv* and *send* interface to the socket layer below, while *recv_proc* and *send_proc* interface to the application layer above. The responsibilities of each task are described briefly below. The *recv* task reads data from the socket interface and transfers this data to the client's internal buffers. The *recv_proc* task does the receive side processing of data that has been read in by the *recv* task. *recv_proc* reassembles packets and checks that the packet headers have not been corrupted (using the HEC field and a simple parity checking algorithm). *recv_proc* processes the feedback in Backward Resource Management (BRM) packets, and adjusts the client's Cell Rate (CR) accordingly. FRM packets are processed and placed in a buffer where they await retransmission back to their source. The *recv_proc* task is also responsible for updating the *recv* statistics.

The *send_proc* task does the transmit side processing of data that is to be transmitted. It determines the type of the next packet to be transmitted (either FRM, BRM or data packet). If the packet to be transmitted is an FRM packet, it places the current transmission rate into the Current Rate (CR) field of the packet. The *send_proc* task performs rate control and ensures that the source only transmits packets at its CR. Rate control is done using the Virtual Scheduling Algorithm described in [8]. The *send_proc* task is also responsible for calculating the parity byte that is placed in the HEC field and is used for detecting data corruption. Once this has been done, the packet is passed on to the *send* task. The *send* task takes data from the *send_proc* task and transmits it through the socket interface to the local router. It then updates the transmit statistics.

The order of execution of the tasks depends on whether the client is active or passive (i.e. a source or a destination), and is constant in each case. One of the key factors in improving the performance of the software was the use of multiplexing IO ([9]) which was done via the *select* system call. Figure 2 shows the order of task execution for the source software. The destination software is the reverse with lines 4 and 8 swapped, and lines 5 and 9 swapped. This reflects our "optimizing for the common case" ([2]).

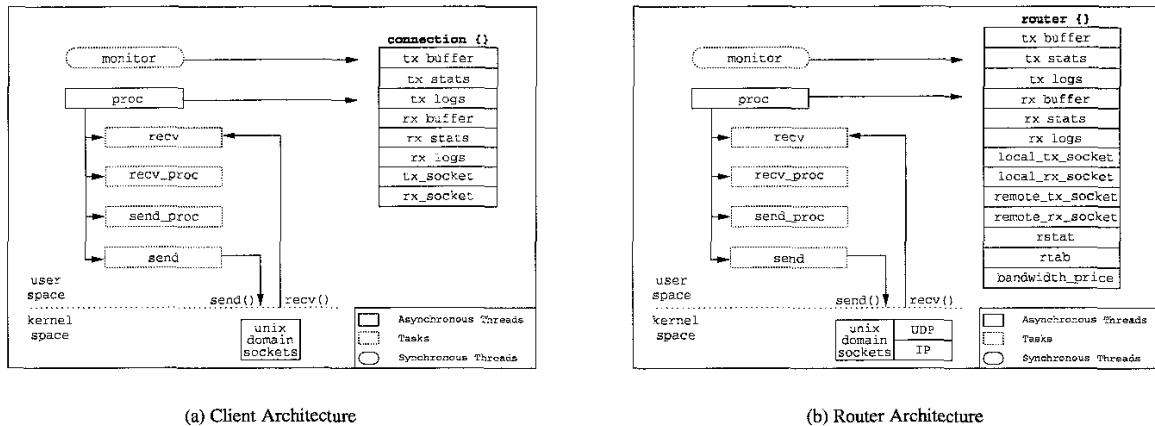


Figure 1: Architecture

```

1. repeat Nrm times
2. {
3.   use select to wait until transmit socket is ready for writing
4.   send_proc()
5.   send()
6. }
7. use select to poll receive socket to see if it is ready for reading
8. recv()
9. recv_proc()

```

Figure 2: Source Scheduling Loop

3.3 Implementing Routing Software

Figure 1(b) shows the process and data structure for the routing software. The structure for the routing software is very similar to that for the client software. The *router* structure maintains two sets of sockets: The “local” sockets are UNIX domain sockets and are used for communicating between the router and the clients on its machine, while the remote sockets are Internet domain sockets used for communicating with routing demons on remote machines. The router checks the local and remote sockets alternately to fairly divide its processing capacity. The router also maintains two tables: *rtab*, and *rstat*. The *rtab* table is a routing table which stores the address of the next hop along each route that passes through the router (indexed on the DST field of packets that pass through). The *rstat* table stores the CR field of FRM packets as they pass through the router (again, indexed on the DST field of packets that pass through). The values in the *rstat* table are the router’s best estimate of each active connection’s current source transmission rate, and is used in the calculation of link bandwidth prices.

General operation of the routing demon is as follows: periodically, the *monitor* thread calculates a bandwidth price for outgoing FRM packets, using the algorithm described in §2, and stores it in the variable *bandwidth_price*. Incoming data is received by the *recv* task via the *recv()* socket

function and placed into a storage buffer, which is then passed on to the *recv_proc* task. The *recv_proc* task then checks that the data is a valid packet, and updates the *rstat* table if the packet is an FRM packet. The packet is then passed on to the *send_proc* task. If the packet is an FRM packet, *send_proc* updates its BP field by adding the current value of *bandwidth_price* to it, before passing it on to the *send* task. The *send* task then looks up the address of the next hop for the packet by using the DST field of the packet to index into the *rtab* routing table and obtain a pointer to the packet’s destination address. The packet is then sent to its destination via the *send()* socket function.

4 Performance Measurements

4.1 Experimental Network

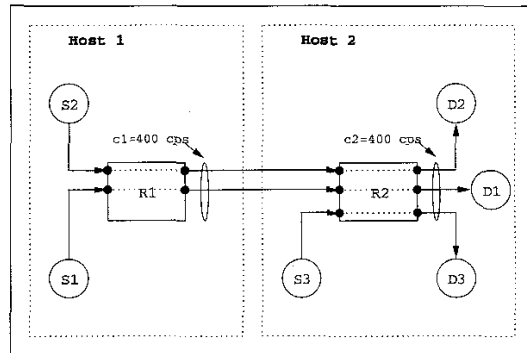


Figure 3: Topology

Our testbed consists of two routing nodes and three source-destination pairs, as depicted in figure 3. It is important to note that in this experiment it is the CPU processing that is the bottleneck and not the physical transmission capacity. Thus, the “links” in our system are the routing processes

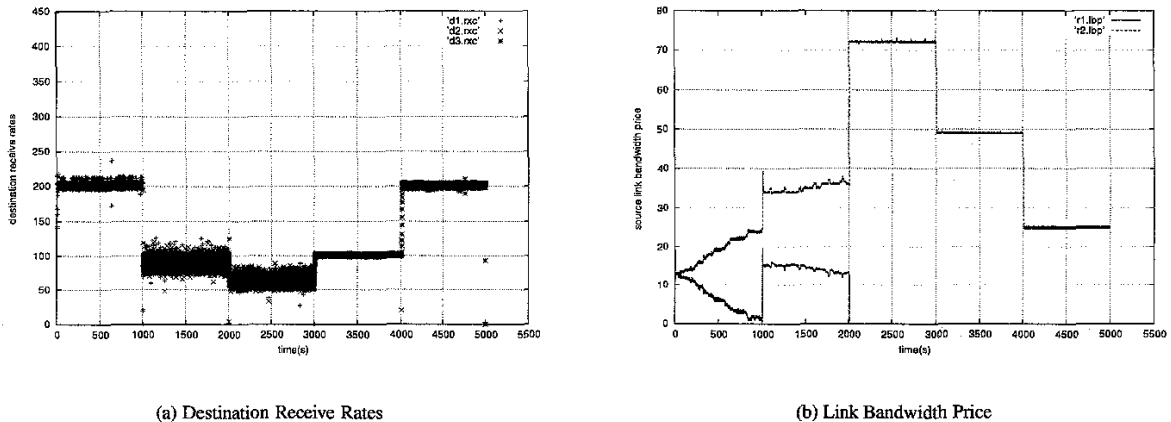


Figure 4: Results

$r1$ and $r2$. However, the dynamics of the system are the same.

4.2 Results

The aim of our experiment was to, firstly, verify that our algorithm worked in the manner predicted by theory, and secondly provide us with insight into the performance of the software. We ran a 5000 s scenario using the network topology shown in figure 3. The long run time ensured that we had sufficient profiling data to provide an accurate representation of the client and router programs' dynamic call graphs. The topology we chose enabled us to observe the way in which bandwidth was distributed between single hop and double hop connections. Each source transmitted data for a total of 3000 s with their starting times staggered by intervals of 1000 s: source 1 started transmitting at time 0, source 2 at time 1000 s, and source 3 at time 2000 s. Staggering the source starting times enabled us to observe the behaviour of OFC algorithm as the system entered and left congestion. The utility functions of the sources was set to $a \log(1 + x_s)$, where a was set to 1×10^4 . The step size γ used by the router to adjust its link prices was set to 1×10^{-2} . Client applications as well as routers dump receive/transmit statistics to file every 500 ms. The routers' also calculate a new link bandwidth price every 500 ms. The target bandwidth was set at 400 packets per second (1.6 Mbps).

Our results are in two parts: we first look briefly at the behaviour of the algorithm (see [4] for more), and then examine the performance of the software

4.2.1 Algorithm behavior

Figure 4(a) shows the raw destination receive rates for each source. These results were obtained from the log file statistics, and record the total number of packets received at the destination for the last 500 ms. We can see that the sum of the traces is constant at 200 packets per interval (400 packets per second) which is the target value set at the routers. We can also see that the destination receive rates vary in accordance with the changes in bandwidth price depicted in figure 4(b). Note that from 2000 s to 3000 s each destination is receiving data at the same rate, and that the longer connections s1-d1 and s2-d2 are not discriminated against. This is because link 1 has zero bandwidth price and is not a bottleneck.

Figure 4(b) shows the link bandwidth prices versus time. We can see that the prices change quite quickly in response to sources commencing/completing transmission. Note that, e.g., during the first 1000s, though the two link prices are different and time-varying, their sum remains constant, being equal to the marginal utility of source s1.

The experimental results that we have obtained from our implementation agree exactly with those predicted by theory and simulation studies. On-going work focuses on more complex scenarios, and scenarios with larger feedback delays.

4.2.2 Software Behaviour

We chose three representative processes to profile: a source, destination and a router. Tables 1, 2 and 3 show cutdown versions of the dynamic call graphs for the *Proc* thread for each of the chosen processes. We have only included the major tasks and the *_select* system call in the

Table 1: S1 Dynamic Call Graph Profile

% Time	Calls	Function
62.79	30004590	_select
12.06	29095360	send_proc
10.40	721188	send
2.73	1	Proc
0.14	22446	recv_proc
0.12	22446	recv

Table 3: R1 Dynamic Call Graph Profile

% Time	Calls	Function
29.09	2547969	_select
23.64	1269587	send
13.63	1269587	recv
10.55	1269587	recv_proc
1.32	1	Proc
0.30	1269587	send_proc

Table 2: D1 Dynamic Call Graph Profile

% Time	Calls	Function
69.11	746107	_select
17.19	718229	recv
6.14	718229	recv_proc
0.79	22446	send
0.49	1	Proc
0.18	22609	send_proc

dynamic call graph. The first column of each table shows the percentage of total program execution time accounted for by the function named in the third column and its descendants. The second column of each table gives the total number of times the function is called throughout the programs' execution.

The `_select` system call implements the multiplexing IO feature mentioned in section §3.2. It suspends the processes until the interface specified is ready to receive or transmit data. This saves valuable CPU cycles and increases the throughput of the system. We can see from the tables that `_select` accounts for more program time than other calls, consistent with our design.

We can also see from Table 1 that the `send` and `send_proc` functions account for a large proportion of the source execution time (10.40% and 8.79% respectively), whereas the `recv` and `recv_proc` functions account for significantly less (0.12% and 0.14% respectively). For the destination process, Table 2 shows that the situation is reversed, reflecting the assymmetric nature of the data transfer.

In the source and destination end systems the proportion of time spent in the `_select` system call is much larger than for the router. This is because the router handles a much higher volume of traffic, and hence spends less time waiting to receive/send packets, and more time receiving and sending packets.

5 Conclusion

In this paper, we have presented a rate-based flow control algorithm that has a sound theoretical base as well as being simple to implement. We have described an implementation of this algorithm on an IP testbed, and we have presented performance measures of the software implementation.

References

- [1] S. Floyd. TCP and Explicit Congestion Notification. In *ACM Computer Communication*, volume 24, October 1994.
- [2] S. Keshav. *An Engineering Approach to Computer Networking*. Addison-Wesley, 1997.
- [3] David E. Lapsley. The OFC Protocol. <http://www.ec.mu.oz.au/pgrad/lapsley/ofc.html>, 1998.
- [4] David E. Lapsley and Steven H. Low. An optimization approach to reactive flow control, II: Implementation and Performance. Submitted for publication, 1998.
- [5] David E. Lapsley and Steven H. Low. An optimization approach to ABR control. In *Proceedings of the ICC*, June 1998.
- [6] Steven H. Low and David E. Lapsley. An optimization approach to reactive flow control, I: Algorithm and Convergence. Submitted for publication, 1998.
- [7] K. K. Ramakrishnan and R. Jain. A binary feedback scheme for congestion avoidance in computer networks with a connectionless network layer. *Proceedings of SIGCOMM'88, ACM*, August 1988.
- [8] S. Sathaye. *Traffic Management Specification v 4.0*. ATM Forum Traffic Management Group, October 1996.
- [9] W. Richard Stevens. *UNIX Network Programming*, volume 1. Prentice-Hall, 2 edition, 1998.